National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

The University of Alberta

A HYBRID STRUCTURE FOR THE REPRESENTATION OF SPATIAL DATA

by

Ⓒ Zhou XiaoYou

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-45636-1

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Zhou XiaoYou

TITLE OF THESIS: A Hybrid Structure for the Representation of Spatial Data

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1988

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) [illegible]

Permanent Address:
Harbin Shipbuilding Engineering Institute
Harbin, Heilong Jiang, China.

Dated [illegible]

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Hybrid Structure for the Representation of Spatial Data** submitted by **Zhou XiaoYou** in partial fulfillment of the requirements for the degree of **Master of Science.**

Wayne A. Davis
Supervisor

[illegible]

[illegible]

[illegible]

Date [illegible] 88

To My Wife and My Parents

ABSTRACT

This Thesis introduces a hybrid structure, an R/Q tree, for representing regions, and presents algorithms for manipulation of data encoded by R/Q trees. The proposed structure aims at representing regions efficiently in terms of computational complexity, memory requirements, and extendibility. The following results are presented: The worst case cost of constructing an R/Q tree is bounded by $O(N_b + N^2/M)$. Where: N_b is the number of maximal linear quadtree blocks in the image, N is the number of the objects in the image, and M is the number of entries per R-Tree node. The cost of translation and rotation of a region is 1 disk access and T disk accesses respectively, where T is the number of nodes in the tree. In addition, the cost of various region operations such as union, intersection, and other geometrical properties of the regions is comparable with that of linear quadtrees. On the other hand, the structure of R/Q trees make them excellent devices for object oriented spatial search as well. The storage utilization of an R/Q tree is 69% which is the same as for a B-tree. Furthermore, R/Q trees lend themselves to compact structures in terms of storage requirements. Many previous algorithms for finding fixed-radius near neighbor and nearest neighbor can be extended with an R/Q tree to more general cases and implemented efficiently. The point in object query problem can also be easily solved using an R/Q tree.

Acknowledgements

I am indebted for advice and guidance to my supervisor Dr. Wayne A. Davis. His knowledge and experience has guided me throughout this research.

I would like to thank the members of my examining committee, Drs. S. Cabay, B. Joe, and A. Peterson, for their valuable comments and suggestions and for their time spent in reading this thesis.

I would also like to express my appreciation to the Department of Computing Science for technical and financial support.

Finally, I also would like to thank Duan Zhen Rong, my wife, who contributed in a great many ways, and my parents who were encouraging and supportive at all times during this endeavor.

Table of Contents

List of Figures

Chapter 1

INTRODUCTION

This thesis is concerned with a data structure for representing regions. As regions are distributed in 2 (or more) dimensions, spatial data structures which aim at representing regions effectively in terms of computational complexity, memory space requirements and extensibility[41-43] are receiving increasing attention from researchers in image processing, pattern recognition, cartography, computer graphics, geographic information systems and robotics[52].

There are a number of methods for representing regions[14,52] among which are vector, raster, and block. A chain code representation, also known as the Freeman-Hoffman code, [42] specifies the boundary of a region, relative to a given starting point, as a sequence of unit vectors in 8 principal directions. This method is very compact in terms of storage requirements, and is also very efficient in detecting features of the boundary, such as sharp turns or concavities. However, a chain code representation does not facilitate the determination of properties such as shape, and it is also difficult to perform region set operations such as union and intersection. A scan line representation model organizes the interior of a region by defining a region as a set of blocks of 1 by m rectangles. The primary drawback of this raster data structure is that it is a wasteful use of computer storage due to its tendency of storing redundant data. A block representation is a class of data structures that treats regions as a union of rectangular blocks that may overlap. A data structure called medial axis transformation(MAT) or skeleton[39,51] represents regions as a union of maximal square blocks by specifying their centers and radii. Data structures such as binary trees [6,8] represent regions as maximal rectangles, and of particular interest is the quadtree originated by Klinger[26,28].

A quadtree is a variant on the maximal block representation. It requires that blocks be disjoint and have sides of lengths that are powers of two and at standard locations. An advantage of a quadtree is its hierarchical structure which lends itself to a compact representation. It is also quite efficient for a number of traditional image processing operations including: finding centroids, labeling connected components, computing perimeters and set properties[24,56]. Furthermore, it is a dynamic structure in that it adapts its shape gracefully in response to the external demands such as insertion and deletion of quadrants and can be easily converted into other representations such as chain codes, rasters, binary arrays, and medial axis transforms [48-51].

Early work on quadtrees represented the hierarchical relations among quadrants and subquadrants through the use of explicit pointers. However, in many real world applications, images may be so large that the space requirements of their quadtree representation exceeds the amount of available memory. The result is that the image must be stored on disk with portions processed in main memory as needed. There are two reasons why the traditional pointer based quadtree structure is considered inappropriate for such applications. First, a large portion of the pointer based quadtree storage space is taken up by GRAY nodes and pointers. Secondly, as pointer based quadtrees do not take memory paging into account [23], the need of following a chain of pointers from the root to the desired node may lead to a larger number of page accesses than are acceptable in an interactive environment.

In an effort to overcome the problem and to further reduce the storage space, there have been studies to represent a quadtree in a linear structure and use a B-tree file structure in organizing the data [1,2,53]. While a hybrid structure of a quadtree and a B-tree for region representation is a significant improvement over the original pointer based quadtrees, it does not conveniently support high level object oriented search [23,47]. However, the problem of how to organize data in external memory so as to facilitate subsequent retrieval, especially high level object oriented search, is crucial in most large databases used in geographic information systems, computer aided design (CAD), etc.

Efficient spatial search requires a more advanced indexing technique than a B-tree. One file structure which is an excellent speed-up device for high level object oriented spatial search is an R-tree [23,47]. The distinct feature of an R-tree is that non-atomic objects are treated as individual entities, upon which the index is constructed, thus being convenient for high level spatial search. In addition, R-trees are basically a natural extension of B-trees [23,47]. The logarithmic height of their tree structure and their the dynamic nature make them very attractive. The primary disadvantage of R-trees is that they do not address the issue of how to structure low level primitives.

From this perspective, this thesis develops a data structure with the following features: (1) The data structure should consider both a high level organization and a low level representation, i.e., the structure should facilitate subsequent operations for both high and low level spatial data. (2) It should be compact. (3) It should be dynamic and able to adapt its structure automatically in response to external demands such as insertion and deletion.

The thesis begins with an introduction to the problem of the representation of regions and the recursive decomposition of images is briefly reviewed along with quadtree and linear quadtree representations. A dynamic index structure for spatial searching, an R-tree is also presented. In Chapter 3, a hybrid structure for region representation is proposed and shown to be superior to a linear quadtree structure with a B-tree incorporated. Chapter 4 investigates operations on images using the proposed structure. A number of algorithms are presented in detail to effectively support these operations. The last chapter presents the conclusions and suggestions for further research.

Chapter 2
BACKGROUND

This chapter covers two topics which are foundations of the rest of this thesis. Section 2.2 contains some basic definitions and terminology for region representations and introduces quadtrees as a data structure for region representation with emphasis on a linear quadtree. The issues of spatial data searching and data structures for handling multi-dimensional data are discussed in Section 2.3. In particular, R-trees are described in detail.

2.1. Definitions and Notation

Definition 2.1: An image is a 2^n by 2^n array of unit square pixels each of which can assume one of 2^c values, where n is called the resolution parameter of the image.

Definition 2.2: An image is called a binary image when its pixels assume either 1 or 0 values. A pixel is said to be BLACK if it has the value of 1, otherwise it is said to be WHITE.

Without loss of generality, only binary images will be considered in this thesis since all the algorithms can be easily extended to nonbinary images.

Definition 2.3: The region of a binary image is composed of all BLACK pixels, and the background of the region is composed of all WHITE pixels.

Example: The region shown in Figure 2.1 is represented by the 2^3 by 2^3 binary array in

Figure 2.2, where 1 and 0 correspond to BLACK and WHITE pixels, respectively.

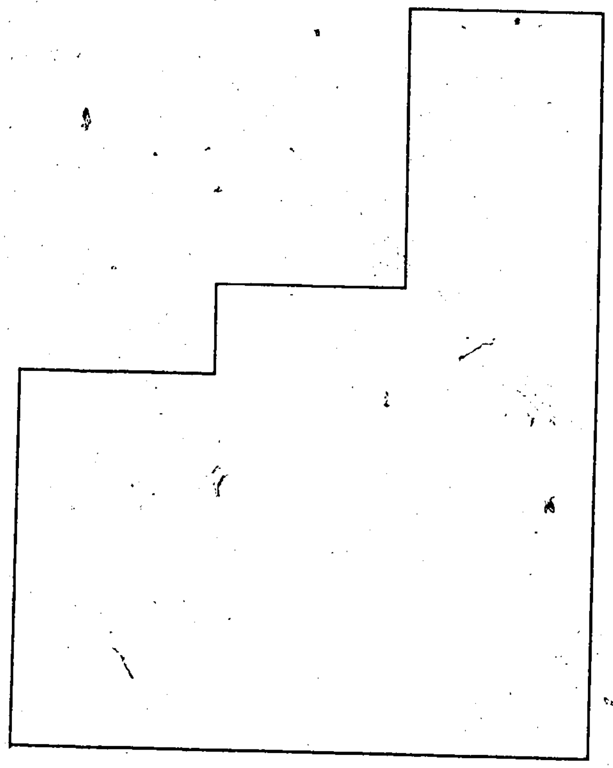


Figure 2.1 A Region

7	0	0	0	0	1	1	0	0
6	0	0	0	0	1	1	0	0
5	0	0	0	0	1	1	0	0
4	0	0	1	1	1	1	0	0
3	1	1	1	1	1	1	0	0
2	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	0	0
	0	1	2	3	4	5	6	7

Figure 2.2 The Region in Binary Array Representation

Definition 2.4: Let (i,j) represent the location of a pixel p in a given image, where i and j are the respective column and row positions of p. P has four horizontal and vertical neighbors located at: (i-1,j), (i,j-1), (i,j+1) and (i+1,j). These pixels are called the 4-neighbors of p, and are said to be 4-adjacent to p.

Definition 2.5: A path from pixel p to pixel q is a sequence of distinct pixels, $p=p_0, p_1, \ldots, p_n=q$, such that p_m is 4-adjacent to p_{m-1}, where $1 \le m \le n$.

Definition 2.6: If p and q are two BLACK pixels of a region, then p is said to be connected to q if there is a path from p to q consisting entirely of pixels of the region.

Definition 2.7: For any BLACK pixel p, the set of pixels connected to p is called a connected component of the region. If a region has only one component, then it is called "connected".

Clearly, a naive way for representing an image is to use an array of pixels. This method is inefficient both in terms of space and time complexities. Since most images are composed of a collection of regions, it is more efficient to represent an image by specifying its regions. The most studied approach to region representation is based on the successive subdivision of the image array into four equal sized quadrants which is similar to the divide and conquer method [3]. If the array does not consist entirely of 1's or entirely of 0's, it is then subdivided into quadrants, subquadrants, etc., until blocks are obtained that consist entirely of 1's or entirely of 0's, i.e., each block is either contained entirely in the region or is disjoint from it. For example, Figure 2.3 is the decomposition of the region shown in Figure 2.1. In this case the image is decomposed into 13 blocks and the maximal blocks A, B, C, D, E and F are totally contained in the region whereas the remaining blocks are disjoint from it.

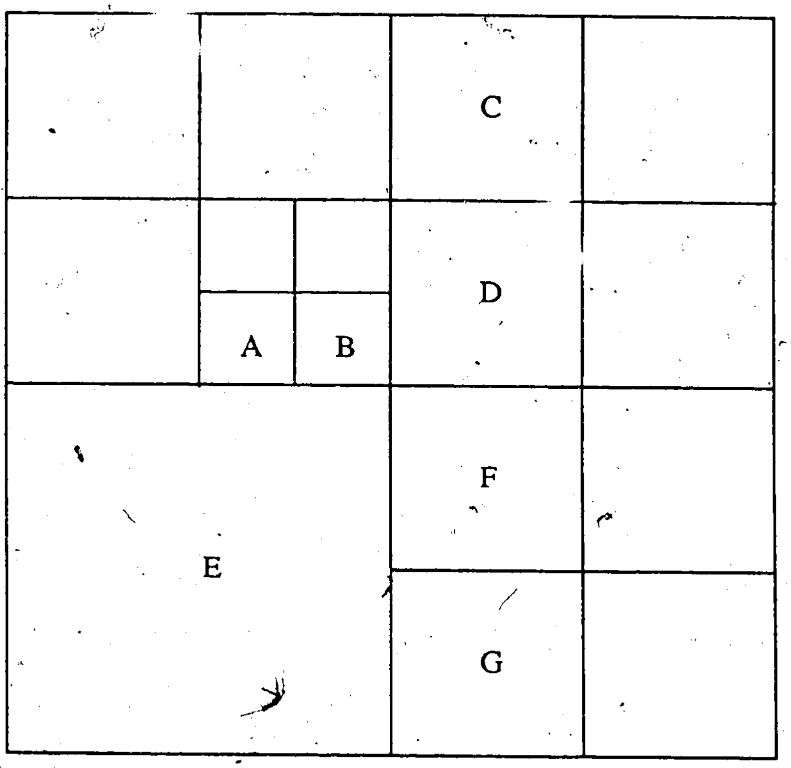


Figure 2.3 Regular Decomposition of the Region in Figure 2.1

The recursive decomposition of an image produces blocks that must have standard sizes (power of 2) and positions. It is clear that representing a region in terms of blocks is much more compact than by individual pixels. To do so, the following definitions are necessary:

Definition 2.8: A block is said to be: BLACK if it contains only BLACK pixels, WHITE if it contains only WHITE pixels, and GREY if it contains both BLACK and WHITE pixels.

The four sides of a block are referred as to its North, East, South and West sides, or N, E, S and W for short. And let **OPSIDE(T)** be the side opposite to T, e.g., **OPSIDE(E)=W.**

Definition 2.9: Two blocks P and Q are said to be 4-adjacent along the side T of P if side T of P touches the **OPSIDE(T)** of Q.

Definition 2.10: BLACK blocks P and Q are said to be connected if there exists a path consisting entirely of BLACK pixels from a pixel of P to a pixel of Q.

2.2. Quadtrees: a Hierarchical Data Structure for Region Representation

This Section deals with the terminology, definitions and an overview of quadtree data structures[12].

2.2.1. The Quadtree Region Representation

The recursive nature of the previous decomposition process facilitates a hierarchical type of data structure for its representation. One such data structure called a quadtree was proposed originally by Klinger [26,28].

Definition 2.11: A quadtree is a directed edge and node labeled tree in which

(1) Leaves are labeled from the set {WHITE,BLACK}.

(2) Nonleaves are labeled GREY.

(3) Edges are labeled from the set {NW, NE, SE, SW}.

(4) Each node is either a leaf or has four children, with the four out-going edges labeled differently.

(5) At least two leaves of the same parent must be assigned different labelings.

As an example, Figure 2.4 demonstrates the quadtree for the region in Figure 2.1, where the symbols ○ , □ , and ■ represent GREY, WHITE and BLACK nodes, respectively. Note that the terms block and node will be used interchangeably throughout.

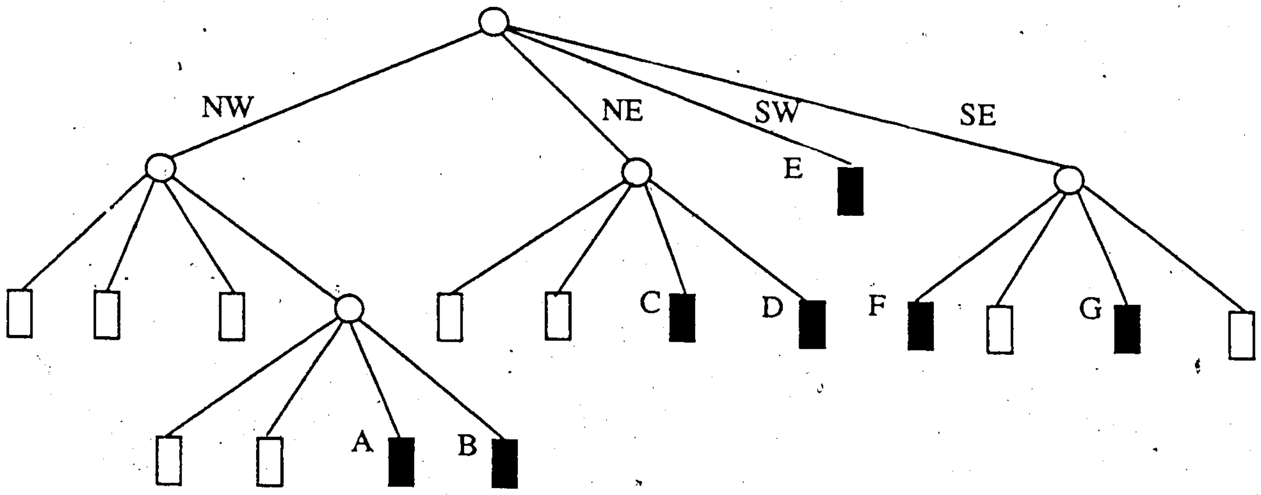


Figure 2.4 The Quadtree of the Region in Figure 2.1

Quadtrees structured with explicit pointers are called pointer based quadtrees. A pointer based quadtree stores each node as a record containing six fields. The first five fields contain pointers to the nodes FATHER and four SONs each of which corresponds to the four quadrants. The sixth field describes the contents of the block of the image which the node represents.

Let N be the total number of nodes in a quadtree. B the number of BLACK nodes, W the number of WHITE nodes, and G the number of GREY nodes. Then N=B+W+G. Knuth [30] has shown that the following relations are true for quadtrees:

$$N=4G+1=(4(B+W)-1)/3,$$

$$B=3G-W+1,$$

$$W=3G-B+1,$$

$$G=(B+W-1)/3.$$

In other words, there are nearly one third as many GREY nodes as there are BLACK and WHITE nodes together.

There are two reasons that a pointer-based quadtree structure is considered inappropriate for many applications, especially large images. First, a considerable amount of overhead is associated with it since a large portion of the pointer-based quadtree's storage space is taken up by GREY nodes and pointers. Furthermore, only GREY nodes effectively use their four pointers to their four sons, while BLACK and WHITE nodes have pointers to empty records. Second, individual leaf nodes within a pointer-based quadtree are located by following a chain of pointers from the root to the desired node, which can require many pointer references for large images. As there may be little relationship between the ordering of nodes in the tree and their ordering on the disk, this can lead to an intolerable number of disk accesses when searching and updating the tree.

Two simple modifications can be made to the traditional pointer-based quadtree structure to alleviate the above problem. Either introduce two different kinds of records for terminal nodes and nonterminal nodes or implement a quadtree as a child-sibling binary tree. As an example of the later case the quadtree in Figure 2.4 is represented as a child-sibling binary tree in Figure 2.5. Both methods will greatly reduce the number of pointers per node from five to two on the average[12]. However, the first approach results in a quadtree that is no longer composed of the same record types and therefore operations must be performed differently when nodes differ and the second approach requires two steps of references on average in reaching a father or child node, thus worsening the second previously mentioned difficulty.

Totally eliminating quadtree pointers together with removing all GREY and WHITE nodes is a better solution. It will also lead to a smaller number of nodes relative to the total number of nodes in the quadtree. The resulting data structure is called a *linear quadtree* which is reviewed in next section[19].

