Discs Publication No. TRD5/91

Technical Report

Which is more Efficient for Window Search Bit Interleaving or Key Concatenation?

*Chuan-Heng Ang and Hanan Samet **

May 1991

DISCS
NUS

DEPARTMENT OF INFORMATION SYSTEMS
AND COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
KENT RIDGE, SINGAPORE 0511

**Computer Science Department and Institute for Advanced Computer Studies and Center for Automation Research, U. of Maryland, College Park, MD 20742, USA.*

FOREWORD

This technical report contains a research paper, development report or tutorial article which has been submitted for publication in a journal or for consideration by the commissioning organization. As a courtesy to the current or future owner of the copyright, it would be appreciated if copying of the report is kept to the essential minimum.

The report represents the ideas of its author, and should not be taken as the official views of the Department or the University. Any discussion of the content of the report, and requests for further copies, should be sent to the author, at the address shown on the cover.

The Department will be pleased to place academic and other research and development organizations on the mailing list for receiving future reports, on the understanding that each recipient will send us similar publications of its own. Please use the form printed on the last page for this purpose.

C K YUEN
Head of Department

WHICH IS MORE EFFICIENT FOR WINDOW SEARCH

BIT INTERLEAVING OR KEY CONCATENATION?

Chuan-Heng Ang

Department of Information Systems and Computer Science
National University of Singapore

Hanan Samet

Computer Science Department and
Institute of Advanced Computer Studies and
Center for Automation Research
University of Maryland
College Park, MD 20742

ABSTRACT

Points in multidimensional space can be well-ordered into points in one dimensional space by either applying bit interleaving transformation or using key concatenation. The algorithms to carry out a window search on the results of the transformations are described and their complexity is analyzed and compared. The condition under which the search operation is executed more efficiently on the space using key concatenation is found. Therefore, the myth that bit interleaving is superior than key concatenation is dispelled.

Keywords and phrases: analysis of algorithm, B-tree, N-tree, bit interleaving, key concatenation, window search, range query, quadtrees.

May 20, 1991

1. INTRODUCTION

When information is organized into records stored on disk, it is not uncommon to specify more than one key to be used in various key index accessing methods. Either several indexes are used with one index for each key, or all the keys are combined into a superkey and only one index is built. The former provide the flexibility in accessing records by using any one of the keys that have been built into indexes. It requires more space for the indexes and additional effort to maintain their consistency. The latter, on the other hand, is modest in its space requirement. It is a popular accessing method used in linear quadtree [Garg82, Abel83} which is the backbone of some Geographic Information Systems [Shaf90]. It is handy to use in accessing spatial data by specifying all component coordinates in a key to locate an object which may be a point or a rectangle. The formation of a superkey is the realization of a mapping from multidimensional space into one dimensional space, turning all grid points into a totally-ordered (well-ordered) set of points.

For ease of discussion, we will call the multidimensional space as k-space with k as its dimensionality, one dimensional space as 1-space, a transformation from k-space to 1-space as a coding scheme, and the values in 1-space as codes. We will also restrict our discussion to 2-space and leave its generalization to k-space to the readers.

Let (x,y) be the coordinates of a given point in 2-space with $0 \le x,y \le 2^n$ where 2^n is the size of the grid, or the coordinate system. Let $x = x_{n-1} \cdots x_1 x_0$ and $y = y_{n-1} \cdots y_1 y_0$ be the binary representation of x and y where x_i and y_i are either 0 or 1 with x_0 and y_0 being the least significant bit. Define the *concatenation* of x and y as $CK(x,y) = x_{n-1} \cdots x_1 x_0 y_{n-1} \cdots y_1 y_0$. $CK(x,y)$ is formed by appending the binary representation of y to that of x. Define the *bit interleaving* of x and y as $IK(X,Y) = x_{n-1} y_{n-1} \cdots x_1 y_1 x_0 y_0$. We use $CK()$ and $IK()$ to represent the two functions defined above. It is purely a matter of taste to assume that the value of x is more significant than that of y in the definitions. We see that $0 \le CK(x,y), IK(x,y) \le 2^{2n}$. Figure 1 shows how the grid points in 2-space are being transformed by these coding schemes into codes.