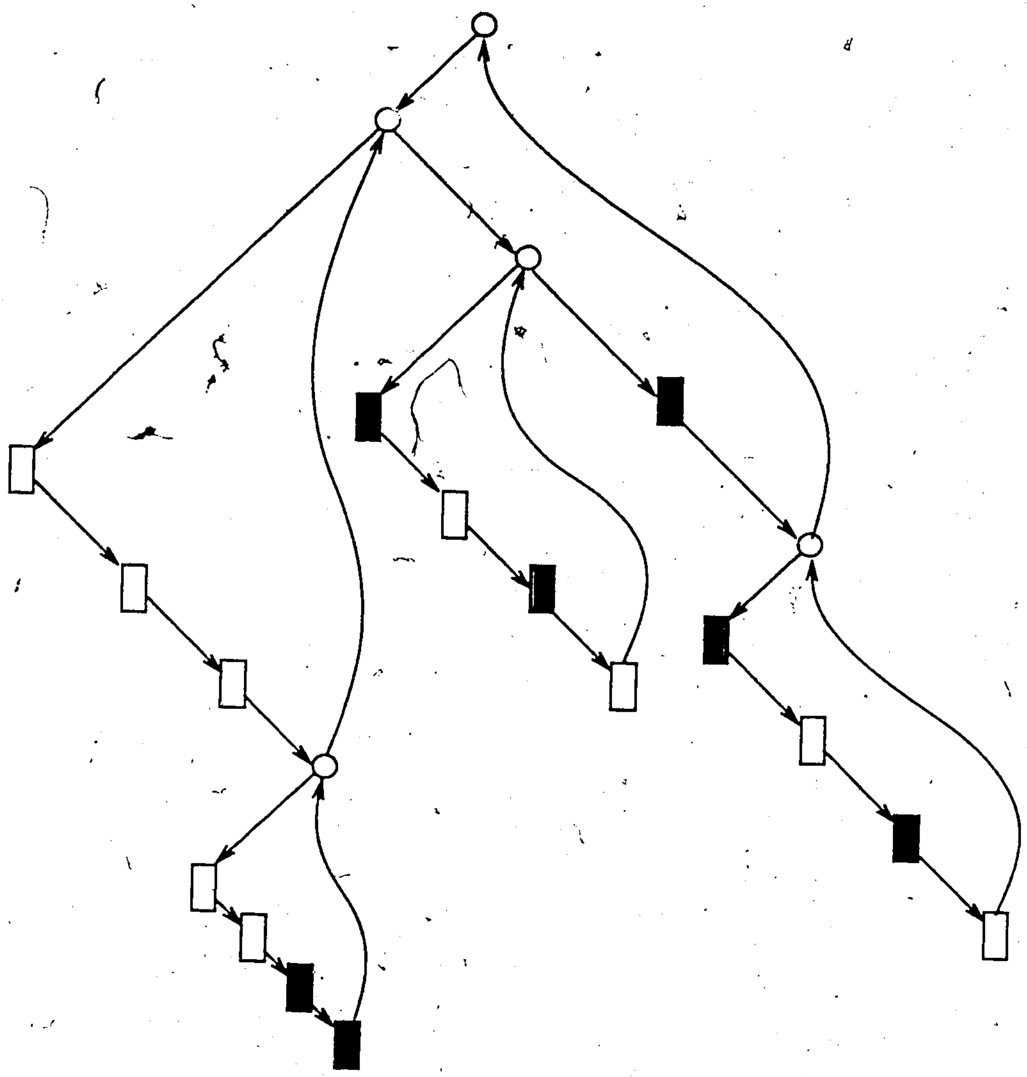


Figure 2.5 A Child-sibling Binary Tree of the Quadtree in Figure 2.4

2.2.2. Pointerless Quadtrees

As is discussed in the last section, the problem with a pointer-based quadtree is that it has a considerable amount of overhead associated with it. To overcome this drawback, a data structure was proposed [19] called a linear quadtree which is pointerless and stores only BLACK nodes.

A *Linear quadtree* encodes its BLACK nodes by a base 4 number, called a location code or key, that corresponds to a sequence of directional codes that locates the leaf along a path from the root of the quadtree. In addition, the length of the directional codes specifies the level of the decomposition process.

Suppose the direction codes for SW, NW, SE, and NE are 0, 1, 2, and 3, respectively.

As an example, the linear quadtree representation for the region in Figure 2.1 is the following sequence:

120, 122, 31X, 30X, 0XX, 21X, 20X[1]

which correspond to the BLACK nodes A, B, C, D, E, F, and G, respectively, of Figure 2.3.

Another effort for a pointerless quadtree is referred to as a *DF-expression* by introducing the BNF notation into the quadtree structure [52]. Thus, an internal node in a quadtree corresponds to an 'operator' in a DF-expression, and the whole quadtree is represented in the form of a traversal of the nodes of the quadtree.

[1] Where X represents a don't care.

For example using a DF-expression, the quadtree in Figure 2.4 is encoded as follow

((WWW(W [illegible] B(CWDWE(FWGW.

where '(' represents an internal node

2.2.3. The Modified Linear Quadtree Approach

Although the linear quadtree as originally proposed has many advantages over a pointer-based quadtree, it suffers from the quaternary encoding of its nodes and is less efficient in supporting many algorithms for the manipulation of regions, see [12,19,52].

On the other hand, a DF-expression, although more efficient spacewise than a linear quadtree, suffers from its nonlinearity and is less efficient in terms of computational complexity.

Mathematically, a block can be uniquely represented as a 3-tuple $<I_1 I_2 s>$, where I_1 the position of the block along the X-axis ; I_2 is the position of the block along the Y-axis, and s is the parameter for specifying the size of the block. Obviously, to encode the blocks of a quadtree by a set of 3-tuples is awkward since to retrieve a data item, a search must be performed in a multi-dimensional parametric space.

The central idea of a linear quadtree is to represent a region as a collection of BLACK nodes each of which corresponds to a quaternary code key. Thus, the original three dimensional parametric space is transformed into a one dimensional key space. Such a transformation is made possible since blocks, which are resultants of the regular subdivision, possess many qualities such as: standard sizes and positions, disjoint, etc. A general technique to transform a point in a multi-dimensional space into a one dimensional key space is called **Morton Sequencing** [37].

In order to introduce Morton Sequencing, the following conventions, which will be adopted throughout this thesis, are necessary:

Definition 2.12: For two integers I and J given by

$$I=\sum_{i=0}^{n-1}(I_i*2^i), \text{ and } J=\sum_{i=0}^{n-1}(J_i*2^i), \text{ where } I_i J_i \varepsilon \{0,1\},$$

$$\text{SHUFFLE}(I,J)=\sum_{i=0}^{n-1}(I_i*2+J_i)*4^i.$$

Definition 2.13: For an integer $S=s_{2n-1}\cdots s_0$, where $s_i\varepsilon\{0,1\}$,

$$\text{EVEN}(S)=\sum_{i=0}^{n-1}s_{2i}*2^i \text{ and } \text{ODD}(S)=\sum_{i=0}^{n-1}s_{2i+1}*2^i.$$

The above three functions are used frequently, therefore **SH**, **EV**, **OD** will be used as abbreviated versions of SHUFFLE, EVEN and ODD, respectively.

Given a 2^n by 2^n image, there are a number of ways to assign consecutive integers to the pixels. Morton Sequencing[37], however, is the best means of capturing the nature of the recursive decomposition described in the previous chapter.

Definition 2.14: Morton sequencing is the assignment of integers to an array of 2^n by 2^n pixels, such that the integer assigned to a pixel p with coordinates (I,J), where I and J are the column and row position respectively, is **SH**(I,J).

Example: Figure 2.6 shows a 2^3 by 2^3 Morton sequence.

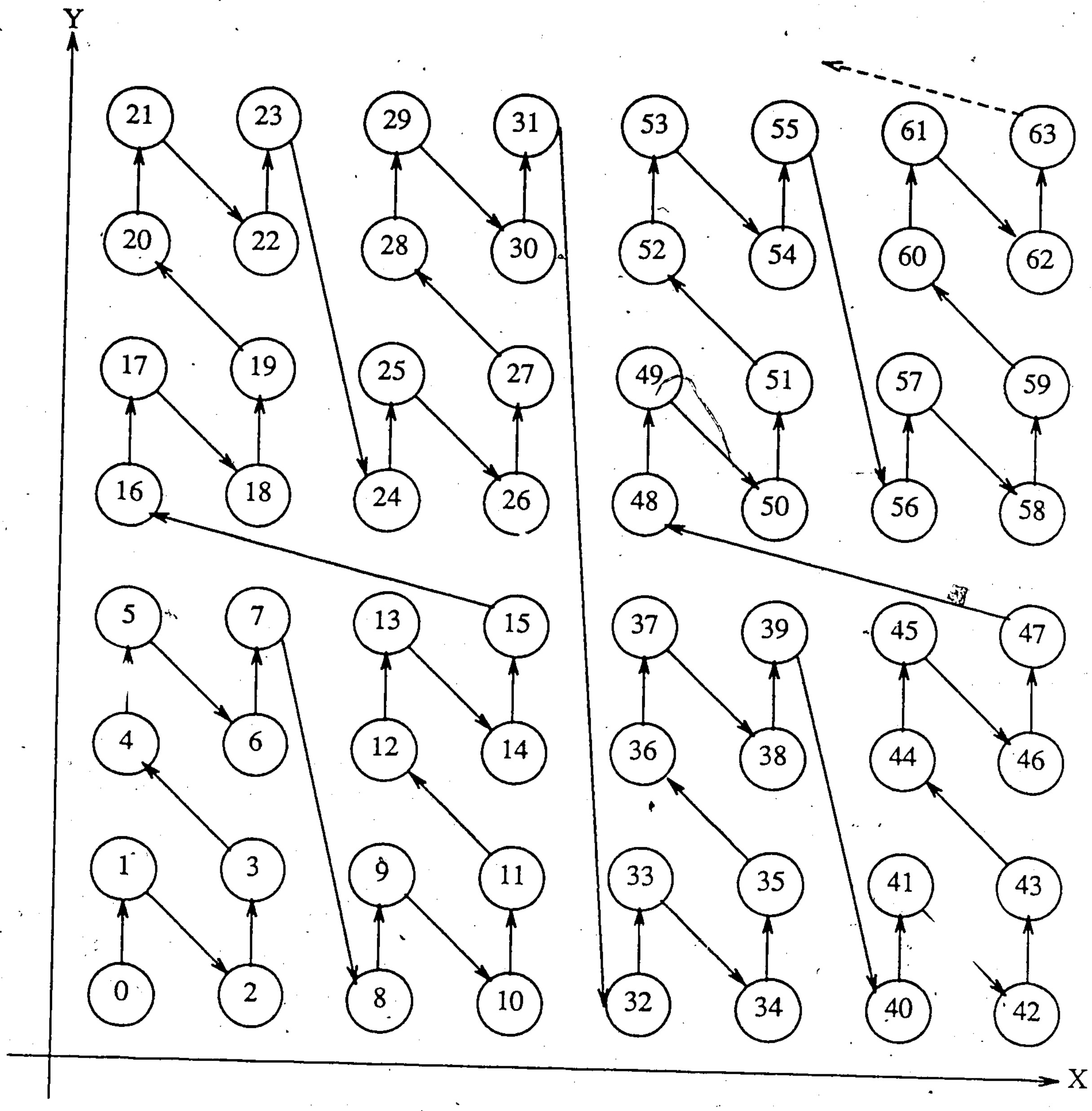


Figure 2.6 Morton Sequence.

Using a Morton sequence, there are a number of ways to represent a block obtained by the recursive decomposition method. For example, the following method can be chosen:

Definition 2.15: The key of a block or node with 2^s by 2^s pixels is the key of its left bottom pixel, where s is called the resolution parameter of the block.

It is now easy to show that the two-tuple $<K,s>$ uniquely represents a block, where K and s are the key and resolution parameter of the block, respectively.

Definition 2.16: Given a block Q, its subblock with label i is called the ith subblock of Q and denoted by Q_i if Q_i is obtained by one subdivision of Q, where $i\varepsilon\ \{0,1,2,3\}$.

The quadrant labeling shown below will be assumed.

1	3
0	2

Example : For the image in Figure 2.2, the entire block E is represented as <0,2>, and G by <32,1>.

The following lemmas[12] grasp the essence of Morton sequencing:

Lemma 2.1: For any two nodes $<K_1,s_1>$ and $<K_2,s_2>$ where $<K_1,s_1>\varepsilon Q_i$ and $<K_2,s_2>\varepsilon Q_j$ for some node Q, if i<j then $K_1<K_2$.

Proof: See [12].

Lemma 2.2: For any two nodes $<K_1,s_1>$ and $<K_2,s_2>$ where $s_1>s_2$, then either $<K_1,s_1>$

contains $<K_2,s_2>$, or the intersection of $<K_1,s_1>$ and $<K_2,s_2>$ is empty.

Proof: See [12].

Lemma 2.3: For any two nodes $<K,s>$ and $<K',s'>$ such that $K \neq K'$, $<K',s'>$ is contained in $<K,s>$ iff $K<K'<K+4^s$.

Proof: Let the coordinate of the SW corner of block $<K,s>$ be $<X,Y>$ and the coordinate of the SW corner of block $<K',s'>$ be $<X',Y'>$.

=> Since $K<K'<K+4^s$, the SW corner of block $<K',s'>$ is contained in block $<K,s>$, by the definition of Morton sequence. Again, since $K<K'$, $<K,s>$ cannot be contained in $<K',s'>$. Therefore, $<K',s'>$ is contained in $<K,s>$, according to **Lemma 2.2.**

<= if $<K',s'>$ is contained in $<K,s>$, then the coordinate of the SW corner of the block $<K',s'>$ is contained in $<K,s>$. Thus $K'=\mathrm{SH}(X,Y)<K'<K+4^s$ is true.

Q.E.D.

Lemma 2.4: Let $B_1,...,B_n$ where $B_i=<K_i,s_i>$ be a sequence of blocks ordered in ascending key order which represents a region, and let $<K,s>$ be a search area in block form. If the intersection of $<K,s>$ and the region $B_1,...,B_n$ is nonempty then there exists a sequence of consecutive blocks $B_a,...,B_b$ where $1 \leq a \leq b \leq n$, which intersect $<K,s>$. Furthermore for any block $<K_j,s_j>$, where j<a or j>b, the intersection of $<K,s>$ and $<K_j,s_j>$ is empty.

Proof: First, Let $K \neq K_i$, for i=1,...,n. If the intersection of $<K,s>$ and $B_1,...,B_n$ is nonempty, then there exists one block B_i such that the intersection of $<K,s>$ and B_i is nonempty. By **Lemma 2.2**, either $<K,s>$ contains B_i or vice versa.

Suppose e is the integer which satisfies:

$$\begin{cases} K < K_{e+1} & \text{if } e=0 \\ K_e < K < K_{e+1} & \text{if } 0<e<n \\ K_e < K & \text{if } e=n. \end{cases}$$

Case 1: $K_e < K < K_e + 4^s$ and e>0.

Block $<K,s>$ is contained in block B_e, by **Lemma 2.3.** Again, for any block $B_i, i \neq e$, the intersection of B_i and $<K,s>$ is empty (if B_i intersects $<K,s>$ then B_i also intersects B_e which is impossible since blocks obtained by recursive subdivision are basically disjoint from one another). Thus a=b=e.

Case 2: $K < K_{e+1} < K + 4^s$ and e<n.

Block B_{e+1} is contained in block $<K,s>$. Let h be the largest integer which satisfies $K_{e+h} < K + 4^s$, then block $B_{e+1},\ldots,B_{e+h}$ are all contained in $<K,s>$, by **Lemma 2.3.**

If there is a block B_i which overlaps $<K,s>$ and $i \leq e$ then $<K,s>$ is totally contained in B_i by **Lemma 2.2**, using the fact that $K_e < K$.

However, this implies that B_i contains B_{e+1} which is impossible.

On the other hand, for any block B_i, where i>e+h, the intersection of B_i and $<K,s>$ is empty since $K_i \geq K + 4^s$

Therefore, a=e+1 and b=e+h.

Alternatively, if there exist a integer j, $1 \leq j \leq n$, such that $K_j = K$ is satisfied, i.e., block B_j intersects $<K,s>$.

Suppose block $<K,s>$ is contained in block B^j, then similar analysis of the case 1 yields, a=b=j.

Otherwise, suppose h is the largest integer which satisfies $K < K_{j+h} < K+4^s$, then a=j and b= j+h, according to the analysis of the case 2.

Q.E.D.

Suppose q=$q_0, \ldots, q_{n-1}$ is the linear quaternary code of a block, then the corresponding Morton code $<K,s>$ can be calculated as follows:

$$s=\begin{cases} 0 & \text{if } q \text{ does not contain the don't care sign } X \\ n-m+1 & \text{if the first don't care is at } m\text{th position} \end{cases}$$

$$K=\sum_{i=0}^{m-1} q_i * 4^{-i+n-m+1}.$$

For instance, the linear quaternary code of block D in Figure 2.3. is 30X, while its Morton code is $<3*4^2+0*4,3-3+1>$ which is equal to <48,1>.

On the other hand, given a Morton code, the corresponding linear quaternary code can also be easily calculated.

It is not all that surprising, that there are variations of linear quadtrees [12,19]. For instance, if the blocks in a region are encoded as linear quaternary codes and are sorted in ascending key order, then a set of lemmas which parallels **Lemma 2.1-2.4** is also derivable from the linear quadtree; both representations posses some properties such as being easy to change in resolution in their representation (or to scale a block by a power of 2), being able to convert efficiently to and from quadtree representation, etc.

Although, there are similarities between the two representations, the linear quadtree as specified in [12] has two advantages over its counterpart[19]: space efficiency and improved execution time. To see the first advantage necessitates a comparison between the storage requirements of the two encoding methods. As reported in [19], each

BLACK node needs 3n bits. By contrast, the proposed encoding scheme requires (2n+logn) bits for each BLACK node, where 2n bits are used for storing the key and (logn) bits for the resolution parameter of the node. The second advantage is achieved by using the explicit key and resolution parameters in developing an efficient algorithm as shown in [12]. In fact, many algorithms reported in [12] are shown to be superior to their counterparts where the region is represented as a traditional pointer-based quadtree and as the linear quadtree originally proposed. On the other hand, the linear quaternary encoding scheme is invariant with the coordinate system, while the other encoding scheme is dependent on the coordinate system since the Morton sequencing will be changed if the coordinate system is different. This implies that the transformation between two different coordinate systems requires that the whole item be changed if the alternative format is used. However, a linear time algorithm can be used to perform the translation [12].

2.2.4. File Schemes for Linear Quadtrees

An important fact of linear quadtrees is that although they are very efficient in terms of space complexity, more often than not they are too big to be kept in main memory [1,13,53], and most algorithms for manipulating a linear quadtree [12,20,64,53] require both sequential and random retrieval from the linear quadtree file.

Thus, given a sequence of sorted blocks, some means must be found to organize it so that insertions, deletions, and information retrieval may be performed efficiently.

To be effective, a file organization scheme should allow both random and sequential access to records in the file and be able to support dynamic file maintenance. With the advent of direct access storage devices, several efficient file organization schemes have emerged (a survey of the basics is contained in [52]). Among them, balanced tree organization and hashing schemes have been the most successful.

There have been studies to represent a quadtree as a linear quadtree and use a B^+-tree file structure in organizing the data [1,2,53]. An important feature of B-trees is that they gracefully adapt their shapes in response to insertion and deletion of records and support retrieval of a record with $O(logn)$ disk accesses, where n is the number of records in the file. The primary disadvantage of B-trees is that there is no implied connection between data buckets in physical storage and regions in the search space and a record in a B-tree is located by following a chain of pointers from the root to the desired node. Thus, although the logarithmic cost of a B-tree is attractive compared with the $O(n)$ of sequential files, it is still far greater than the $O(1)$ of direct access methods.

Hashing methods[13], on the other hand, organize data by exploring relationships between data buckets in physical storage and data in the search space. If a file is static, hashing allows a record to be retrieved with one disk access, in general. Traditional hashing methods, however, are not useful for two reasons:

(1) The storage allocation for the hash table is static. That is the size of a file must be estimated in advance and the storage space must be allocated for the whole file at once. Thus, a high estimate of the data volume results in waste space while a low estimate of the data volume results in either a costly reorganization of the whole file through rehashing or the attachment of overflow buckets, which slowly degrades the $O(1)$ access time characteristic of hashing toward $O(n)$ in the worst case.

(2) The sequential retrieval of data in key order is rather difficult with traditional hash-

ing methods since little knowledge is known as to where the next key is. Given a sequence of spatial data encoded as a linear quadtree, the result of **Lemma 2.4** implies that range search cannot be performed efficiently unless the retrieval of data in ascending key order can be performed efficiently. In other words, traditional hashing methods do not conveniently lend themselves to range search.