From Figure 1, it seems that $IK()$ always assigns the next code to the next grid point nearer to the origin. For example, $IK()$ assigns 0 and 1 to (0,0) and (0,1). Instead of assigning 2 and 3 to (0,2) and (0,3) as what $CK()$ does, $IK()$ assigns them to (1,0) and (1,1), the two points which are closer to the origin in terms of their Euclidean distances or their Chessboard distances [Same90a, Same90b] from the origin. As such, guided by our intuition, it is quite natural to assert that $IK()$ will be a better coding scheme for a more efficient window search operation, or a range query as it is commonly called.

In [Whit82], White remarked that "(using bit interleaving,) the path from one point to the next is usually quite short. Two nearby points in 2-space tend to be nearby ... Most points in a small neighborhood of a point in k-space will be in a small neighborhood of the same point in 1-space ... Thus ordinary B-tree searches and sequential retrieval would collect most neighboring points.". Motivated by such a staunch belief that bit interleaving is superior than key concatenation for window search, he proposes to use N-tree, a B-tree structure superimposed on the range of $IK()$. It is called N-tree since we will get letter N when we connect the points according to the first four code values, namely 0, 1, 2, and 3 in that order. In contrast, the shape of the four points connected according to the first four $CK()$ code values is a vertical line that looks like "1" and we shall call it 1-tree for ease of reference.

In this paper, we would like to dispel the myth that "bit interleaving is better than key concatenation". The statement is misleading when it is stated without a comparison between the time requirements involved. In section 2 and 3, we describe how to perform window search on N-trees as well as 1-trees. In section 4, we derive the complexity bounds of the two search algorithms and investigate the conditions under which the search on 1-trees will be more efficient. We make our conclusion in section 5.

2. Window Search On N-trees

N-tree is proposed by White [Whit82] to organize the range of function $IK()$. The codes and the pointers in a page of a B-tree are arranged as $s_0 code_1 s_1 code_2 \cdots s_{c-1} code_c s_c$, where c is the number of codes in the page, s_i is the pointer pointing to the left subtree of $code_{i+1}$ with pages containing codes which are between $code_i$ and $code_{i+1}$ for $1 < i < c$, s_0 points to the subtree with codes smaller than $code_1$, and s_c, greater than $code_c$. Each page covers an interval $[code_1, code_c]$ in 1-space.

Let (x_1,y_1) and (x_2,y_2) be the lower left and upper right corners of the given window w, i.e., $w=\{(x,y) \mid x_1 \le x < x_2, y_1 \le y \le y_2\}$. We denote w by $[(x_1,y_1),(x_2,y_2)]$. The window search operation is to find all the points of an N-tree that are within the window. Let $IK(w)=\{IK(x,y) \mid (x,y) \in w\}$. It is easy to see that the minimum and the maximum values of $IK(w)$ are $minw=IK(x_1,y_1)$ and $maxw=IK(x_2,y_2)$. These values are used to prune the tree quickly during stage 1. Discarding the remaining points that are not within the window will be done in stage 2.

Given a page $[p_1,p_2]$, if $p_2 < minw$ or $p_1 > maxw$, then $[p_1,p_2] \cap IK(w)=\emptyset$. This means that all the points with codes in this page as well as those in the subtrees rooted in this page are not covered by w and hence they can be discarded. If $[p_1,p_2] \cap IK(w) \ne \emptyset$, then it may happen that $IK^{-1}([p_1,p_2]) \cap w=\emptyset$. An example can be given easily from Figure 1 in which $w=[(2,1),(4,5)]$ and $(x,y)=(2,6)$. By applying $IK()$ to the points, we have $minw=9$, $maxw=49$, and $IK(2,6)=28$. Although $minw \le IK(2,6) \le maxw$, (2,6) is not in w.

By just looking at the codes in a page, we do not gain much information as to how close their corresponding points in 2-space are. We may have two points whose codes are different by only 1 and yet they are far apart in 2-space. For instance, $IK(3,7)=31$ and $IK(4,0)=32$ yet the difference in their y coordinate is 7. This big gap between points in 2-space that is well-hidden by the coding scheme poses some problems when we want to perform an intersection test between the interval and the window.

Given two objects in 2-space, a quick intersection test can always be carried out by checking whether the corresponding enclosing rectangles intersect each other. Under $IK^{-1}()$, an interval $[p_1,p_2]$ in 1-space actually describes a wiggly segment in 2-space. In the following discussion, we will call the inverse image of an interval in 1-space under $IK()$ as a segment. The existence of a big gap in a segment always causes an enclosing rectangle to be found which is far too large to be of any use in the quick test. White proposes to partition the interval (or rather the segment) along the gap g so that each of the resulting wiggly segments of $[p_1,g-1]$ and $[g,p_2]$ can be enclosed in much smaller rectangles before the quick test is carried out. These rectangles are chosen from those resulted from performing regular decomposition [Same90a] on 2-space. Partitioning of an interval can be repeated recursively until there are only a few points left in which case it is faster to test individual points for possible inclusion in the window.

For example, suppose a page P contains codes 28, 31, and 35. Since $minw \leq 28 \leq maxw$, $IK^{-1}(P)$ may intersect w. When we refer to Figure 1, we may be deceived into believing that $IK^{-1}(P) \cap w \neq \emptyset$ since the line segment joining the points [3,7] and [4,0] with codes 31 and 32 cuts across the window. If P is partitioned into [28,31] and [32,35] at the gap 32, we see that the corresponding small enclosing rectangles [(2,6),(3,7)] and [(4,0),(5,1)] do not intersect w.

The details of the action taken in a window search operation can be found in [Whit82, Trop81]. Although Tropf uses 1-2 brother tree [Ottm80] in his implementation, the way that the tree is being trimmed during the search is the same. Below is just a list of the essential steps required for each page accessed, with the first two steps being done in stage 1.

(1) Trim the root from the left and the right by discarding all the keys that have codes smaller than $minw$ or greater than $maxw$. Let the resulting interval be $[p_1,p_2]$.

(2) The left subtree of p_1 will be trimmed from the left, and the right subtree of p_2 will be trimmed from the right.

(3) When a subtree intersect $IK(w)$, push the information of current page onto a stack so that testing of the rest of the page can be resumed after the subtree has been processed.

(4) Check whether there is a big gap if necessary before the segment is tested for intersection with the window. Discard any portion with empty intersection.

(5) Test each point for possible inclusion and report it if it is included.

(6) When a page is completely processed, resume the testing of its parent page by popping the required information from the stack. The search is completed when the stack is empty.

3. Window Search On 1-trees

Given a window $w=[(x_1,y_1),(x_2,y_2)]$, we would like to find all the points of a given 1-tree that fall within w. The minimum and maximum of $CK(w)$ are $minw=CK(x_1,y_1)$ and $maxw=CK(x_2,y_2)$. They can be used to trim the interval covered by the codes in a page by discarding from the left those points in the page as well as in the subtrees with codes smaller than $minw$ and from the right those points in this page and the subtrees with codes greater than $maxw$. The first stage to trim the interval is thus the same as that used in window search on N-trees. The second stage will be quite different.

Let $[p_1,p_2]$ be the interval containing the codes of those points that remain after step 1. Although they can be tested for inclusion in a straight forward manner with each test involves $2k$ comparisons, it is desirable to replace them by a single comparison using the codes. Since $minw \leq p_1 \leq maxw$, we have $x_1 \leq x_{p_1} \leq x_2$. There are 3 cases to consider when y_{p_1} is compared with y_1 and y_2. These cases are also illustrated in Figure 2.

Case 1 : The point $CK^{-1}(p_1)$ lies below the window when $y_{p_1} < y_1$.