Recently, however, several dynamic hashing approaches have been developed. Dynamic hashing schemes can be categorized into two classes. The first class of dynamic hashing has a file space expansion operation when an overflow occurs. This class includes virtual hashing by Litwin [33], dynamic hashing by Larson [31] and extendible hashing by Fagin et al. [16]. To maintain the relationship between split buckets and the remaining buckets, indexes are usually used.

The second class of dynamic hashing avoids splitting until the global storage utilization factor exceeds a predefined upper threshold. Therefore, records that overflow are chained in a set of overflow buckets. Then, in case the global storage utilization factor exceeds the threshold, some buckets are split into a large address space. No index is necessary for this class of dynamic hashing. This class includes linear hashing by Litwin [34] and interpolation-based index maintenance by Burkhard [9].

Among various approaches of particular interest, extendible hashing or its variations is chosen as the main implementation paradigm because it is neat, easily parametrizable and completely determined by one hash function which may or may not be order preserving [58]. For example, an extendible hashing with a buddy system partition strategy incorporated can be described as follows:

Suppose a hash function h maps the key space K onto an address space A. The distinct feature of the scheme is that h is an order preserving function. That is h splits A into m blocks defined by $m+1$ boundaries, $\alpha_0, \ldots, \alpha_m$ by constructing a hash table of size m to establish the correspondence between data buckets and blocks. To make the hash function dynamic, the hash table must be extendible to have a variable number of variable sized blocks. Thus, if a data bucket overflows due to the arrival of new data, the buddy system is invoked and the address space is halved. As the result, the key space is divided into a regular grid of intervals corresponding to storage areas called buckets. The multidimensional version of this idea is called the cell method [7,8,13]. Since this scheme can organize the data buckets in key order and the partition of search space into cells is always at the midpoint of an interval and alternately perpendicular to the X and Y axes, the shape of cells is compatible with that of quadtree blocks.

The extendible hashing method was intended to retrieve data associated with a given key with two disk accesses: one access to retrieve the desired part of the directory and another to retrieve the data bucket that contains the given key.

If data is distributed uniformly over the study area, the above scheme requires two disk access to locate a record. In the worst case when data is distributed unevenly over the study area, with the above approach, the hash table will become large and unwieldy. In fact, in the worst case, the size of the directory may reach to $O(n)$, where n is the number of records in the file.

Linear hashing methods, on the other hand, do not use an explicit directory. Instead of resolving collisions by splitting the overflow address they resolve the overflow conditions by chaining the overflow address. Bucket splitting is performed in a systematic way, when the global storage utilization factor exceeds a predefined upper threshold[34].

The advantages of a linear hashing scheme are offset by two major disadvantages: unbalanced data bucket occupancy and a possible long chain of overflow buckets. Non-uniform distribution of data may force it to keep many overflow buckets as well as many underfilled primary buckets at the same time. In addition, the overflow chain may degrade the time bound for retrieving data associated with a given key from $O(1)$ towards $O(n)$ in the worst case, where n is the number of records in the file. In addition, a linear hashing scheme does not support order preserving address transformations efficiently [34,58].

A better hashing scheme called the adaptive cell method [13] provides a compromise in the tradeoff between extendible hashing methods and linear hashing based methods, namely, the tradeoff between the overhead of maintaining a large directory and the advantage of balancing bucket occupancy. This method, although it alleviates the problems in both extendible hashing methods and linear hashing methods, does not improve the time bound for retrieving data associated with a given key in the worst case.

In conclusion, the selection of a file scheme is domain dependent. If the distribution of data is nonuniform then a B^+-tree file scheme is the better choice since it supports retrieval of a record with $O(logn)$ disk accesses, where n is the number of records in the file, which is superior to the intolerable $O(n)$ disk accesses which may occur if a hashing method is chosen. However, many simulation results [13,34] show that the probability of this worst case happening in practical data is expected to be exceedingly low. Thus, the above mentioned hashing schemes seem to be preferable in most cases.

2.2.5. Conclusion and Discussions

Four different variations of quadtrees have been presented in this section, namely, the pointer-based quadtree, the linear-quaternary quadtree, the DF-expression, and the modified linear quadtree. The traditional pointer based quadtree is shown to be inappropriate for large images. On the other hand, a linear quadtree represents a quadtree as a sequence of terminal nodes in a specific order, while nonterminal nodes, or even WHITE nodes of the quadtree are omitted, and pointers are replaced by keys (and resolution parameters). Thus the spatial efficiency is greatly improved.

Linear quadtrees also conveniently lend themselves to efficient spatial data processing. For example, when dealing with spatial data retrievals, exact matching is almost impossible. For instance, it is not possible to store all the potential points that may be requested for a point-in-polygon problem. Instead, the point-in-polygon problem is solved by referring to the information in the neighborhood of the given point. Hence spatial search can be characterized as traversals referring to each node and more often than not these references involve information from the node's neighbors. To be effective, it is desired that the data structure for region representation preserves location. That is, blocks that are near each other in the study area should also be near each other in the representation, with a high probability of being stored in the same physical disk block in the case of large images. From **Lemma 2.4**, it is clear, when organized as a sequential file, using either a B-tree scheme or a hashing scheme, linear quadtrees meet this requirement. Furthermore, the topological order of the quadtr[illegible] compatible with the order of the keys of the blocks, from **Lemma 2.1**. When the keys of the leaf nodes are listed in ascending key order, the resulting sequence is in the same order as a depth-first traversal of the quadtree blocks. More important, this feature allows the quadtree topology to be obtained from the keys by using modular arithmetic rather than following a chain of pointers and makes the conversion between a quadtree representation and a linear quadtree representation very simple [12,19].

Advantages of quadtrees are offset by two main drawbacks: firstly, quadtrees do not conveniently lend themselves to high level object oriented searching; secondly, they are shift variant.

The first problem is caused by the regular decomposition process. As a result, objects in the study area are partitioned into small pieces. Thus, to retrieve a whole object it is necessary to search for all the primitives. To alleviate the drawback, several structures have been reported in the literature. For instance, the concept of a forest of quadtrees is proposed by Jones and Iyengar[26]. A forest of quadtree specifies a quadtree as a collection of subtrees. The main idea of a forest of quadtree is to utilize a Grey node to indicate the contents of its subtrees. i.e, an internal node is said to be of type GB if at least two of its sons are BLACK or of type GB. Otherwise the node is said to be GW. It turns out, the number of maximal GB blocks for each object is considerablly smaller than the number of maximal BLACK blocks for the same object. By using a directory, the location of GB blocks can be determined.

Another related structure called a field tree is attributed to Frank[18]. A field tree is a variation of a quadtree. The whole image in a field tree corresponds to the root of the tree. The root node is subdivided into four quadrants as is in a quadtree. The main difference is that the subquadrants in a field tree are then translated in both x and y directions by 1/4 the size of its parent node (thus nine subblocks are required to cover the parent node). Because of this orderly translation of the levels of a field tree, a small polygon split by a cell boundary on level i is enclosed by a cell at level i+1 or i+2, a small polygon centered in the study area is never split by cell boundary, etc.[18].

However, both structures as originally proposed do not take paging and I/O buffering into account. Moreover, a forest of quadtrees only partially alleviates the need for the reconstruction process and a field tree is not very useful for handling images which containing large objects.

The second difficulty of being sensitive to shift stems primarily from the fact that in order to obtain a quadtree a grid of specific resolution is placed on the image and the region is encoded as a set of maximal blocks. As the result of this technique, blocks must have standard sizes (powers of 2) which are position dependent. A prime example can be found in [15] where a spatial complexity analysis is provided for a region of size $2^k \times 2^k$ appearing in a $2^n \times 2^n$ array.

Efforts to overcome this drawback have been reported in the literature since 1982. A data structure, invented by Samet [51], called Quadtrees and Medial Axis Transform encodes the skeleton of the quadtrees and the resulting data is less sensitive to shifting. A method for normalizing quadtrees with respect to translations was reported in [32]. This method offers an optimal solution to the problem in a sense that it requires the least amount of space and is shift invariant.

2.3. R-trees: a Dynamic Index Structure for Spatial Searching

2.3.1. Preliminary

Regions are areas in a two-dimensional space and are not well defined by points. For example, a map contains many regions of non-zero size in two-dimensions. A common operation on an image is a search for all connected regions in an area, for example to find all objects within 10 miles of a particular point. This kind of spatial search occurs frequently in computer aided design, geographic-data applications, computer graphics and robotics. It is, therefore, important to be able to perform this kind of spatial searching efficiently.

The above spatial searching is often called an object oriented search. In general, the type of pictorial object may be a "point", as cities in a map, or a "line segment", as for a highway, or "region", as in an island.

To simplify the discussion, a pictorial object is deemed to be equivalent to a "connected region". This limitation, however, does not lose generality since all the algorithms can be extended to allow richer criteria in segmenting objects.

The primary concern in organizing spatial data in a file is the efficiency of access. Many studies have been performed on access mechanisms for multi-dimensional data, but most focus on point data. Spatial access to point data in a k-dimensional coordinate space is analogous to multi-key access to records with composite keys in the sense that coordinates can be used together as a key. Access to vector data or region data, however, is more problematic because exact-match queries are almost impossible. In other words, no predetermined keys corresponding to access are known *a priori* to a query.

A number of structures had been proposed for handling multi-dimensional spatial data, among which there are cell methods [7,8,13], quadtrees, k-d trees [6,8], k-d-B trees [41], index intervals [61], and grid files [35]. These techniques are shown to be inadequate in retrieving objects of nonzero size in an multi-dimensional environment[24]. For example, many approaches are based on the principle of subdivision of the image, i.e. cell methods, quadtrees, grid files, k-d trees, k-d-B trees etc. As a result, objects in the study area are partitioned into small pieces. These techniques are very useful for a number of low level spatial data processing algorithms, such as: set operations, point-in-polygon, and low level range search. However, the absence of an ability to index spatial objects of nonzero size directly based on the analog form of the spatial objects is a significant disadvantage. As the result, a high level query usually requires the whole file to be retrieved. For instance, the problem of finding all objects which are contained in a given rectangular area requires an elaborate reconstruction process of the spatial objects from the low level primitives. In addition, a data structure such as a k-d tree and a k-d-B tree are useful only for point data; index intervals cannot be used in multiple dimensional cases.

Recently, however, a new data structure called an R-tree[24] has received increasing attention. An R-tree is essentially an index based on spatial location. Its capabilities in dealing with advanced queries, including dynamic computation of the spatial relationship between objects, paging and I/O buffering, labels it as an excellent index scheme for high level spatial data retrieval.

2.3.2. R-tree Index Structure

The underlying assumption of an R-tree is that there is a spatial database which consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier which can be used to retrieve it.

An R-tree is composed of leaf nodes and nonleaf nodes with a distinct node called the root. As defined by Guttman [24], leaf nodes of the R-tree contain entries of the form:

$$(I, tuple-identifier),$$

where tuple-identifier refers to a tuple in the database and I is an n-dimensional rectangle which is the bounding box of the spatial object index:

$$I=(I_0, I_1, \ldots, I_{n-1})$$

Here n is the dimension and I_i is the closed bounded interval [a,b] describing the extent of the object along dimension i. Alternatively I_i may have one or both endpoints equal to infinity, indicating that the object extends outward indefinitely.

On the other hand, non-leaf nodes contain entries of the form:

$$(I, child-pointer)$$

Where *child-pointer* is the address of the successor node in the next level of the R-tree and I is the minimal rectangle which bounds all rectangles in the descendent node's entries.

Let M be the maximum number of entries that fit in one node and let $m \leq M/2$ be a parameter specifying the minimum number of entries in the node. Nodes in an R-tree corresponding to disk pages of reasonable size have values of M that produce good performance so that a spatial search needs to visit only a small number of nodes.

To make an R-tree into a dynamic structure that needs no periodic reorganization and to utilize the spatial relationship efficiently, the following requirements are imposed:

(1) Every leaf node contains between m and M index records unless it is the root.

(2) For each index record $(I, tuple\text{-}identifier)$ in a leaf node, I is the smallest rectangle that contains the n-dimensional data object represented by the indicated tuple.

(3) Every non-leaf node has between m and M children unless it is the root.

(4) For each entry $(I, child\text{-}pointer)$ in a non-leaf node, I is the smallest rectangle that contains the rectangles in the child node.

(5) The root node has at least two children unless it is a leaf.

(6) All leaves appear on the same level.

Figures 2.7 and 2.8[2] show the structure of an R-tree and illustrate the containment and overlapping relationship that can exist between its rectangles.

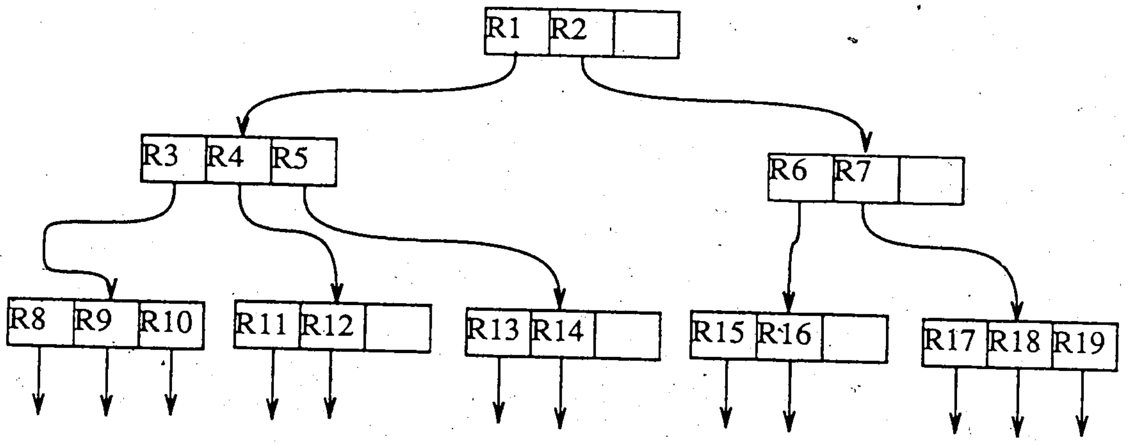


Figure 2.7 An R-tree

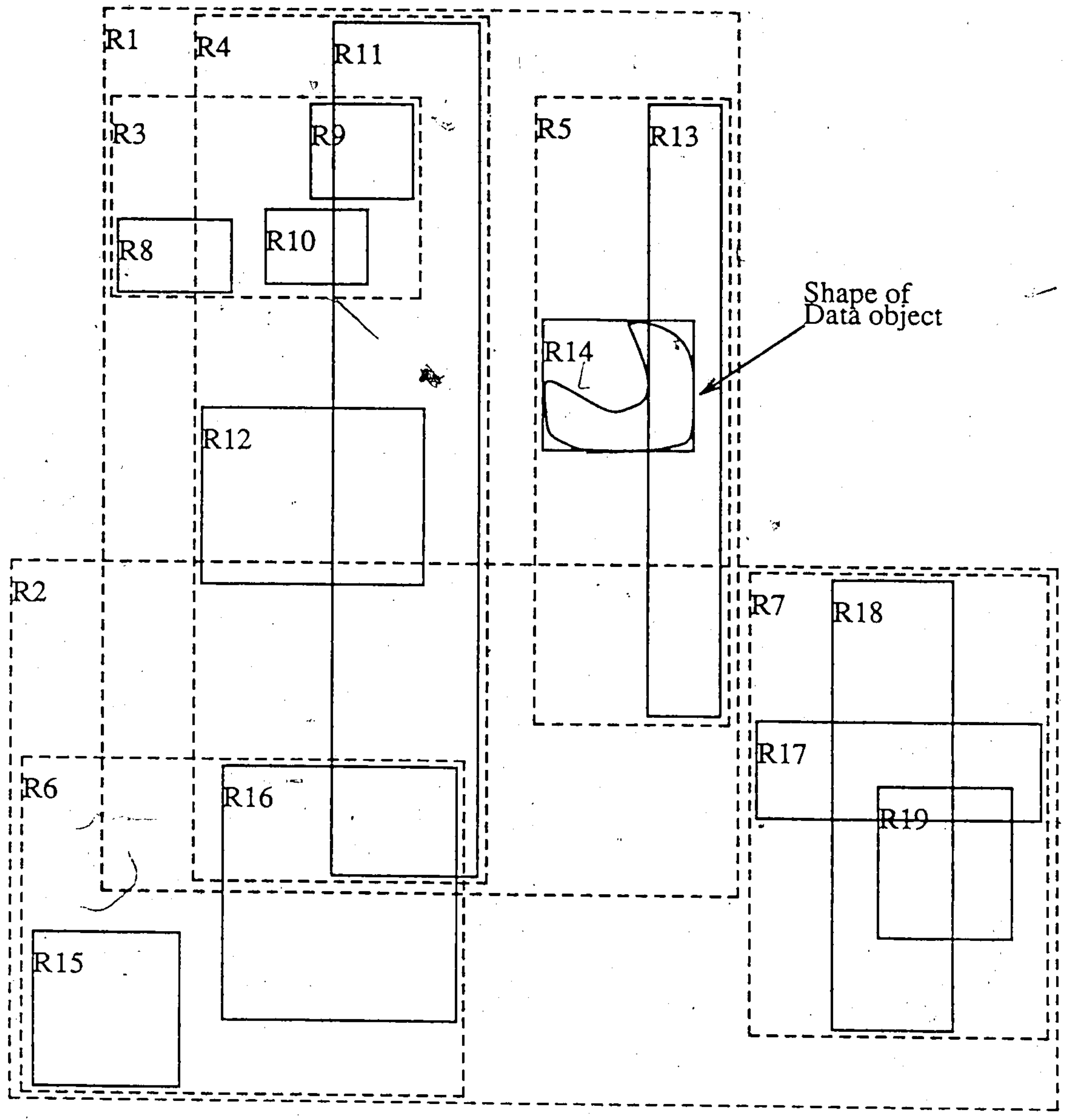


Figure 2.8 A Possible Spatial Relationship Among R-tree Nodes in Figure 2.7

---

[2] According to requirement (4), any non-leaf block is the smallest rectangle that contains the rectangles of its children. For instance, block R2, R6 and R15 share a same left bottom corner.

When a spatial organization on a relation is defined, an R-tree is a height balanced tree constructed the same way a B-tree is for indexing. The only difference between the two is that the spatial relationships among the objects represented by the relation tuples may not necessarily be part of the values stored in the tuple itself and thus they must be provided externally. For this reason R-trees are described as a higher-dimensional generalization of B-trees [24,47].

On the other hand, at first glance, an R-tree may be considered to be very similar to a traditional pointer-based quadtree. In fact, they are both trees composed of nodes representing two-dimensional rectangles (or squares) with each internal node bounding all the space represented by all descendent nodes; both are hierarchical in nature and pointer-based in structure.

There are, however, many significant features which distinguish an R-tree from a traditional pointer-based quadtree. Firstly, an R-tree is more flexible. For example, as previously mentioned, blocks in quadtrees are disjoint and have standard sizes (powers of 2) and positions due to a grid that is imposed on the image during the hierarchical decomposition process. These requirements are no longer needed in an R-tree. Secondly, an R-tree is dynamic and height balanced and the storage organization of an R-tree is based on a B-tree. On the other hand, traditional pointer-based quadtrees are unbalanced and they do not take paging of secondary memory into account. Thus, an R-tree is better in dealing with paging and I/O buffering [5,11,24,47]. The most important feature that distinguishes R-trees from quadtrees is the fact that, at the leaf level, the former stores full and non-atomic spatial objects whereas the latter may indiscriminately decompose the objects into lower level pictorial primitives. In addition, the corresponding rectangle of each node in an R-tree is the minimal rectangle which bounds all the data objects in all descendent nodes. Such features provide R-trees with the ability of higher level object oriented search. A similar search with a quadtree requires an elaborate reconstruction process of spatial objects from the lower level primitives.

An R-tree have many advantages over traditional pointer-based quadtree that are desirable both in theory and in practice(e.g., high level object oriented search). They cannot, however, be applied to image representation in current digital computer systems since at the terminal level an R-tree stores full non-zero size objects which may possess different shapes. In other words, an R-tree is a data structure for index purposes only.

Unfortunately, the traditional quadtree cannot be easily extended to include an R-tree like index mechanism. Since the objects are decomposed into primitives in quadtree representation and primitives are grouped together to form a upperlevel block by quadrants, a block in a quadtree cannot bound all the objects which intersect it. This implies that many nodes in a quadtree are not useful for object indexing of objects. Thus an extra index file needs to be built that is separate and distinct from the quadtree, and this index requires considerable space.

Finally, the height of an R-tree containing N index records is bounded by $\lceil \log_m N \rceil - 1$, since the branching factor of each node except the root is at least m. Empirical results in [24,47] show that R-trees are excellent speed-up devices for spatial searching. The main reasons for this are as follows:

(1) The storage organization of R-trees is based on B-trees which lend themselves conveniently to deal with paging and disk I/O buffering [5,11,24].

(2) The data objects of interest are accurately represented in a form analogous to their spatial nature which is very useful for efficient and direct spatial search based on the

analog form of spatial objects.

(3) The dynamic nature of R-trees makes it possible to adjust the overall organization of the tree so that the resulting R-tree is denser with less "coverage" and "overlap", achieves significant efficiency in terms of space and time complexity [24,47].

2.4. Conclusions

The problem of region representation and the problem of spatial data searching have been presented. Quadtrees and R-trees have been shown to be useful for region representation and for indexing spatial data objects that have nonzero size respectively.

In the following chapter a hybrid structure called R/Q tree is proposed which combines the features of R-trees and quadtrees for region representation and spatial searching purposes.

Chapter 3

A HYBRID DATA STRUCTURE: AN R/Q TREE

The purpose of this chapter is two-fold. The first is to introduce the hybrid R/Q tree, and the second is to establish a foundation upon which algorithms will be developed in subsequent chapters. A computation model will be used in the analysis of the asymptotic time bound of the algorithms. This model requires disk-like secondary storage and images stored on it with portions of the data, as needed, processed in main memory. Since the I/O operations to be performed are a dominant factor of disk-based algorithms, the number of pages accessed during an algorithm provides a measure of its cost.

3.1. Definitions and Conventions

Definition 3.1 An R/Q tree is an R-tree built on top of a sequence of linear quadtrees with each linear quadtree organized by either a B^+-tree or a hashing scheme.

Example 3.1 Figure 3.1 shows the structure of an R/Q tree.

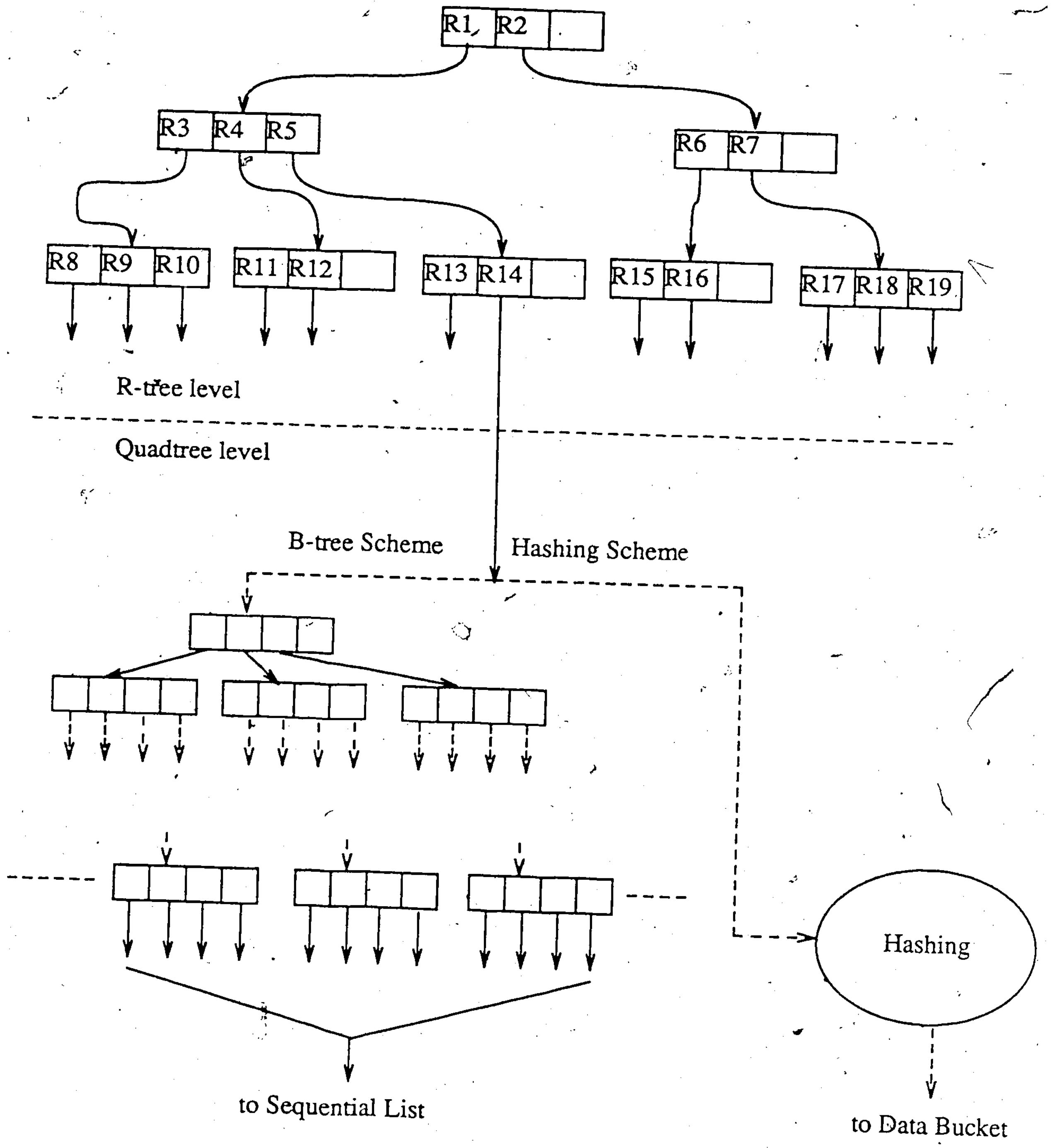


Figure 3.1 An R/Q tree of Figure 2.8.

The following conventions are adopted in this thesis:

(1) Each leaf node except the root in the R-tree contains entries of the form:

$(I, tuple-identifier)$

where *tuple-identifier* is a pointer to the root of the B^+-tree of the object and I is a two-dimensional minimal rectangle which bounds its constituent data objects:

$I=(I_0, I_1)$.

Here I_0 and I_1 are the coordinates of the bottom-left vertex and the top-right vertex of the minimal rectangle relative to the bottom-left vertex of the minimal rectangle of the father node respectively. Let the relative coordinates of I_i be $[\Delta I_{ix}, \Delta I_{iy}]$. Then the absolute coordinates of I_i, $[aI_{ix}, aI_{iy}]$ are defined recursively as follows:

(a) $[aI_{ix}, aI_{iy}]=[\Delta I_{ix}, \Delta I_{iy}]$ if the node is the root.

(b) $[aI_{ix}, aI_{iy}]=[\Delta I_{ix}+afl_x, \Delta I_{iy}+afl_y]$ if the node is not the root.

where $[afl_x, afl_y]$ are the absolute coordinates of the bottom-left vertex of the minimal rectangle of the father node.

The advantage of this small modification in R-trees will be shown in the next chapter.

(2) A non-leaf node except the root in an R-tree contains entries of the form:

$(I, child-pointer)$

where I is defined as specified in (1) and the child-pointer is defined the same as a standard R-tree.

(3) A root node is defined the same as a standard R-tree.

The secondary index file which uses a B^+-tree structure can also be specified similarly except the first parameter of the entry for each node is the key and the second parameter of the leaf nodes is the resolution parameter of the corresponding block as defined in Chapter 2. Furthermore, all leaf nodes form a linked list in which the nodes are sorted in ascending key order. This linked list of leaves is referred to as the sequence set of B^+-trees [11].

Using PASCAL-like syntax, leaf and non-leaf nodes of an R-tree with M entries can be defined as follows:

```
type ENTRY = record
                    X1,Y1,X2,Y2: integer;
                    POINTER: integer;
     end;
     NODE = record
                    CLASS:(leaf,non-leaf);
                    DESC: array [1...M] of ENTRY;
                    VALID: integer;
            end;
```

Similarly, leaf and non-leaf non-leaf nodes of a B^+-tree with K entries can be defined as follows:

```
type ENTRY = record
                    KEY: integer;
                    POINTER: integer;
     end;
     NODE = record
                    CLASS:(leaf,non-leaf);
                    DESC: array [1...K] of ENTRY;
                    VALID: integer;
            end;
```

The POINTER field of a leaf node is used for storing the resolution parameter of the block and the last entry can be used as the pointer of the sequence set.

3.2. Constructing an R/Q tree

Constructing of an R/Q tree from the original array description of the image consists of four phases:

Phase 1. Partition the whole image into separate objects, each identified by a minimum bounding rectangle.

The following algorithm [12] can be used in this phase:

```
Procedure ARRAY-TO-MLQ(E,key,s)
  begin
    if s=0 then
      if E is a BLACK pixel then
        begin
          add pair <key,s> to LIST;
          return(BLACK);
        end
        else return(NONBLACK)
      else
        begin
          for i:=0 to 3 do
            color[i]:=ARRAY-TO-MLQ($E_i$,key+i*4**(s-1),s-1);
          if color[i] is BLACK, $0 \le i \le 3$, then
            begin
              replace the last 4 pairs in LIST by <key,s>;
              return(BLACK);
            end
            else return(NONBLACK);
        end;
    end.
```

This procedure takes as input three parameters E, key and s, where E is a 2^s by 2^s binary array, key and s initially correspond to zero and n, respectively. The algorithm examines each pixel value of the binary image in Morton sequence order. If a pixel is BLACK, then its two-tuple representation is formed and added to LIST which is initially empty. It then recursively merges the four small BLACK nodes corresponding to the last four two-tuples in LIS[illegible] a bigger BLACK node. Upon termination of the algorithm, LIST contains all maximal BLACK nodes.

Phase 2. Generate, for each separate object a linear quadtree description that is centered on the object.

Many criteria, which depend on domain knowledge, can be applied in this step; however, to simplify the discussion, objects are identical to connected components of the image. Thus, there are two things that need to be done in this step. First, label the connected components of the image, using the algorithm given in [12]. Second, for each connected component, calculate the normalized linear quadtree description with respect to translation.

The advantage of the proposed approach is that it is simple and can generate the desired description in reasonable time with a tolerable amount of space. In addition, it is also less sensitive to data error.

An alternative approach could place the object to the left-bottom most corner, i.e. the minimal value of the X-coordinate and the Y-coordinate of an connected component are $X_{\min}$ and $Y_{\min}$ respectively. The desired linear quadtree description is obtainable by translating the $X_{\min}$ and $Y_{\min}$ positively. A procedure named **TRANSLATE** can be used to achieve this purpose [12].

There are, of course, some other heuristic rules similar to the one suggested herein that can be chosen for normalizing purpose.

Phase 3. Organize the data for each linear quadtree by a hashing scheme [13] or by a B^+-tree.

Case 1: B^+-tree scheme:

lgorithms for manipulating B-trees can be found in [5,11]. Since B^+-trees are variants of B-trees, the corresponding B^+-tree algorithms can be easily constructed. As a matter of fact, B^+-trees can be constructed by means of insertion similar to inserting an index record to an R-tree. The later algorithm will be presented later in this chapter.

Case 2: hashing scheme:

The construction of a hashing file can be done by insertion. For example, with an extendible hashing scheme, starting with a single bucket. If a data bucket overflows, the corresponding block in the address space is halved, a new data bucket is added, and the directory is updated to reflect the changes being made [13].

Phase 4. Organize the separate objects into an R-Tree.

Although R-trees are natural extension of B-trees, they are more difficult to cope with. The difficulties are caused by the two main phenomena of R-trees: coverage and overlap, as illustrated in Figure 2.8. Coverage is defined as the total area of all the minimal bounding rectangles (MBRs) of all R-tree nodes at the same level and overlap is defined as the total area contained within two or more MBRs of the nodes in R-tree. Efficient R-tree searching demands that both overlap and coverage be minimized, although overlap seems to be the more critical of the two.

An interesting question is: Given an arbitrary set of data objects, can a zero overlap R-tree always be obtained? It was proved that for pointer objects at the leaf level the answer is yes [53]. Unfortunately, it was also shown that a zero overlap R-tree does not always exist when the data objects have nonzero sizes. Thus, at best, it is possible to minimize the overlap, but the issue of coverage remains. The simultaneous minimization of both coverage and overlap is a complex task.

An algorithm, call PACK, proposed in [53] addresses both of these problems by grouping nearest neighbors of a specified node at each level in forming a higher level node. The result of this technique is that all descendent nodes of any node in the R-tree are relatively close to one another in spatial position, hence both coverage and overlap can be minimized. Since optimal grouping requires considering M items simultaneously, which could be combinatorially explosive, PACK produces as output a near-optimal packed R-tree.

To present the algorithm, assume all nodes are packed as full as possible and each node contains M entries. Also assume that the total number of data objects is a power of M. This would be highly unlikely in any real application, but this assumption makes it possible to dispense with the trivial special cases of one partially-filled node for leftover entries per level.

Algorithm PACK is a recursive procedure. Its sole argument is DLIST, a list of data objects to be packed. NN is a nearest neighbor function which takes two arguments. NN(DLIST,I) returns the item in the list DLIST which is spatially closest to item I and has the additional effect of deleting that item from DLIST. The algorithm is:

```
Procedure PACK(DLIST)
  begin
    if DLIST contains M objects then
      begin
        Allocate a pointer to a new R-tree node, No;
        Cause pointer of No to point to items of DLIST;
        return(No);
      end
    else
      begin
        Order objects of DLIST by some spatial criterion;
        NLIST:={};
        While DLIST is not empty do
          begin
            I[1]:=first object from DLIST;
            DLIST:=tail(DLIST);
              for i:=2 to M do
                I[i]:=NN(DLIST,I[1]);
              Allocate a new R-tree node, N1;
              Cause pointers of N1 to point to I[1],...,I[M];
              APPEND(NLIST,N1);
          end;
        return(PACK(NLIST));
      end;
    end.
```

It should be mentioned that at the leaf level the pointers point to the root of the B^+-tree obtained from Phase 2, and the minimal bounding rectangle of the corresponding object obtained in Phase 1 of the algorithm.

3.3. Algorithms for Maintainance an R/Q tree

In this section the problem of maintaining an R/Q tree index structure for a dynamically changing random access file is considered. At the quadtree level, changing is equivalent to node insertion and deletion. At the R-tree level, however, changing low level primitives may not necessarily result in deletions or insertions of objects, although the spatial properties of the objects and the spatial relationship among objects are subject to change. To prevent gradual deterioration in an R-tree, however, the deletion and insertion algorithms to be explained are applied to adjust the R-tree whenever the low level primitive changes.

Of the two different approaches for organizing low level primitives, the hashing scheme is simpler than the B^+-tree scheme. The procedures for maintaining a dynamic hashing file can be found in [13,16]. In the following it is assumed that the index file for the low level primitive is a B^+-tree.

3.3.1. Algorithms for Maintaining of an R-tree

This section follows a set of algorithms for maintaining an R-tree during and after update of the tree.

3.3.1.1. Insertion

Inserting index records for new data values is similar to insertion in a B-tree in that new index records are added to the leaf nodes. Nodes that overflow are split, which then propagates up the tree.

The main procedure is called **INSERTION** and is invoked with a pointer R to the root of the R-tree and a new index entry E. Step 1. calls a procedure **CHOOSELEAF** to select a leaf node in which to place E. **CHOOSELEAF** finds the path from the root to the leaf node that needs the least enlargement to include E and then properly modifies the minimal bounding rectangle of the nodes along the path. Procedure **SPLITNODE** is invoked whenever an attempt is made to add a new entry to a full node containing M entries. Similar to a B-tree, it is necessary to divide the collection of M+1 entries between two nodes in such a case. Procedure **ADJUSTTREE** propagates node splitting in the R-tree until either the root is reached or there is space where the split node can be installed.

```
Procedure INSERTION(R,E)
  begin
    L:=CHOOSELEAF(R,E);
    if L.VALID<M then
      begin
        install E into L;
        L.VALID:=L.VALID+1;
      end
    else
      begin
        (L,LL):=SPLITNODE(L,E);
        (D,DD):=ADJUSTTREE(R,L,LL);
        if R.^=D then
          begin
            Create a new node N;
            Install D and DD into N;
            Set pointer R point to N;
          end;
      end;
  end.

Procedure ADJUSTTREE(R,N,NN)
  begin
    if N<>R.^ then
      if FATHER(N).VALID<M then
        begin
          Install NN into FATHER(N);
          FATHER(N).VALID:=FATHER(N).VALID+1;
          return(N,NN);
        end
      else
        begin
          (L,LL):=SPLITNODE(FATHER(N),NN);
          return(ADJUSTTREE(R,L,LL));
        end;
    else return(N,NN);
  end.
```

3.3.1.2. Deletion

Deletion removes an index record E from an R-tree. It is an inverse operation of insertion and is also similar to deletion in a B-tree.

The main procedure, **DELETION**, takes as input two parameters R and E, where R is a pointer which points to the root of the R-tree and E is the index record to be deleted. Step 1 calls **FINDLEAF** which returns the leaf node containing the index entry E if there is one. In the following steps entry E is deleted from the leaf L and Procedure **CONDENSETREE** is invoked. Procedure **CONDENSETREE** eliminates L from the R-tree and relocates its entries if it has too few entries. Then the node elimination is propagated upwards as necessary and all covering rectangles on the path to the root are adjusted accordingly.

```
Procedure DELETION(R;E)
  begin
    L:=FINDLEAF(R,E);
    if L=NIL then return;
    Remove E from L;
    L.VALID:=L.VALID-1;
    CONDENSETREE(R,L);
    if R.^.VALID=1 then Set R point to the child of R.^;
  end.
```

```
Procedure CONDENSETREE(R,N)
  begin
    Q:={};
    while R.^<>N do
      begin
        F:=FATHER(N);
        if N.VALID < m then
          begin
            Remove entry which pointer to N from F;
            F.VALID:=F.VALID-1;
            Add N to set Q;
          end
        else adjust entry which pointer to N in F tightly;
        N:=F;
      end;
    while Q is nonempty do
      begin
        N:=first node from Q;
        Q:=tail(Q);
        Reinsert all entries of N into R-tree[3];
      end;
  end.
```

3.3.2. Algorithms for Maintaining a B tree

At a very high level description of the algorithms, the algorithms for maintaining an R-tree are the same as for maintaining a B tree. In fact, the index of an R-tree is constructed the same way as for a *B*-tree except in the case of an R-tree the spatial relationships among the objects are explicitly stored, see Chapter 2. Again, the only difference between a B^+-tree and a *B*-tree is at the leaf level where a sequential linked list is constructed for a B^+-tree.

It is much more difficult to maintain an R-tree than to maintain a B^+-tree of the same size. To see the differences compare the procedure **INSERTION** and **DELETION** as

[3] Reinsertion is similar to procedure **INSERTION** except entries from higher-level nodes must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

follows:

3.3.2.1. Insertion

The main procedure **INSERTION** for a B^+-tree is exactly the same as that described in Section 3.4.1. The subroutine **CHOOSELEAF** in a B^+-tree is simpler in that no node modification is needed during tree traversal. Furthermore, the path from the root to a desired leaf node is unique in a B^+-tree. **CHOOSELEAF** in a B^+-tree is simply a key search. The same operation on an R-tree, as in Section 3.4.1, requires adjusting the covering rectangles along the path from the root to the leaf node. Since the path is not unique, extra effort must be made in selecting the best path for an R-tree. Again, **NODESPLITING** in a B-tree is a trivial operation, but the same operation in an R-tree is rather difficult since the division should be done in a way that the resulting two new nodes have least coverage. No efficient optimal splitting algorithm has been reported yet. A linear-cost algorithm proposed by Guttman [23], however, was shown to be fast, and the slightly worse quality of the splits did not noticeably affect the search performance.

The subroutine **ADJUSTTREE** is the same with respect to operations related to entries.

3.3.2.2. Deletion

The main procedure **DELETION** for a B^+-tree is also the same as that described in Section 3.4.1:

The function of the subroutine **FINDLEAF** is essentially the same as its counterpart **FINDLEAF** described in Section 3.4.1. except the finding of the desired node in a B^+-tree is simply path finding in which backtracking is not required. Whereas the same task for an R-tree requires an exhaustive depth-first search due to the overlapping problem existing in R-trees.