Construct the code $CK1=x_p y_1$. Discard all the codes p in this page as well as their left subtrees when $p < CK1$. The testing starts from p_1 onwards and terminates as soon as the first code q found to be greater than or equal to $CK1$. Proceed to the left subtree of q to discard all the codes that are less than $CK1$.

Case 2: The point is in the window when $y_1 \le y_{p_i} \le y_2$.

Report the point and construct the code $CK2 = x_p y_2$. Report all the points $CK^{-1}(p)$ such that $p \le CK2$ until the first code q greater than $CK2$ is found. Proceed to the left subtree of q to report those points with codes smaller than or equal to $CK2$.

Case 3: The point is above the window when $y_{p_i} > y_2$.

Construct the code $CK3 = (x_{p_i}+1)y_1$ and discard all codes less than $CK3$ until the first code $q > CK3$ is found. Discard all codes which are less than $CK3$ from the left subtree of q as well.

To understand how stages 1 and 2 are being carried out, let us look at one example. Using the same window $w=[(2,1),(4,5)]$, we have $minw=2\times8+1=17$ and $maxw=4\times8+5=37$. Suppose a page contains $s_0 5 s_1 10 s_2 20 s_3 30 s_4 40 s_5 50 s_6$. After stage 1, we discard 5, 10, 40, and 50 and also all the codes in the subtrees pointed to by s_0, s_1, s_5, and s_6. The subtree pointed to by s_2 has to be trimmed from the left and that pointed to by s_4 has to be trimmed from the right.

When $p_1=20$, $x_{p_1}=\lfloor 20/8 \rfloor=2$, $y_{p_1}=20 \bmod 8=4$. Therefore $CK^{-1}(p_1)=(2,4)$ is in the window. It will be reported after the subtree pointed to by s_2 has been processed in an inorder tree traversal. $CK2=2\times8+5=21$. $CK2$ is used to report those points with codes p in the subtree pointed to by s_3 such that $p \le CK2$. When the subtree pointed to by s_3 is completely processed, we proceed to the next code which is 30.

When $p_1=30$, $x_{p_1}=\lfloor 30/8 \rfloor=3$, $y_{p_1}=30 \bmod 8=6>y_2$. Therefore 30 is not in w. Since $y_{p_1}>y_2$, we need to calculate $CK3=4\times8+1=33$. We use 33 to prune and process the subtree pointed to by s_4 before we resume to process the value 40.

4. Complexity Bound of Window Search Operation

4.1. Window Search on N-trees

Although White has analyzed the complexity of window search on k-space represented by an N-tree, the analysis is incomplete and with some faults. Firstly, the time required to locate a point in N-tree should be $O(\log N)$ where N is the total number of points in the tree. It is not $O(k \log N)$ as stated in [Whit82] since all the comparisons are made using the codes, not each of the k coordinate components.

Secondly, for a small window with volume c, the complexity of the window search is stated to be $O(k \log N + F)$ where F depends on the number of answers retrieved. It seems that the volume does not play a part in the complexity of the search. Although it has been pointed out that c "influences the constant hidden by $O(k \log N + F)$", it is not clear whether the constant is an additive constant or a multiplicative one, and if it is multiplicative, whether it affects the term $k \log N$ or F.

Tropf carried out two experiments to find out the performance of the window search operation on the range of $IK()$ organized by a 1-2 brother tree [Trop81]. Using the results obtained from the first experiment, he plots d, the difference between the number of records inspected and the number of records found versus l, the edge length of a square window. He finds that the time complexity of the window search operation can be bounded by $O(k \log N + F)$. In the second experiment, he keeps N, the number of points in the tree, as constant and varies k, the dimensionality of the space (domain). To his surprise, the value d decreases as k increases, contradicting to the prediction made based on the complexity bound $O(k \log N + F)$.