The procedure **CONDENSETREE** in a B^+-tree is, however, quite different from the one described in Section 3.4.1. Firstly, there is no such operation like adjusting the covering rectangle in a B^+-tree. Secondly, the disposing of under-full nodes in a B^+-tree is done by merging two or more adjacent nodes, therefore reinsertion is not needed. A B-tree like approach is possible for R-trees, although there is no adjacency in the B-tree sense: an underfull node can be merged with whichever sibling will have its area increased least, or the orphaned entries can be distributed among sibling nodes. Two problems are associated with this approach: it causes nodes to split and makes an R-tree deteriorate [23].

On the other hand, if the key to be deleted also appeared in an internal node of the B^+-tree, then the index needs to be adjusted properly. Furthermore, extra attention mus be paid to take care of the linked list at the leaf level of the B^+-tree. Each adjustm operation, however, requires at most one additional page in and one additional page ut.

3.4. The Cost of Constructing and Maintaining an R/Q tree

To analyze the cost of constructing and maintaining an R/Q tree and the algorithms for manipulating an R/Q tree, it is necessary to know how many pages must be transformed from bulk storage to main memory and how many pages must be written onto bulk storage. For this analysis the following assumption is made: Any page, whose content is examined or modified during a single retrieval, insertion, or deletion of an entry, is respectively fetched or paged out exactly once. It will become clear during the course of this section that a page area to hold $\max\{h_R, h_B\}+1$ pages in main store is sufficient to accomplish this. A more powerful paging scheme, i.e. virtual memory using the LRU paging algorithm [11], will, of course, decrease the number of pages which must be fetched or paged out. For the sake of simplicity such schemes will not be analyzed.

Suppose the number of objects in the image is N, the number of pixels in the image is N_p, the number of blocks in the description of the ith object is $B[i]$, and the number of blocks in the image is N_b, i.e. $N_b=\sum_{i=1}^{N} B(i)$. Let M and K be the maximum number of entries that will fit in one node in the R-tree and a B^+-tree, and let $m \le M/2$ and $k \le K/2$ be the parameters specifying the minimum number of entries in a node of the R-tree and a B^+-tree respectively.

Clearly, the height of the R-tree h_R is bounded by $\lceil \log_m N \rceil - 1$ and the height of the ith B^+-tree $h_{B[i]}$ is bounded by $\lceil \log_k B[i] \rceil - 1$. The height of R/Q tree, therefore, is at most $h_R + h_B$, where $h_B = \max\{h_{Bi}\}$, i=1,...,N.

Denote by $f_{\min}(f_{\max})$ the minimal (maximal) number of pages fetched, and by $w_{\min}(w_{\max})$ the minimal (maximal) number of pages written.

3.4.1. The Cost of Constructing an R/Q tree

The cost of constructing an R/Q tree is the total cost of the four phases:

The cost of phase 1:

Suppose the input array of pixels is a sequential file in Morton order, and LIST is actually a sequential file in reversed Morton order so that the operation merge of the last four nodes in algorithm **ARRAY-TO-MLQ** can be done efficiently.

Theorem 3.1: Procedure **ARRAY-TO-MLQ** constructs an MLQ, at a cost bounded by $O(N_p)$, where N_p is the number of pixels in the image.

Proof: Let $F(4^n)$ and $W(4^n)$ be the number of pages fetched and the number of pages written as required by procedure **ARRAY-TO-MLQ** to generate an MLQ. Clearly, when n=0, F(1)=W(1)=1. If n>0, $F(4^n)$ $(W(4^n))$ is the total number of pages fetched (written) in the four calls of **ARRAY-TO-MLQ** on an array of size 4^{n-1}, plus at most 2 pages in and 2 pages out for the operation: replace the last 4 pairs in the LIST by <key,s> in the algorithm, under the assumption of our paging scheme. That is,

$$F(4^n) = 4F(4^{n-1})+2, \quad n>0$$

$$W(4^n) = 4W(4^{n-1})+2, \quad n>0$$

The theorem follows by solving the recurrence.

Q.E.D.

The cost of phase 2:

Case 1: B^+-tree scheme

The cost of building a B-tree from a list of pre-sorted N_b data is proportional to the maximum number of nodes in the B-tree which is bounded by $O(N_b)$ since a recursive procedure similar to **ARRAY-TO-MLQ** can be used for the initial construction purpose and the cost of the algorithm can be estimated similar to Theorem 3.1 for procedure **ARRAY-TO-MLQ**.

Case 2: hashing scheme

The worst case analysis of any hashing scheme is discouraging. However, since the worst case is rare and the construction is a global process, the cost of this phase is equal to $A \times N_b$ rather than $W \times N_b$, where $A(W)$ is the average (worst case) cost for inserting a record into a hashing file. Again, the average cost for inserting a record into a hashing file tends to be bounded by a constant [58]. Thus, the construction of the hashing file is bounded by $O(N_b)$.

The algorithm for labeling connected components given an linear quadtree description of the region can be found in [12]. It is divided into three steps. The first step is to discover all possible adjacencies between any pair of nodes. To do so, at most the finding of four neighbors is required for each node. Hence if a B^+-tree scheme is adopted, the maximum number of pages fetched in Step 1 is 4h, where h is the height of the B-tree and is bounded by $O(log_k N_b)$. If a hashing scheme is adopted, then the number of pages fetched in Step 1 can be bounded by $O(1)$. The second step, however, establishes a set of bottom up trees for each set of connected components. This is achieved by using the well known **UNION-FIND** algorithm [3]. That is, whenever a adjacency is discovered between two nodes, their corresponding equivalence classes will be merged to form a larger equivalence class, where nodes in the same equivalence class are connected. Similar to the analysis in [1], the number of pages in and the number of pages out are all bounded by $O(N_b \log N_b)$. Step 3 finds the normalized linear quadtree description for each connected component. The normalized linear quadtree description, as discussed in Section 3.3, can be obtained by a single translation in blocks, if a heuristic normalizing method is adopted. Translation may cause splitting and merging in blocks, but in the worst case, simply assume blocks are split into pixels and the normalized linear quadtree is constructed from a set of pixels in Morton sequence. By **Theorem 3.1**, it is not difficult to see, that the number of pages in and the number of pages out during the course of normalizing can never exceed $O(N_p)$.

Theorem 3.2: The cost of phase 2 is bounded by $O(N_p)$

Proof: The last step is the dominant factor contributing to the entire phase and is bounded by $O(N_p)$.

Q.E.D.

The cost of the phase 3:

Theorem 3.3: The cost of phase 3 is bounded by $O(N+N_b)$.

Proof: Constructing of the B^+-tree for object i requires at most $O(B[i])$, by the analysis of Phase 2. Thus, the total cost is bounded by $O(N+N_b)$, where N is the number of objects in the image and $N_b = \sum_{i=1}^{N} B[i]$ is the total number of primitive blocks.

Q.E.D.

The cost of phase 4:

Lemma 3.1: The number of pages at level $h_R - i$, where $i=0,\ldots,h_R-1$, is N/M^{i+1}, where N is a power of M.

Proof: (By induction)

If i=0, then since each page can store M records, N records require N/M pages.

Thus, the conclusion is true.

Suppose, the conclusion is true for $0 \le i < k$, where $k < h_R - 1$. The following shows that it is also true for i=k.

Since each entry in a node at the level $h_R - k$ points to a node at the level $h_R - k + 1$ and there are exactly M entries per node, the number of pages required at the level $h_R - k$ is equal to the number of pages at level $h_R - k + 1$ divided by M. That is $\frac{N/M^k}{M} = N/M^{k+1}$.

Therefore, the lemma is proved.

Q.E.D.

Theorem 3.4: The cost of phase 4 is bounded by $O(N^2/M)$.

Proof: The algorithm **PACK** constructs an R-tree in a bottom up fashion. At the level $h_R - i$, it packs a group of M nodes which are spatially close to each other to form a node at level $h_R - i - 1$. Each such operation requires a scan of the entire DLIST which requires $O(N/M^{i+1})$ pages in, by **Lemma 3.1**. Again, by **Lemma 3.1** the number of pages in level $h_R - i - 1$ is N/M^{i+2}. Therefore, to form the DLIST at level $h_R - i - 1$, $O(N^2/M^{i+1})$ nodes are paged in and $O(N/M^{i+1})$ nodes are paged out. In other words, the number of pages in and the number of pages out are bounded by $O(N^2/M^{i+1})$. The cost of constructing an R-tree is, therefore, bounded by:

$$O\left[\sum_{i=0}^{h_R-1} (N^2/M^{i+1})\right] = O(N^2/M).$$

Q.E.D.

Theorem 3.5 The cost of constructing an R/Q tree is bounded by $O(N_p+N_b^2/M)$.

Proof: Directly derivable from **Theorems 3.1-3.4**.

Q.E.D.

Remark: the first term is an upperbound of the cost for constructing a linear quadtree and the second term is the cost for constructing a packed R-tree. Using a technique proposed recently by Samet[53], the cost of the first term can further be reduced to $O(N_b)$. This is achieved by processing the image in raster-scan (top to bottom, left to right) order and always forming the largest node for which the current pixel is the upper leftmost pixel. Thus the necessity of merging smaller blocks into a larger block is avoided. Extra memory space (approximately 2^{n+1} for a $2^n \times 2^n$ image) is needed to achieve this goal. Interested readers are referred to [53] for a detailed algorithm. Therefore, in general, it is not difficult to see that the cost of constructing an R/Q tree can be bounded by $O(N_b+N^2/M)$, not including the cost of segmenting of the objects in the region.

3.4.2. The Cost of Maintaining an R/Q tree

Since it is hard to theoretically establish an expected case analysis for a hashing scheme due to the difficulty in constructing a representative data model, the cost of maintaining an R/Q tree will focus on the B^+-tree scheme for organizing low level primitives.

Cost of Insertion in an R-tree. From the procedure **INSERTION** it is clear that the least work is required if no page splitting occurs, and:

$$f_{\min}=h_R; \quad w_{\min}=h_R+1.$$

Most work is required if all pages in the retrieval path including the root page are split in two. Since the retrieval path contains h_R pages and a new root page must be written, resulting in:

$$f_{\max}=h_R; \quad w_{\max}=2h_R+1.$$

Cost of Deletion in an R-tree. First, searching the desired node in which the index record to be deleted is contained requires an exhaustive depth first tree search for all subtrees overlapping the record. Second, all under-full nodes caused by entry deletion need to be reinserted as described in subroutine **CONDENSETREE.**

The least work is needed if there is at most one node that overlaps the index record at each level of the R-tree and there is no under-full node[4] after the deletion, then

$$f_{\min}=h_R; \quad w_{\min}=h_R+1.$$

The most work is required if there are overlaps and under-full nodes. The following bound results:

$$f_{\max}=O(N), \text{ where N is the number of objects.}$$

On the other hand, deletion can cause under-full nodes. The deposing of under-full nodes as described in Section 3.4.2 necessitates reinsertions, since each under-full node has exactly $m-1$ children which must be reinserted, the deposing of an under-full node at level i of the R-tree needs at most $(2i+1)\times(m-1)$ pages out, by the worst case analysis of insertion. Assuming, in the worst case, that one under-full node is encountered at each level of an R-tree during the course of condensing the tree. Then

$$w_{\max}=\sum_{i=0}^{h_R-1}(2i-1)\times(m-1).$$

Thus, $w_{\max}=h_R(h_R-1)(m-1)$.

4 A node is said to be under-full if the fanout is less than m.

Cost of Insertion in a B^+-tree. Suppose the height of the B^+-tree is h, then

$$f_{min}=h; \quad w_{min}=1, \text{ and}$$

$$f_{max}=h+1; \quad w_{max}=2h+2.$$

This is because in the case of a B^+-tree no adjusting of the coverage of the minimal bounding rectangles is needed except at most one more page to be fetched and one more page written for the sequential linked list at the leaf level of the B^+-tree during the course of insertion.

Cost of Deletion in a B^+-tree. For a successful deletion the least amount of work is required if no node merging is required, then

$$f_{min}=h; \quad w_{min}=1.$$

A maximal amount of work must be done if all nodes in the retrieval path will become under-full. This requires:

$$f_{max}=2h; \quad w_{max}=h+2.$$

Including one extra page to be fetched and written to update the sequential linked list.

The analysis of the cost of maintaining an R/Q tree is based on how a primitive unit operation impacts the R/Q tree due to the hybrid structure of an R/Q tree. For example, consider the problem of inserting a block into a quadtree description of an object. Since, in general, the minimal bounding rectangle of the object will be enlarged as a result of the operation, the current placement of the index record may become bad after updating. To prevent deterioration in an R-tree, three steps are required: first delete the object to be changed from the R-tree, then insert the block into the B^+-tree and finally reinsert the object according to the new spatial condition. The cost of the operation is, therefore, the total cost of the three steps.

3.5. Conclusion

In this chapter, a hybrid data structure R/Q tree is proposed for region representation. The proposed R/Q trees are very powerful structures which allow both high level object oriented search and low level primitive manipulations with reasonable space cost. An R/Q tree is especially good for applications in which images are large and high level search is needed. Good examples include: geographic information systems, image processing, pattern recognition, computer graphics, etc. The main justification for such applications are:

(1) A linear quadtree should be used to alleviate problems associated with traditional pointer-based quadtrees.

(2) An index file must be build for the linear quadtree description for the region so that insert, delete and retrieve operations can be done efficiently.

Constructing a set of index files for each object instead of constructing one index file for whole image has the following advantages[5]:

1). The size of each individual index file is much smaller in comparison with the size of the index file for the whole image. An obvious advantage is that the maintenance of the index file becomes easier and more efficient. For example, it is possible that all nodes on a path to be updated from the root to a leaf or even the whole index file can reside in main memory. This will greatly help in improving the performance of the algorithm.

[5] An example is contained in Appendix A2 in which a comparison is made between two different approaches with respect to the selected data.

2). Index files for each object act as a "window" of the whole image. This localization property provides expertise to minimize the number of disk accesses for a given query within the window since only only those parts of the data which are spatially relevant to processing the query are retrieved.

3). The data distribution for each object is expected to be uniform since "dead space" in the image is clipped and the resulting data pattern of each object is somewhat like a local "plateau" with respect to the distribution. Thus, a direct file scheme such as hashing, can be adopted to achieve efficient data representation in most of cases.

The idea of building an R-tree on top of the index files for each object to form a hybrid structure further enhances the power of the representation since the hierarchical nature of an R/Q index structure makes it a more efficient index scheme since the secondary index not only speeds up the retrieval but also provides information for high level operation, as explained in the next chapter. In addition, since at the quadtree level objects are encoded in a standard form, an R/Q tree is very useful for a number of applications such as pattern recognition, image translation etc.

The cost of constructing an R/Q tree is bounded by $O(N_b+N^2/M)$ in the worst case, not including the cost of segmenting the objects. This suggests that an R/Q tree is applicable in most of cases. However, in comparison with the linear quadtree with a B^+-tree organization, an R/Q tree has two drawbacks: it is not height balanced and maintenance, as presented in this chapter, is more difficult. Thus, for some applications an R/Q tree may not be a good choice.

Chapter 4
ALGORITHMS FOR MANIPULATING AN R/Q TREE

This chapter contains a set of algorithms for the manipulation of R/Q trees. In particular, an algorithm for finding all objects which intersect a given rectangular area is presented in Section 4.1. As a direct application of the algorithm, two neighbor finding algorithms are developed: fixed radius near neighbor searching and nearest neighbor finding. In addition, an algorithm for testing whether a point is within a region represented by an R/Q tree is provided in Section 4.3. Efficient algorithms for translation and rotation of a region are given in Section 4.3 and 4.4. Two set-theoretical operations are presented in Sections 4.5 and 4.6, where the importance of the hybrid structure is demonstrated. An analysis of the performance of these algorithms supports the statement that R/Q trees increase the efficiency of region operations, particularly those operations which relate to high level object oriented search.

4.1. Algorithm for Region Search

Finding all objects which intersect a given rectangular area is a fundamental operation in systems using spatial data such as computer aided design (CAD) and geographical information systems [23,46]. It is also a cornerstone of many other operations such as neighbor finding, membership testing, and others, which will be discussed in the following sections. The algorithm for region search by Guttman [23] is as follows.

```
Procedure SEARCH(R,E)
  begin
    LIST:={};
    for i=1 to R.VALID do
      begin
        Case
          :R.DESC[i] overlaps E and R.CLASS='nonleaf':
            add SEARCH(R.DESC[i].POINTER,E) to LIST;
          :R.DESC[i] overlaps[6] E and R.CLASS='leaf':†
            add R.DESC[i] to LIST;
          :otherwise:
            do nothing;
        end of case
        return(LIST);
      end;
  end.
```

The procedure **SEARCH** is invoked with a node, R, and a region, E, which correspond to the root of the R tree and the rectangular area to be searched respectively. The algorithm examines each entry of the node R in sequential order and recursively explores promising subtrees. The objects of interest are obtained at the leaf level.

The performance of **SEARCH** can best be evaluated in terms of the number of pages visited per object retrieved. In the best case, every promising subtree does not lead to a dead end. Since the number of nodes in a balanced tree is bounded by $O(B)$, where B is the number of leaf nodes in the tree. It is, therefore, not difficult to see that the procedure **SEARCH** retrieves Q objects with $O(Q)$ disk access in the best case.

In the worst case, on the other hand, promising subtrees lead to dead ends. Hence, in retrieving a single object, **SEARCH** may require $O(N/M)$ disk accesses, where N is the number of objects in the study area and M is the maximum number of entries per node.

[6] In general, different predicates such as covers (or is covered by), etc. can be used to suit a particular query, see 4.1.2 and appendix A2. For some queries, the low level primitives need to be further searched for.

Therefore, the time complexity of **SEARCH** is a function of E due to the tree structure of an R-tree. For example, the best case happens when E covers an area that is significantly large in comparison with the whole image, and the worst case can only occur when E covers an area that is considerably smaller than the whole image. This is because as the amount of data retrieved in each search increases, the cost of processing nodes higher in the tree becomes less significant.

Although performance in the worst case is rather discouraging, the chance of the worst case happening in practical data is expected to be rare. Empirical results in [24] show that the number of pages accessed to retrieve qualifying records decreases as the amount of data retrieved in each search increases. The low cost per qualifying record when retrieving 3% to 6% of the data shows that an R-tree is quite effective in narrowing the search space.

The procedure **SEARCH** can be conveniently used for many spatial search algorithms such as *Point in Object Query, Fixed Radius Near Neighbor Finding*, and *Nearest Neighbor Finding*.

4.1.1. Point in Object Query

In storing descriptive data associated with a specific geographical point, it often becomes necessary to identify the region that contains the point. This problem is often described as the point in polygon problem [13]; however a similar and more general problem, *Point in Object Query*, is necessary in various computational algorithms for geometric data. Prime examples of the use of the algorithm occur in hidden line and surface removal.

Using the procedure **SEARCH**, the *Point in Object Query* can be answered easily.

```
Procedure POINT-IN-OBJECT(R,POINT,O)
 begin
   LIST:={};
   LIST:=SEARCH(R,POINT);
   If O ∈ LIST then return(True) else return(False)
 end.
```

The procedure **POINT-IN-OBJECT**, takes as input three parameters: R, POINT, and O. Where R is a pointer which points to the root of the R-tree, POINT is the index record which corresponds to the point of interest(since point is a special case of rectangle), and O is the object to be tested.

4.1.2. Fixed Radius Near Neighbor Finding

The fixed-radius near neighbor problem deals with objects within a constant radius of one another in a multi-dimensional space. It is precisely formulated as follows: given a set F of N points in a k dimensional Euclidean space, enumerate all pairs of objects within a specified distance of each other. This problem arises in a number of applications such as molecular graphics, cluster analysis, decoding noisy data, and others [7].

In the following, an algorithm by Bentley, J.L. et. al. [7] for the solution of the problem is presented and it is shown how an R-tree can be used to speed up and extend the algorithm.

A crucial concept of the algorithm is a "cell". Partition the k dimensional space into hypercubes of side d called cells of radius d. In doing so, associate each cell with a k-tuple of integers. That is the point $(X_1,\ldots,X_k)$ is in cell $(\lfloor X_1/d \rfloor,\ldots,\lfloor X_k/d \rfloor)$, where $\lfloor X \rfloor$ is the floor of X.

```
Procedure FR-SEARCH(D,R,S)
  begin
    LIST := {};
    Determine the indexes, (I1,J1),(I2,J2), of cells i  which
    (x1,y1), the left bottom corner of S, and
    (x2,y2), the right upper corner of S, are containe
    for i=I1 to I2 do
      for j=J1 to J2 do
        C[i,j]:=SEARCH(R,E);
        {where E is a search area corresponding to c   (i,j).}
    for i=I1 to I2 do
      for j=J1 to J2 do
        append (CHECKPOINTS(i,j,C,D)) to LIST;
    return(LIST);
  end

Procedure CHECKPOINTS(i,j,C,D)
  begin
    LIST:={};
    for every pair <X,Y> of points in C[i,j] do
      add <X,Y> to LIST as a near neighbor pair;
    for every cell K among 8 neighbors of C[i,j] do
      begin
        for every pair of points X in C[i,j] and Y in K do
          if Distance(X,Y) <= D[7] then
            add <X,Y> to LIST as a near neighbor pair;
      end;
    return(LIST);
  end
```

The algorithm takes, as input, three parameters: the search radius D, the root of the R-tree, and the study area S. First it puts the points of the study area S into cells according to their spatial location. It then checks the closeness of any pair of points between two adjacent cells (since the distance measure adopted guarantees that any pair of points within the same cell is a near neighbor pair).

[7] The distance measure here is that of "maximum coordinate" metric. That is if the i-th dimension of point X is denoted by X_i then

$$\mathrm{Distance}(X,Y)=\max_{1\le i\le k}|X_i-Y_i|$$

The original algorithm [7], however, constructs cells by using either an array, AVL trees, or a hash table. The drawbacks of each approach are: An array structure, although very efficient in terms of time complexity [7], is not useful for large images; an AVL structure does not take paging into account and is not efficient for range search; a hash table may become awkward if the data distribution is uneven.

On the other hand, an R-tree is disk oriented and is a dynamic data structure for indexing spatial objects. For point data, a near optimal configuration of the R-tree provides very efficient performance in terms of both time and space complexity. This statement is supported by two facts. First, a near optimal R-tree with respect to coverage can generally be obtained by applying packing techniques as discussed in Chapter 2. Second, zero overlap at the leaf level of a R-tree can theoretically be obtained by rotating the orientation of the major axes, according to the **Zero Overlap Theorem** in [47].

More importantly, the above algorithm can be extended to finding fixed-radius near neighbors for objects of nonzero size in a number of aspects. For example, the above algorithm can be used to solve problems such as:

(1) Given a set F of N objects of nonzero size in a k dimensional Euclidean space, enumerate all pairs of objects within a specified distance of each other from their borders.

(2) Given a set F of N objects of nonzero size in a k dimensional Euclidean space, enumerate all pairs of objects within a specified distance of each other from their centroids.[8]

[8] A minor modification must be made on the statement marked † in the procedure **SEARCH** so that only those objects whose centroids fall in E are picked up.

The above problems, however, cannot be solved efficiently if the previous approaches are adopted because object oriented search is required for every cell.

4.1.3. Nearest Neighbor Finding

The problem of nearest neighbor finding first originated as follows:

(P1) [Closest pair.] Given a set F of N points in K-dimensional space, find the minimum distance between any two points in F.

To solve this problem it is necessary to solve an intermediate problem. The following concepts are necessary to specify the second problem.

Definition 4.1: A hypercube is nonempty if it contains at least one point in F. The distance between any pair of nonempty hypercubes, X, Y, is defined as follows.

$$D(X,Y) = \min\{\text{Distance}(v,u) \mid v \in X, u \in Y\}$$

where the Distance function is defined as before.

The second problem can now be presented:

(P2). [Closest sub-hypercube pair.] Given a hypercube which is decomposed into a set of sub-hypercubes, find the minimum distance between any two nonempty sub-hypercubes.

The second problem can easily be solved, using the procedure **FR-SEARCH**. However, the subroutine **CHECKPOINTS** must be modified to the following:

```
Procedure CHECKPOINTS(i,j,C,D)
  begin
    if C[i,j] is empty then return(D);
    MD := 2*D;
    for every cell K among 8 neighbors of C[i,j] do
      if K is nonempty then
        begin
          for every pair of points X in C[i,j] and Y in K do
            if Distance(X,Y) < MD then
              MD := Distance(X,Y);
        end;
    return({MD});
  end
```

Furthermore, to simplify the algorithm assume that the variables C, I1, I2, J1, and J2 in the procedure **FR-SEARCH** are global variables.

The algorithm is given as procedure **NN-SEARCH**. It takes as input three parameters D, R, and S, where D is radius of the subdivision, R is the root of the R-tree, S is the study area. Initially D is set to the Diameter(S).

```
Procedure NN-SEARCH(D,R,S)
  begin
    if 2*D > Diameter(S) then D := (Diameter(S))/2 ;
    M := min(FR-SEARCH(D,R,S));
    for i=I1 to I2 do
      for j=J1 to J2 do
        begin
          Case
            :D > M and C[i,j] contains more than 1 point:
              M := min{NN-SEARCH(M,R,E), M} ;
            :D <= M and C[i,j] contains more than 1 point:
              M := min{NN-SEARCH(D/2,R,E),M} ;
            :otherwise:
              do nothing;
          end of Case;
          {where E is a region corresponding to cell C[i,j]}
        end;
      return(M);
    end.
```

The technique employed in **NN-SEARCH** is that of divide-and-conquer [3]; the method is similar to one previously used by Yuval[63]. The procedure **NN-SEARCH** divides S into a set of subregions, then the shortest distance between any pair of subregions (P2) is calculated, using the procedure **FR-SEARCH**. The minimum distance between any two points is then obtained by solving (P1) recursively for each subregion.

Similar to **FR-SEARCH**, the procedure **NN-SEARCH** is superior to its previous counterpart in a number of aspects such as being more efficient and able to deal with objects of nonzero size.

4.2. Region Transformation

Two algorithms, namely translation and rotation, for the transformation on regions represented as an R/Q tree are developed in this section. Both translation and rotation by a multiple of 90° can be done very efficiently due to the structure of R/Q trees.

4.2.1. Region Translation

Since an R/Q tree is essentially an R-tree built on top of a set of quadtrees, region translation involves translating the R-tree and the quadtrees.

However, since all quadtrees are normalized with respect to translation, the description of each quadtree remains unchanged after translation. Furthermore, since the coordinates of the MBR of the lower level of the nodes in the R-tree is defined by using the address relative to their parent nodes, and the MBR of any node bounds all its descendent nodes, see Chapter 3, the only MBR to be modified after translation is that of the root node of the R-tree.

It is now clear that the translation of a region represented as an R/Q tree can be done by one disk access.

4.2.2. Region Rotation

Rotating a point (I,J) by a clockwise angle α about the origin produces a new point (I',J'):

$$\begin{cases} I'=I\cos\alpha+J\sin\alpha \\ J'=-I\sin\alpha+J\cos\alpha \end{cases} \tag{4.1}$$

Since both the R-tree and the quadtree are constructed by nodes parallel to the major axes, the rotation of a region represented as an R/Q tree is difficult. Fortunately, the size of a node either in the R-tree or in a quadtree is invariant under rotation by multiples of 90^o about the center of the image. Thus, decomposing a quadtree node into smaller ones and reconstructing the whole R-tree is not necessary. In the following, an algorithm for rotating an R/Q tree by 90^o about the center of the image is developed.

The rotation of point (I,J) by 90^o about the center of the image is achieved, however, by three processes in sequence: translate the origin to the center of the image; rotate the pixel by 90^o, using formula (4.1); translate the origin back.

Step 1. Translating the origin to the center of the image

Let the resolution of the image be n. After translating the origin to the center of the image, coordinates (I,J) become (X,Y) given by

$$\begin{cases} X=I-2^{n-1} \\ Y=J-2^{n-1} \end{cases} \tag{4.2}$$

Step 2. Rotating 90° in a clockwise direction

Let (X',Y') be the new pixel generated by (X,Y), then by (4.1) :

$$\begin{cases} X'=Y \\ Y'=-X \end{cases} \tag{4.3}$$

Step 3. Translating the origin back

Let (I',J') be the new pixel generated by (X',Y'), then

$$\begin{cases} I'=X'+2^{n-1} \\ J'=Y'+2^{n-1} \end{cases} \tag{4.4}$$

Combining the results of (4.2)-(4.4) yields the formula for rotating a point (I,J) 90° around the center of the image as follows:

$$\begin{cases} I'=J \\ J'=2^{n}-I \end{cases} \tag{4.5}$$

The rotation of a block, on the other hand, is achieved by rotating every pixel inside the block. However, since a block is determined solely by its vertex, only the vertex need be considered.

Let P be a block to be rotated and Q be the new block generated by rotating P. Let Plb, Plt, Prb, and Prt be the left bottom corner, left top corner, right bottom corner, and right top corner of P respectively. Then the clockwise rotation of 90° about the center of the image maps Plb, Plt, Prb, and Prt onto Qlt, Qrt, Qlb, and Qrb respectively.

4.2.2.1. Rotation of an R-tree Node

By **Definition 3.1** each node in an R-tree contains entries in which the first part is the smallest-rectangle that spatially covers the child node (or the object), with each minimum bounding rectangle represented as a four tuples $(X1,Y1,X2,Y2)$, where $(X1,Y1)$ and $(X2,Y2)$ are the left bottom corner and the right top corner of the corresponding minimum bounding rectangle.

Suppose $(X1,Y1,X2,Y2)$ is the description of the minimum bounding rectangle for node P and $(X1',Y1',X2',Y2')$ is the description of the minimum bounding rectangle for node Q, then

$$\begin{aligned} X1' &= Qlb_x \\ &= Prb_y \text{ (By (4.5))} \\ &= Y1. \\ Y1' &= Qlb_y \\ &= 2^n - Prb_x \text{ (By (4.5))} \\ &= 2^n - X2. \end{aligned}$$

and,

$$\begin{aligned} X2' &= Qrt_x \\ &= Plt_y \text{ (By (4.5))} \\ &= Y2. \\ Y2' &= Qrt_y \\ &= 2^n - Plt_x \text{ (By (4.5))} \\ &= 2^n - X1. \end{aligned}$$

4.2.2.2. Rotation of a Quadtree Node

Unlike R-trees, a node in a linear quadtree is represented by its key together with the resolution parameter. By **Definition 2.12-2.15**, the coordinates of the left bottom pixel of P is

$$(Plb_x, Plb_y) = (\mathrm{OD}(\mathrm{P.KEY}), \mathrm{EV}(\mathrm{P.KEY})).$$

Then:

$$\begin{aligned} Qlb_x &= Prb_y \text{ (By (4.5))} \\ &= Plb_y \\ &= \mathrm{EV}(\mathrm{P.KEY})) \\ Qlb_y &= 2^n - Prb_x \text{ (By (4.5))}. \end{aligned}$$

Since Prb_x is $2^s - 1$ pixels apart from Plb_x:

$$Qlb_y = 2^n - 2^s - \mathrm{OD}(\mathrm{P.KEY}) + 1.$$

Thus,

$$\begin{cases} Q.KEY = \mathrm{SH}(\mathrm{EV}(\mathrm{P.KEY}), 2^n - 2^s - \mathrm{OD}(\mathrm{P.KEY}) + 1). \\ Q.RES = P.RES. \end{cases}$$

The following algorithm rotates a region by 90^o in a clockwise direction around the center of the image. The algorithm consists of two parts corresponding to two procedure called **ROTATE** and **ROTATE1**[12]. The input of the procedure **ROTATE** is the root of the R/Q tree and an integer corresponding to the resolution of the image. It generates the new R-tree nodes recursively and at the leaf level of the R-tree invokes **ROTATE1** to generate the new quadtree nodes.

```
Procedure ROTATE(R,n)
  begin
    for i=1 to R.VALID do
      begin
        Temp := R.DESC[i].X1;
        R.DESC[i].X1 := R.DESC[i].Y1;
        R.DESC[i].Y1 := 2**n - R.DESC[i].X2
        R.DESC[i].X2 := R.DESC[i].Y2;
        R.DESC[i].Y2 := 2**n - Temp;
        if R.CLASS='leaf' then
          ROTATE1(R.DESC[i].POINTER,n)
        else
          ROTATE(R.DESC[i].POINTER,n);
      end
  end.

Procedure ROTATE1(Head,n)
  begin
    Q-list:={};
    while not eof do
      begin
        N := NEXT-P(Head);
        I := EV(N.KEY) + offset(X);
        J := 2**n - 2**N.RES - OD(N.KEY)+1 + offset(Y);
        add pair <SH(I,J),N.RES> to Q-list;
      end;
    reconstruct B-tree or hashing table from Q-list;
  end;
```

4.3. Set-theoretic Operations

Direct spatial search on pictorial databases often requires performing set operations. Prime examples are spatial databases and (CAD) applications [47].

Set operations may occur either at the primitive level or at the object level. In the former case, a set of primitives for representing objects is obtained by the union (intersection) of two regions. Since the low level structure of an R/Q tree is that of a linear quadtree, the set operation algorithms for linear quadtrees directly support efficient set operations for an R/Q tree at the primitive level and these algorithms can be easily constructed from the algorithms in [12,36].

In the following, however, the set algorithms are presented at the object level. The problem is precisely formulated as follows:

Given two sets of pictorial objects, find the union (intersection).

4.3.1. Union Operation

The R/Q tree representation is especially efficient for performing the union of several sets of spatial objects since only the high level structure, the R-tree, needs to be examined.

Let R1 and R2 be R-trees for each set of spatial objects. The union of the two trees is obtained by the following procedure termed **UNION**. For any given root of an R-tree, R, the function **NEXT-O**(R) returns the next objects in the R-tree. Since the order of the objects is not important, the function **NEXT-O** can be accomplished simply by following a chain of linked lists. Furthermore, the test condition for the while loop can also be performed efficiently, using the algorithm **SEARCH**.

```
Procedure UNION(R1,R2)
  begin
    while not eof(R1) do
      begin
        O := NEXT-O(R1);
        if O is not in R2 then
          INSERTION(R2,O);
      end;
    return(R2);
  end.
```

4.3.2. Intersection Operation

Intersection of two sets of pictorial objects can be accomplished by a slight modification of the previously stated algorithm union. The same objects in both sets are picked up, then procedure **PACK** is used to construct an R-tree for them.

```
Procedure INTERSECTION(R1,R2)
  begin
     LIST := { };
     while not eof(R1) do
        begin
           O := NEXT-O(R1);
           if O is in R2 then
           add O into LIST;
        end;
     return(PACK(LIST));
  end
```

4.4. Conclusion

In summary, a set of algorithms has been developed in this chapter to justify the proposed data structure. Two issues are addressed in this chapter: the spatial data search and manipulation. The following three main characteristics of an R/Q tree supports the achievement of the goal: being able to efficiently support the retrieval of spatial data, the hybrid nature of the structure, and faciliting a compact region representation.

Chapter 5
CONCLUSIONS

This thesis is concerned with data structures for spatial data and in particular search and manipulation. Two types of data structures, one for region representation and another for spatial data search, were reviewed. In the first class, data structures such as chain codes, medial axis transforms and quadtrees were discussed with emphasis on linear quadtrees. Furthermore, for large images, two file schemes: a B^+-tree and order preserved extendible hashing (or its variations) for organizing a linear quadtree were presented and compared. Hashing schemes were shown to be superior to B^+-tree schemes for most cases since they save memory space on one hand, and allow $O(1)$ direct access in retrieval of a record on the other hand. A B^+-tree, however, is shown to be useful when data distribution is uneven since it supports retrieval of a record with $O(logn)$ disk accesses in the worst case, where n is the number of the records in the file. In the second class, data structures such as cell methods, quadtrees, k-d trees, k-d-B trees, index intervals, and grid files were shown to be inadequate in several respects for handling object oriented spatial searches for multi-dimensional spatial data of nonzero size. On the other hand, R-trees were shown to be a better choice for such applications.

If spatial data of nonzero size is distributed in a multi-dimensional space, then the second type of structure should also be able to represent regions, In other words, a data structure which handles features of both types is mandatory. For this reason, a hybrid data structure called an R/Q tree for region representation has been presented, and various operations on images have been developed to support the data structure.

The hybrid data structure has been developed based on both a linear quadtree and an R-tree. R/Q trees are intended to combine the best features of both, and to have a more efficient representation in terms of both the storage requirement and in addition the cost for various operations.

At low level, an R/Q tree retains the advantages of linear quadtrees, i.e., it is very compact in terms of the storage requirement, and facilitates low level image operations as well as the calculation of various geometrical properties of an object. On the other hand, from the structure of an R-tree, an R/Q tree can retrieve spatial data quickly according to its spatial location. Empirical results demonstrate that range search in an R/Q tree can be performed efficiently. Therefore, at a high level description, an R/Q tree outperforms previous data structures for region representation in the sense that it is capable of handling regions which contain objects having nonzero size. For instance, by using an R/Q tree, many previous algorithms for point data processing can be extended to deal with spatial objects with nonzero size. In particular, the algorithm for fixed-radius near neighbor searching is superior to the one proposed by Bentley [7]. By the same token, the algorithms for nearest neighbor finding and set operations share the same advantage.

Another important feature of the proposed scheme is that an R-tree represents a region as a collection of normalized quadtrees, each of which corresponds an object in the region. Many advantages are gained from the nature of this structure.

First of all, an R/Q tree is less sensitive to the distribution of data. Since most of the "dead space" is not there, data distribution for each object is expected to be even. Furthermore, any non-uniformity of the data distribution affects the file structure only locally rather than for the entire study area. Thus, in most cases, a hashing scheme can be applied to organize the linear quadtree data.

Secondly, the representation for each object is position independent. As a result translation of a region represented by an R/Q tree can be accomplished by one disk access as shown in Chapter 4, and object comparisons such as template matching and pattern recognition can be performed efficiently.

Finally, an R-tree entails space efficiency. The fact that in most cases a hashing scheme can be adopted implies a considerable saving in space for constructing the index which is needed if a B-tree scheme is chosen for the image. In addition, if the same object appears at different positions in the image, only one copy of the description is necessary.

As a disk oriented scheme, an R/Q tree is particularly useful in many real world applications where the spatial data is voluminous since nodes in the R/Q tree corresponding disk pages of reasonable size can be chosen to produce good performance. With smaller nodes, however, R/Q trees should also be effective as a main memory scheme since CPU performance would be comparable with previous data structures such as quad-trees.

Although R-trees have many appealing advantages, they are not without problems. For instance, an increasing number of applications, such as information or reservation systems, require concurrent access to a file system. Concurrency control is complicated in R/Q tree structures because the root is a bottleneck shared by all access paths. If a process has the potential of modifying the data structure near the root (such as insertion or deletion in an R-tree), other processes may be slowed down by the adherence to locking protocols even if they access disjoint data. Another problem with the structure is that some deterioration in performance during and after update. To prevent performance deterioration, any modification made to the primitives of an object necessitates a re-insertion in the R-tree; furthermore, the deletion of an object in the region may involve many re-insertions to take care of underfull nodes in the R-tree. Thus, the maintenance of an R/Q tree can be difficult.

Future work can be divided into two general classes: theoretical developments, and practical implementations. In the former category, the expected performance of an R/Q tree should be investigated. In addition, as hash methods have the property that access paths to separate buckets are disjoint, thus allowing simpler concurrent control protocols, the possibility of using a hashing scheme to index spatial objects of nonzero size should also be studied. The problem is how to represent a rectangular block as a k-dimensional key in a way that is convenient for range search using a hashing scheme. In the later category, simulation based on real data should be attempted.

Bibliography

[1] Abel, D.J. "A B-Tree Structure for Large Quadtrees", *Computer Vision, Graphics, and Image Processing,* Vol.27, pp. 19-31, Jul. 1984.

[2] Abel, D.J. and Smith J.L. "A Data Structure and Algorithm Based on a Linear Key for Rectangle Retrieval Problem", *Computer Vision, Graphics, and Image Processing,* Vol.24, pp. 1-13, Oct. 1983.

[3] Aho, A., Hopcroft, J. et. al. "The Design and Analysis of Computer Algorithms", *Reading, MA: Addison-Wesly,* 1974.

[4] Bauer, M.A. "Note on Set Operations on Linear Quadtree", *Computer Vision Graphics Image Process.*, Vol. 29, pp. 248-258, Feb. 1985.

[5] Bayer, R. and McCreight, E. "Organization and Maintainnance of Large Ordered Indices", *Proc. 1970 ACM-SIGFIDET Workshop on data Description and Access,* Houston, Texas, pp. 107-141, Nov. 1970.

[6] Bentley, J.L. "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM,* Vol.18-19, pp. 509-517, Sep. 1975.

[7] Bentley, J.L., Friedman, J.H. and Williams, E.H. "The Complexity of Fixed-radius Near Neighbor Searching", *Inf. Proc. Lett.*, Vol. 6, pp. 209-212, Dec. 1977.

[8] Bentley, J. L. and J. H. Friedman, "Data Structures for Range Searching", *Computing Surveys,* Vol. 11, pp. 397-409, Dec. 1979.

[9] Burkhard, Walter, A. "Interpolation-Based Index Maintenance", *BIT*, Vol. 23, pp. 274-294, 1983.

[10] Chang, S.K., and Kunii, L.K., "Pictorial Database Systems", *IEEE Computer*, Vol. 14, Nov. 1981.

[11] Comer, D. "The Ubiquitous B-tree", *Computing Surveys*, Vol. 11, pp. 121- 138, Jun. 1979.

[12] Davis, W.A. and Wang, X. "A New Approach to Linear Quadtrees", *Technical Report TR 84-9,* University of Alberta, 1985.

[13] Davis, W.A. and Haung, C. H., "File Organization Schemes For Geomrtric Data", *Technical Report TR 85-14* University of Alberta, 1985.

[14] Dyer, C.R., Rosenfeld, A. and Samet, H. "Region Representation: Boundary Codes from Quadtrees", *Comm. ACM*, Vol. 23, pp. 171-179, Mar. 1980.

[15] Dyer, C.R. "The Space Efficiency of Quadtrees", *Comput. Graphics and Image Process.*, Vol. 19, pp. 335-438, Aug. 1982.

[16] Fagin,R., J.Nievergelt, N.Pippenger and H.R.Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM Trans.* Mathematical Software, Vol.3, No.3, pp. 209-266, Sep. 1977.

[17] Finkel, R.A. and Bentley, J.L. "Quadtrees: a Data Structure for Retrieval on Composite Keys", *ACTA-Informatica*, Vol. 4, pp. 1-9, 1974.

[18] TREE", Bericht Nr. 71, Eidgenossische Technische Hochschule (ETH), Zurich, Switzerland, 1983.

[19] Gargantini, I. "An Efficient Way to Represent Properties of Quadtrees", *Comm. ACM*, Vol. 25, pp. 905-910, Dec. 1982.

[20] Gargantini, I. "Translation, Rotation and Superposition of Linear Quadtrees", *Int. J. Man-Mach. Stud.*, Vol. 18, pp. 253-263, Mar. 1983.

[21] Gargantini, I. "Detection of Connectivity of Regions Using Linear Quadtrees", *Comp. & Math. With Appl.*, Vol. 8, pp 319-327, 1982.

[22] Gillespie R. and Davis W.A. "Tree Data Structures for Graphics and Image Processing", *Proc. Canadian Man Computer Communication Society Conference*, Waterloo, Ontario, pp. 155-162, 1981.

[23] Guttman, A. "R-Trees: a Dynamic Index Structure for Spatial Searching", *ACM-SIGMOD Proc. of Annual Meeting*, Vol. 14, pp. 47-57, 1984.

[24] Hunter, G.M. and Steiglitz, K. "Operations on Images Using Quadtrees", *IEEE Trans. Pattern Analy. & Mach. Intell.*, Vol. PAMI-1, pp. 145-153, Jul. 1979.

[25] Hunter, G.M. and Steiglitz, K. "Linear Transformation of Pictures Represented by Quadtreee", *Comput. Graphics and Image Process.*, Vol. 10 pp. 289-296, Jul. 1979.

[26] Jones, L. and Iyengar, "Space and Time Efficient Virtual Quadtrees", IEEE Trans. Pattern Anal. Mach. Intell., Vol. 6, No.2, pp.244-247, 1984.

[27] Kinger, A. and Dyer, R.C. "Experiments on Picture Representation Using Regular Decomposition", *Comput. Graphic and Image Processing,* Vol. 5, pp. 68-105, Mar. 1976.

[28] Klinger, A., Rhodes, M.L. and Omolayole, J. "Image Data Organization", *Proc. of San Diego Biomedical Symp.,* Vol. 5, New York: Academic press, pp. 175-180, 1976.

[29] Klinger, A., Rhodes, M.L. and Omolayole, J. "Organization and Access of Image Data by Areas", *IEEE Trans. Pattern Analy. & Mach. Intell.,* Vol. 20, pp. 72-81, 1982.

[30] Knuth, D.E. "The Art of Computer Programming", Vol. 1&3, Addison-Wesley Publishing company, 2nd Edition, 1973.

[31] Larson, Per-Ake, "Dynamic Hashing", *BIT*, Vol.18, pp. 184-201, 1978.

[32] Li, M., Grosky, W.I. and Jain, R. "Normalized Quadtrees with Respect to Translations", *Comput. Graphics and Image Process.*, Vol. 20, pp. 72-82, Sep. 1979.

[33] Litwin,Witold, "Virtual Hashing: A Dynamically Changing Hashing", *Proceedings, 4th International Conference on Very Large Data Bases*, pp. 517-523, 1978.

[34] Litwin,Witold, "Linear Hashing: A New Tool for File and Table Addressing", *Proceedings 6th International Conference on Very Large DataBases* pp. 212-223, 1980.

[35] Nievergelt,J., H.Hinterberger and K.C.Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure", *ACM Trans. DataBase Systems,* Vol. 9, pp. 38-71, Mar. 1984.

[36] Mark, D.M. and Lauzon, J.P. "Linear Quadtrees for Geographic Information Systems", *Proc. International Symposium on Spatial Data Handling,* Vol. 2, pp. 412-430, Zurich, Switzerland, Aug. 1984.

[37] Morton, G.M. "A Computer Oriented Geodetic Data Base, and a New Technique in File Sequencing", *IBM Canada Limited, Unpublished Report,* March 1966,

[38] Peuquet, D.J. "Data Structures for Spatial Data Handling", *International Symposium on Spatial Data Handling,* Background Material to Workshop W2, Zurich, Switzerland, 20-24, Aug. 1984.

[39] Pfaltz, J.L. and Rosenfeld, A., "Computer Representation of Planar Regions by their Skeletons", *Communications of the ACM.* Vol. 10(2), 1967, pp. 119-122.

[40] Robinson, J.T. "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes", *ACM-SIGMOD Conference Proc.,* pp. 10-18, Apr. 1981.

[41] Rosenfeld, A. and Pfaltz, J. L., "Sequential Operations in Digital Picture Processing", *Journal of the ACM,* Vol. 13, pp. 471-496, 1976.

[42] Rosenfeld, A. and Kak, A.C., "Digital Picture Processing", *Academic Press,* New York, 1976.

[43] Rosenfeld, A., "Tree Structures for Region Representation", *Comput. Graphics and Image Process.*, Vol. 11, pp. 137-150, May 1980.

[44] Rosenfeld, A. "Survey Picture Processing: 1981", *Comput. Graphics and Image Process.*, Vol. 19, pp. 35-75, 1982.

[45] Rosenfeld, A. "Survey Picture Processing: 1982", *Comput. Graphics and Image Process.*, Vol. 22, pp. 339-387, 1983.

[46] Rosenfeld, A. "Image Analysis: Progress, Problems and Prospects", *Proceedings of the 6th International Conference on Pattern Recognition.*, Vol. 1, pp, 7-15, Munich, Germany, Oct. 1982.

[47] Roussopoulos, N. and Leifker, D. "Direct Spatial Search on Pictorial DataBases Using Packed R-Trees", *Proc. of ACM-SIGMOD International Conference on Management of Data*, Vol. 14, pp. 17-31, May 1985.

[48] Samet, H. "Region Representation: Quadtrees from Binary Arrays", *Comput. Graphics and Image Process.*, Vol. 13, pp. 88-93, 1980.

[49] Samet, H. "Region Representation: Quadtrees from Boundary Codes", *Comm. ACM*, Vol. 23, pp. 163-170, 1980.

[50] Samet, H. "An Algorithm for Converting Raster to Quadtrees", *IEEE Transaction on PAMI*, Vol. 3(1), pp. 93-95, 1981.

[51] Samet, H. "Quadtrees and Medial Axis Transforms", *Procedings of the 6th International Conference on Pattern Recognition*, Vol. 1, pp. 184-187, Munich, Germany, Oct. 1982.

[52] Samet, H. "The Quadtree and Related Hierarchical Data Structures", *Computing Surveys.*, Vol. 16, pp. 187-260, Jun. 1984.

[53] Shaffer, C. A. and Samet, H. "Optimal Quadtree Construction Algorithms", *Computer Vision, Graphics and Image Processing*, Vol. 27, pp. 402-419, Oct. 1987.

[54] Samet, H. and et. al., "Recent Developments in Linear Quadtree-Based Geographic Information Systems", Image And Vision Computing, Vol. 5, No. 3, pp. 187-197, 1987.

[55] Shapiro, L.G., "Data Structures for Picture Processing: A survey", *Computer Graphics and Image Processing*, Vol. 11, pp.162-184, 1979.

[56] Shneier, M. "Note: Calculations of Geometric Properties Using Quadtrees", *Comput. Graphics and Image Process*, Vol. 16, pp. 296-302, Jul. 1981.

[57] Stonebraker, et al. "Application of Abstract Data types and Abstract indices", *Engineering Design and Applications, Database Week, ACM-SIGMOD*, San Jose, May pp. 23-26, 1983.

[58] Tamminen, Markku, "Order Preserving Extendible Hashing and Bucket Tries", *BIT*. Vol. 21, pp. 419-435, 1981.

[59] Tanimoto, S.L. "A Comparison of Some Image Searching Methods", *Proc. 1978 IEEE Computer Soc. Conf. on Pattern Recognition and Image Processing*, pp. 280-286, Jun. 1978.

[60] Tang, G.Y., "A logical Data Organization for the Database of Pictures an Alphanumerical Data," *IEEE Proc. of the Workshop on Picture Data Description and Management*, Asilomar, California, Aug. 1980.

[61] K.C. Wong and M. Edelberg, "Interval Hierachies and Their Application to Predicate Files", *ACM Trans. On Database Systems*, Vol. 1-2, pp. 223- 232, Sep. 1977.

[62] Wright, W.E., "Some Average Performance Measures for the B-tree", *ACTA informatica* Vol. 21, pp.541-557, 1985.

[63] Yao, A., "On Random 2-3trees", *ACTA informatica* Vol. 9, pp. 159-170, 1978.

Appendix A

The Storage Utilization of R-Trees

The storage utilization ratio of R-trees can be controlled by setting parameter m, $\lfloor M/2 \rfloor \le m \le M$, where M is the number of entries in each node. To compare the storage utilization of an R-tree with a B-tree, however, a special case: $m=M/2$ is studied.

Obviously, the minimum storage utilization is 50%(excluding the root) which is exactly the same as that of B-trees. In the following, an average performance measure for R-trees with respect to storage utilization is presented.

The result of the storage utilization of an R-tree is obtained with the aid of the following lemma.

Lemma 1: Let E and I be the number of external nodes and the number of internal nodes in an R-tree, α be the average fanout per internal node. The following relationship is true

$$(\alpha-1)I=E-1.$$

Proof: Suppose the height of the R-tree is h, the number of node in the ith level of the tree is N_i, the average fanout per node among the nodes from the ith level is β_i, and the average fanout among the nodes from the first level to ith level is α_i.

Obviously,

$$\alpha_i=\frac{\sum_{j=1}^{i}\beta_j N_j}{\sum_{j=1}^{i}N_j} \qquad \text{(EQ. 1.a)}$$

$$N_i=\beta_{i-1}N_{i-1} \qquad \text{(EQ. 1.b)}$$

the lemma will be proved by induction on h as follows.

Let h=2, then

$$I=1$$

$$E = \text{the fanout index of the root node}$$

Since there is only one internal node,

$$\alpha = \text{the fanout index of the root node}$$

Therefore,

$$(\alpha-1)I=\alpha-1=E-1.$$

Suppose the formula is true for h=k-1, it is necessary to prove it is also true for h=k

Since an R-tree is height balanced, any node from first level to (k-1)th level is an internal node and all nodes at level k are external nodes; that is

$$I=\sum_{j=1}^{k-1}N_j$$

$$E=N_k$$

$$\alpha=\alpha_{k-1}=\frac{\sum_{j=1}^{k-1}\beta_j N_j}{I}.$$

Therefore,

$$(\alpha-1)I=\sum_{j=1}^{k-1}\beta_j N_j-I$$

$$=\sum_{j=1}^{k-2}\beta_j N_j+\beta_{k-1}N_{k-1}-\sum_{j=1}^{k-2}N_j-N_{k-1}$$

$$=(\alpha_{k-2}-1)\sum_{j=1}^{k-2}N_j+\beta_{k-1}N_{k-1}-N_{k-1}\text{ (By EQ. 1.a).}$$

By induction hypothesis,

$$(\alpha-1)I=N_{k-1}-1+\beta_{k-1}N_{k-1}-N_{k-1}.$$

Therefore,

$$(\alpha-1)I=\beta_{k-1}N_{k-1}-1=N_k-1\text{ (By EQ. 1.b).}$$

Q.E.D.

Suppose the R-tree has been generated from a sequence of random insertions. Let n denote the number of objects in the tree, and B_{in}, i=0,...,m, denote the expected number of nodes of fanout m+i on the bottom level of the tree.

Suppose an insertion is to be made randomly into the region, then the probability of an object appearing at any position in the study area is equally likely. Furthermore, assume that the probability of an object close to any other object is the same. Although the later assumption does not take the size and shape of the objects into account, it seems plausible because for large images the position of the objects tends to be a dominant factor in determining the closeness of the spatial objects, and the size and shape of the objects are relatively less significant.

On the other hand, according to the procedure **INSERTION** in Chapter 3, the object is always put into node that will lead to the least enlargement. This implies that the objects that are spatially close to each other will likely be clustered together. Hence, the probability that a random insertion will fall in a node of fanout i+m is $\frac{(m+i)B_{in}}{n+1}$, by the assumption. Furthermore, a insertion will result in a decrease of 1 in the number of nodes of fanout i+m, and increase of 1 in the number of nodes of fanout m+i+1. In case of a split, the insertion will result in a decrease of 1 in the number of nodes of fanout 2m, and an increase of 1 in the number of nodes of fanout m+1 and of 1 in the number of nodes of fanout m respectively.

Hence, have the following relationships follow:

$$B_{0,n+1}=B_{0n}+\frac{1}{n+1}(B_{mn}2m-B_{0n}m)$$

$$B_{1,n+1}=B_{0n}+\frac{1}{n+1}\left[B_{mn}2m-B_{0n}m-B_{1n}(m+1)\right]$$

$$B_{i,n+1}=B_{in}+\frac{1}{n+1}\left[B_{i-1,n}(m-1+i)-B_{in}(m+i)\right] \quad i=2,\ldots,m$$

Denoting by $B(n)$ the $m+1$ component vector $(B_{0n}, \ldots, B_{mn})$, the above equations can be written in matrix notation as

$$B(N) = \left[I + \frac{1}{N}D\right] B(N-1) \qquad \text{(EQ. 2)}$$

where I is the $(m+1)\times(m+1)$ identity matrix, and D is defined by

$$D = \begin{bmatrix} -m & 0 & 0 & \cdots & 2m \\ m & -(m+1) & 0 & \cdots & 2m \\ 0 & m+1 & -(m+2) & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & 2m-1 & -2m \end{bmatrix}.$$

with zeroes elsewhere.

To solve recurrence $B(N)$, define a m+1 component vector $b(n)=(b_{0n}, \ldots, b_{mn})$ by

$$b_{in} = \frac{B_{in}}{n+1}. \qquad \text{(EQ. 3)}$$

Equations (2) and (3) yield to the following relation

$$b(N) = \left[I + \frac{1}{N+1}(D-I)\right] b(N-1).$$

The characteristic polynomial $C(\lambda)$ of D-I is computed to be

$$C(\lambda) = (-1)^{m+1}(\lambda+2m+1)\left[\prod_{i=1}^{m}(\lambda+m+i) - \prod_{i=1}^{m}(m+i)\right].$$

Therefore as shown in [28], the eigenvalues are 0 and m other distinct values having negative real parts and th[illegible] xists a column vector $u=(u_0, \ldots, u_m)^T$ such that

$$\lim_{n\to\infty} b(n) = u \qquad \text{(EQ. 4)}$$

and

$$(D-I)u = 0.$$

That is:

$$-(m+1)u_0 + 2mu_m = 0$$

$$\mu_0 - (m+2)u_1 + 2mu_m = 0$$

$$(m+i-1)u_{i-1}-(m+i+1)u_i=0 \quad \text{for } 1<i\leq m.$$

Solving the recurrence yields

$$\begin{cases} u_i=\dfrac{2m(2m+1)}{(m+i)(m+i+1)}u_m, \\ u_0=\dfrac{2m}{m+1}u_m \end{cases} \quad i=1,\ldots,m. \tag{EQ. 5}$$

On the other hand, the total number of objects n satisfies the relationship is:

$$n=\sum_{i=0}^{m}(m+i)B_{in}.$$

Thus

$$\sum_{i=0}^{m}(m+i)\frac{B_{in}}{n+1}=\frac{n}{n+1}$$

$$\sum_{i=0}^{m}(m+i)b_{in}=\frac{n}{n+1}$$

$$\lim_{n\to\infty}\sum_{i=0}^{m}(m+i)b_{in}=\sum_{i=0}^{m}(m+i)u_i=1.$$

Substituting u_i with equation (5) yields:

$$1=\sum_{i=0}^{m}\left(\frac{2m(2m+1)}{(m+i+1)}u_m\right)$$

$$=2m(2m+1)u_m\sum_{i=0}^{m}\frac{1}{m+i+1}$$

$$=2m(2m+1)(H(2m+1)-H(m+1))u_m,$$

where

$$H(k)=\sum_{i=1}^{k}\frac{1}{i}, \text{ for } i\geq 1.$$

Therefore

$$u_m=\frac{1}{2m(2m+1)[H(2m+1)-H(m+1)]}$$

and

$$\begin{cases} u_i=\dfrac{1}{(m+i)(m+i+1)(H(2m+1)-H(m+1))} \\ u_0=\dfrac{1}{(m+1)(2m+1)(H(2m+1)-H(m+1))} \end{cases} \quad i=1,\ldots,m. \tag{EQ. 6}$$

The storage utilization of of an R-tree is calculated by following formula

$$S=\frac{\sum_{i=0}^{m}(m+i)B_{in}+\alpha R}{2m\sum_{i=0}^{m}B_{in}+2mR}$$

where R denotes the number of internal nodes in the R-tree, and α denotes is the average number of fanout per internal node.

Hence, we have the following relationship, by **Lemma 1**.

$$(\alpha-1)R=\sum_{i=0}^{m}B_{in}-1,$$

$$R=\frac{\sum_{i=0}^{m}B_{in}-1}{\alpha-1},$$

$$S=\frac{\sum_{i=0}^{m}(m+i)B_{in}+\frac{\alpha(\sum_{i=0}^{m}B_{in}-1)}{\alpha-1}}{2m\sum_{i=0}^{m}B_{in}+\frac{2m\left[\sum_{i=0}^{m}B_{in}-1\right]}{\alpha-1}}$$

$$=\frac{\frac{1}{m}\sum_{i=0}^{m}(m+i+\frac{\alpha}{\alpha-1})B_{in}-\frac{\alpha}{m(\alpha-1)}}{2\left[1+\frac{1}{\alpha-1}\right]\sum_{i=0}^{m}B_{in}-\frac{2}{\alpha-1}}$$

Since every node except possibly the root must contain at least m keys, $\alpha\geq m$ is generally true for large R-trees. Therefore,

$$0\leq\lim_{n,m\to\infty}\frac{1}{\alpha-1}\leq\lim_{n,m\to\infty}\frac{1}{m-1}=0,$$

$$\lim_{n,m\to\infty}S=\lim_{n,m\to\infty}\frac{\frac{1}{m}\sum_{i=0}^{m}(m+i+1)B_{in}}{2\sum_{i=0}^{m}B_{in}}$$

$$=\lim_{n,m\to\infty}\frac{\frac{1}{m}\sum_{i=0}^{m}(m+i+1)b_{in}}{2\sum_{i=0}^{m}b_{in}}\quad\text{(By EQ. 3)}$$

$$= \lim_{m \to \infty} \frac{\frac{1}{m}\sum_{i=0}^{m}(m+i+1)u_i}{2\sum_{i=0}^{m}u_i} \quad \text{(By EQ. 4)}$$

$$= \lim_{m \to \infty} \frac{\sum_{i=0}^{m}\frac{m+i+1}{(m+i)(m+i+1)}}{2m\sum_{i=0}^{m}(m+i)(m+i+1)} \quad \text{(By EQ. 5)}$$

$$= \lim_{m \to \infty} \frac{\sum_{i=0}^{m}\frac{1}{(m+i)}}{2m\sum_{i=0}^{m}\left[\frac{1}{(m+i)}-\frac{1}{(m+i+1)}\right]}$$

$$= \lim_{m \to \infty} \frac{H(2m)-H(m-1)}{2m\left[\frac{1}{m}-\frac{1}{(2m+1)}\right]}$$

$$= \lim_{m \to \infty} \frac{H(2m)-H(m-1)}{\frac{2(m+1)}{(2m+1)}}$$

$$= \lim_{m \to \infty} [H(2m)-H(m-1)]$$

$$= \ln 2.$$

Therefore,

$$\lim_{n,m \to \infty} S = \ln 2 = 0.69315$$

That is the storage utilization of an R-trees is comparable with that of a B-tree[59].

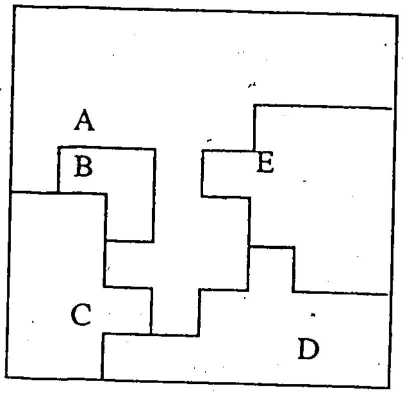
Appendix B

Comparisons Between an R/Q tree and a B/Q tree

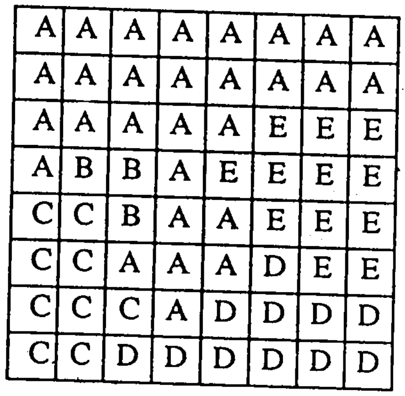
To highlight the advantages of an R/Q tree in spatial search, two typical kinds of orthogonal range query are considered with respect to a selected data set. The comparison is made between an R/Q tree and a quadtree with a B tree incorporated (denoted as a B/Q tree).

Consider the data shown in Figure B.1-3 and a rectangular search area, I, in Figure B.4. Suppose the R/Q tree and the B/Q tree are used as the implementation paradigm respectively. Also assume that the maximum fanout M and minimum fanout m are 3 and 1 respectively. This would be highly unlikely in any real application, but this assumption makes the data structures possible to be handled manually. Furthermore, to simplify the discussion, the low level quadtrees in the R/Q tree are not normalized. Finally, to be convincing, B/Q trees are adopted to organize the low level quadtrees in the R/Q tree instead of hashing schemes.

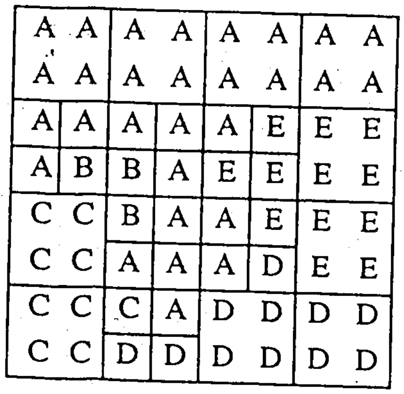
Following quadtree blocks are obtained from the data:
<0,1,C>,<4,1,C>,<8,0,D>,<9,0,C>,<10,0,D>,<11,0,A>,
<12,0,A>,<13,0,B>,<14,0,A>,<15,0,A>,<16,0,A>,<17,0,A>,
<18,0,B>,<19,0,A>,<20,1,A>,<24,0,B>,<25,0,A>,<26,0,A>,
<27,0,A>,<28,1,A>,<32,1,D>,<36,0,A>,<37,0,A>,<38,0,D>,
<39,0,E>,<40,1,D>,<48,0,E>,<49,0,A>,<50,0,E>,<51,0,E>,
<52,1,A>,<56,1,E>,<60,1,A>.



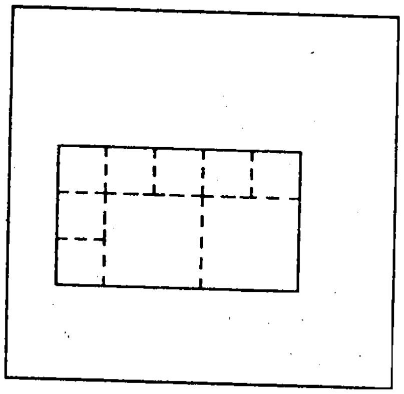
(1) The Original Data set



(2) An Array Representation



(3) A Maximal Block Representation



(4) A Rectangular Search Area

Figure B.1. An Data Set And a Search Area

The corresponding R/Q tree and B/Q tree, therefore, are contained in Figure B.2 and Figure B.3 respectively. In this example, the storage requirement for the R/Q tree is 20 pages while the storage requirement for the B/Q tree is 19 pages.

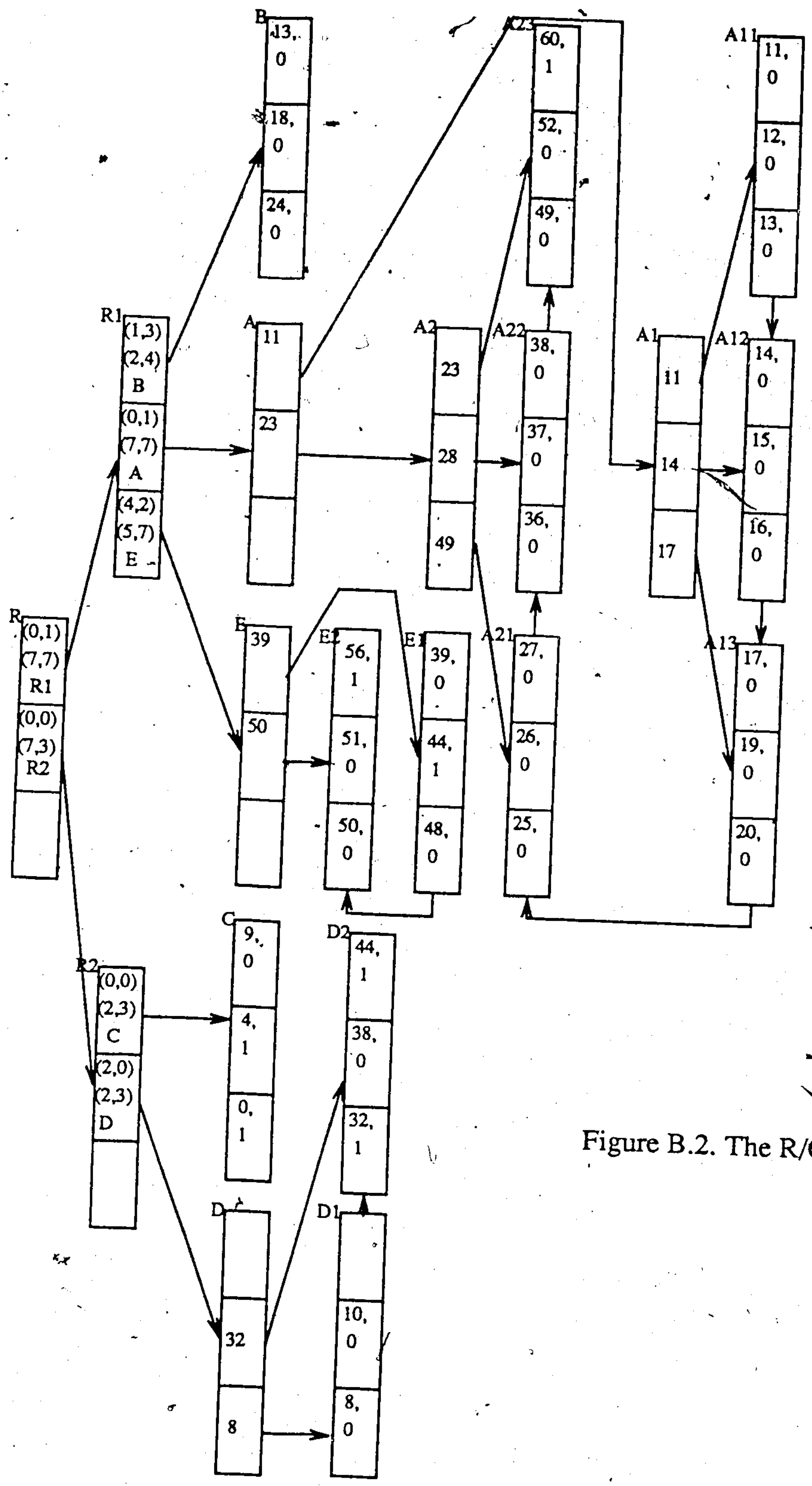


Figure B.2. The R/Q tree

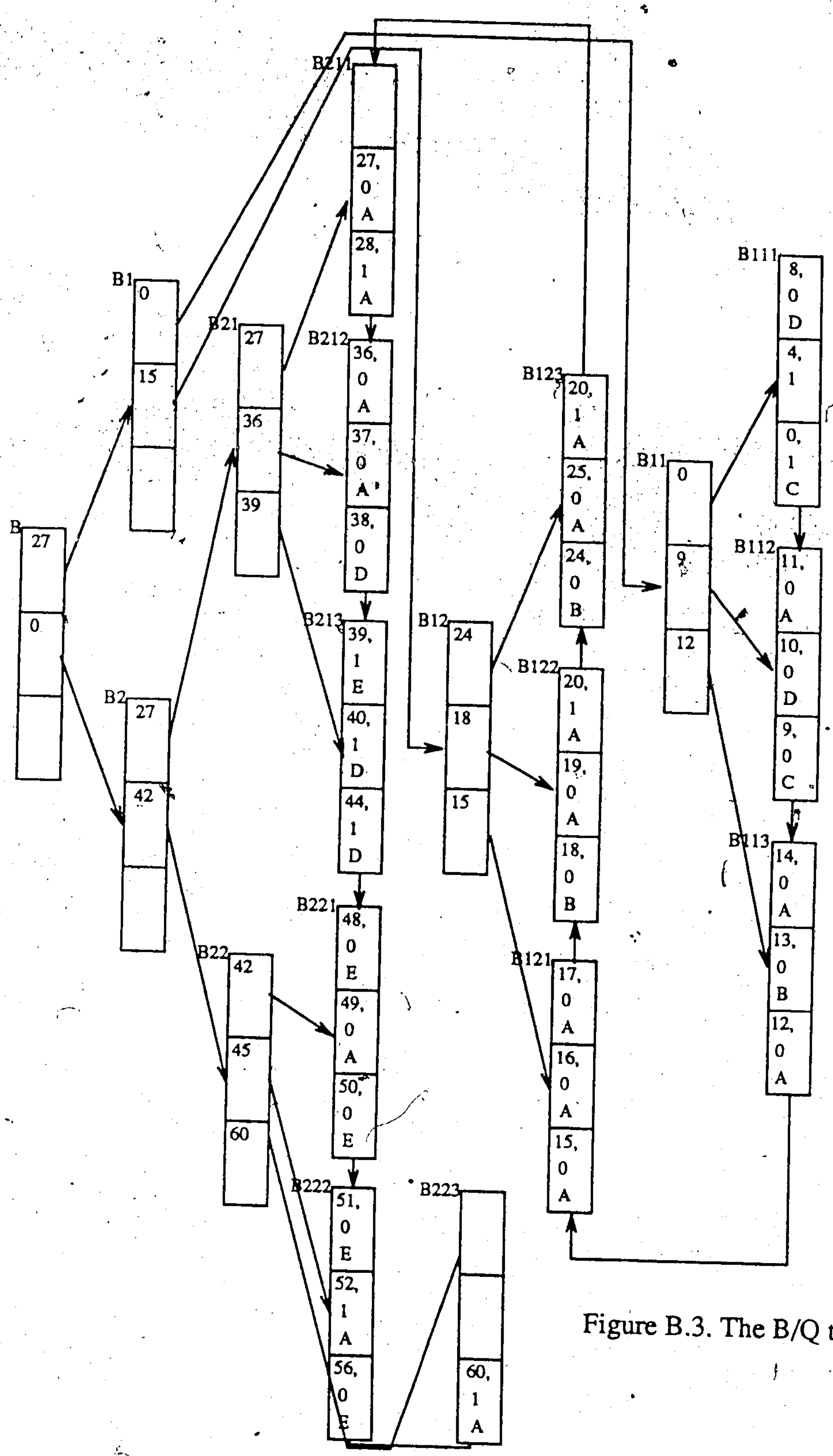


Figure B.3. The B/Q tree.

Following queries, in particular, are compared:

1). Find all objects contained in I.

2). Find all objects intersect I.

Let I^* = {<6,0>, <7,0>, <12,1>, <18,0>, <24,0>, <25,0>, <27,0>, <36,1>, <48,0>, <50,0> } be the set of maximum blocks obtained by decomposing of I, see Figure B.4. Let A @ $MBR(N)$ denote the set of blocks in set A that intersects $MBR(N)$, where $MBR(N)$ is the MBR of node N in the R/Q tree..i.e. I^* @ $MBR(B)$ = {<7,0>,<12,1>,<18,0>,<24,0>}.

F[illegible] Query:

R/Q tree

The search starts at the root node R. Since the MBR of the first entry of R intersects I, node R1 is examined. Being a leaf node of the R-tree in the R/Q tree, R1 is treated differently then it is a nonleaf node(i.e. node R). The following processes are performed on R1:

(1) Examine the first entry of R1:

Although I overlaps object A, MBR(A) does not contained in I. The fact that MBR(A) is the minimal bounding rectangle of A implies that A cannot be covered by I. Thus the search failed.

(2) Examine the second entry of R1:

Since I covers R(B), object B is contained in I. Therefore, B is a candidate.

(3) Examine the third entry of R1:

This result is similar to (1), since I does not cove E completely, the search failed.

Upon finishing the third entry of R1, the process backtracks to an upper level. Therefore, the second entry of R node is then examined. Since the MBR of the second entry of R overlaps I, subtree R1 is explored in the same fashion as previously described. If, however, the intersection between the MBR of an entry and I is empty, then the whole subtree associated with the entry is ignored.

By tracing, the search process, it is not difficult to see that the total number of disk access is 3 and the answer is B. Note the low level description of the structure, the linear quadtrees in the R/Q tree, is untouched.

B/Q tree:

Since primitives are organized solely by their keys, the whole structure is a miscellaneous collection of the primitives from the objects in the study area, see Figure B.4. As the global object information is needed for the solution of the problem, completely reconstructing the objects from their primitives or brutal searching is necessary. In other words, the B/Q tree does not conveniently lend itself to the query.

Second Query:

R/Q tree:

In solving this problem, only local information is necessary. However, the global information stored in the high level structure, the R-tree, can be effectively utilized to narrow the search space or even to obtain the answer as well. On the other hand, since objects may have arbitrary shape, the low level quadtree structures need to be searched in some cases. To be specific, the statement marked † in the procedure **SEARCH** is replaced by following piece of code:

```
(1) If E covers more than one corner of the MBR of R.DESC[i], then add R.DESC[i] to LIST;
(2) If E covers less than two corners of the MBR of R.DESC[i], then search the quadtree level: R.DESC[i].POINTER. If R.DESC[i].POINTER intersects E, then add R.DESC[i] to LIST;
(3) Otherwise do nothing.
```

If the low level structure need to be searched, the following operations are recommended to increase the efficiency:

```
(1) Compute: $\lambda = E^{*} @MBR(R.DESC[i].POINTER)$.
(2) First sort $\lambda$ in descending size order then in ascending key order.
(3) For each block $b \in \lambda$ do
        If b intersects the quadtree, $R.DESC[i].POINTER$, then return success.
```

The orthogonal range search in a quadtree involves two steps: decompose the query rectangle into maximal regular blocks and perform range search to each of these blocks [13,17]. Since the description of the minimal bounding rectangle of an object stored as an R/Q tree is explicitly in the leaf level of the R-tree, it can be used to eliminate any unnecessary search. On the other hand, (2) enables the search process to consider the blocks in decreasing size order with a high probability of being successful.

In this example, the search starts at the root R then all promising subtrees are explored one by one. In other words, the R-tree is explored in a depth first fashion. To illustrate how the quadtrees are explored, consider the process in which object D is obtained.

Suppose the subtree rooted at R1 has already been explored (thus, objects A, B, and E have been obtained). Since R2 is the root of a promising subtree, it is examined:

Since $I \cap MBR(D) \neq \Phi$, $R1.CLASS = leaf$ and I covers less than two corners of $MBR(D)$, where $D = R2.DESC[1].POINTER$, the following operations are performed:

(1) Compute $\lambda = I^{*} @ MBR(D)$: $\lambda = \{<12,1>, <36,1>\}$.

(2) Sort λ: $\lambda = \{<12,1>, <36,1>\}$.

<12,1> is chosen first. The succeeding search path in the quadtree is : ->D->D1. Since <12,1> is neither contained in D1.DESC[1] nor D1.DESC[2], D1.DESC[2].KEY <12, and D is a leaf node, D2 is explored through sequentail list. Again, because D2.DESC[1] is not contained in <12,1>, the intersection of <12,1> and object D is deemed to be empty, by **Lemma 2.4**. Therefore, the first trial fails. Then the process backtracks to the R-tree level where <36,1> is chosen. The succeeding search path in the quadtree is : ->D2. Since <36,1> is contained in D2.DESC[2] and D2 is a leaf node, D is a candidate.

The Object A,B,C, and E are obtained similarly except that the quadtree of object B is untouched. The algorithm terminates with the answer {A,B,C,D,E} and the number of disk accesses is 12.

Although, the strategy proposed for the second phase usually leads to an efficient solution, it is not always optimal. For instance, assume that the pointer of the sequentail list of an object is stored in the corresponding leaf node of the R-tree, e.g., R1.DESC[1] has two pointers: one points to node A and the other points to A11 (the head of the sequentail list of object A). Then the query can be answered in 8 disk accesses. This is achieved by sorting blocks in the query rectangle in Morton sequence and searching low level quadtrees along their sequential lists.

B/Q tree:

As previously described, the orthogonal range search in a quadtree is accomplished by searching the maximal blocks of the query rectangle. However, since the quadtree is constructed for the whole image rather than for each object as an R/Q tree, little knowledge is known prior to a query. Thus, to find all objects that intersect a given rectangular area, every block in the query rectangle necessitates a search in the tree. Obviously, a B^+/Q tree is awkward in dealing with this kind of query since both redundant (when search leads to a same object for more than once) and irrelevant (when block overlaps no object) searches may occur.

When searching for I^* in Morton sequence order, and using a sequentail list, the number of disk accesses is caculated to be 11. It is not very difficult to see that this is also the minmum cost of the approach.

In conclusion, although both an R/Q tree and a B^+/Q tree are able to deal with the second query type, an R/Q tree is believed to be superior. More importantly, an R/Q tree structure facilitates subsequent operations such as performing transformations, set operations, etc.