The contradicting result obtained by Tropf is the consequence of a mistake that he made in the experiment. Although he maintains the number of points in the space as a constant while the dimensionality increases, he overlooks the impact of the ratio R_k between the volume of the window and that of the k-space on d. Assume that the grid size is g and k=2. We have R_2=l^2/g^2. When k increases to 3, Tropf simply adds one more dimension to 2-space with grid size g and generates the same number of random points from it. For 3-space, we have R_3=l^3/g^3 and R_3<R_2. Since R_k decreases as k increases, the number of random points falling within and around the window decreases. Since d is a measure of the effort required or wasted on inspecting those points which are near but not in the window, d also decreases. If R_k is maintained as a constant, then the effect of an increasing k should be felt and detected.

In the following, we would like to use a different approach to determine the time bound for the window search operation on N-tree. It is conceivable that the I/O time incurred by reading the pages from a B-tree will be the major contribution towards its time complexity. Since we are considering an upper bound of the complexity, we may assume that the whole N-tree is fully populated with all the possible values which can be generated by $IK()$. The more points are stored in the tree, the more time will be needed to search. To make thing worse, all the pages are just half filled as a result from the splits after page overflows. The idea is to make the search to access as many pages as possible so that more time will be needed.

Assume that the capacity of a page (or bucket size) is $2m$. We may further assume that $2m$=r^2 for simplicity. As the following derivation turns out, the assumption does not affect the end result which is expressed in big-Oh notation. Using these assumptions, a quadtree block area of size r^2 will have r^2 points with their corresponding codes stored in 2 pages. Assume that the four corners of the window are contained in four pages, every $r/2$ points along the boundary of the window are contained in a page, and every r^2 points in the interior of the window are contained in 2 pages. This is illustrated in Figure 3. Let L, W, and A denote the length, width, and area of w. Then the number of leaf pages needed is

$$NPN=4+\left\lceil \frac{L-2}{r/2} \right\rceil \times 2+\left\lceil \frac{W-2}{r/2} \right\rceil \times 2+\frac{(W-2)(L-2)}{r^2/2}=2A/r^2+(L+W-4)(1/r-4/r^2)$$

NPN is the maximum number of leaf pages required to cover all the points in a given window. The actual number needed for a given N-tree will be less than NPN. For a fixed page capacity, NPN=$O(A+P)$ where P is the perimeter of the window. When we include those internal pages accessed along the path from the root to these leaf pages, the number of pages accessed will be bounded by $O((A+P)\log N)$ since the height of a B-tree used to store N codes is $O(\log N)$. For each page accessed, there are at most $2k$ comparisons for each code. If the time required to read one page and the time to make one comparison are t_r and t_c respectively, then the total time required will be $O(A \log N(t_r+2k \times r^2/2 \times t_c))$. Since t_c is small, if both k and the page capacity are small, then t_r will dominate the term $k \times r^2 t_c$ and hence the complexity is $O((A+P)\log N t_r)$. In other words, the complexity totally depends on the area and the perimeter of the window for a given N-tree on a particular storage device with certain performance characteristics. When the areas of two windows are equal, the one with greater perimeter requires more time in its search. The time bound also reveal that more time will be required to search a bigger N-tree when a window of fixed size is specified. This is exactly what Tropf has stated "Experimental results show that for small hypercube ranges the average number of records to be inspected is logarithmic with the number of records." in [Trop81].

The above is only a worst case analysis. The actual cost of a window search is dependent on the placement of the window. The effect of the placement of the window has been studied by Dyer [Dyer82] although what he is interested is to find out the space requirement of placing a square in an image plane. As indicated in [Dyer82], the difference in the space requirements of the same square with different placements can be very significant, and so are the time required to access the resulting blocks covering the square.

4.2. Window Search On 1-trees

Using the same B-tree structure, the page capacity remains as r^2. We assume that each point on the horizontal boundary of the window is included in a separate leaf page. The rest of the points in the window will have their codes stored in a leaf page for every $r^2/2$ points. The illustration can be found in Figure 4. The number of leaf pages used is

$$NP1=2L+\frac{(W-2)\times L}{r^2/2}=2A/r^2+2L(1-2/r^2)=O(A+L).$$

By including those related internal pages in the counting, the total number of pages accessed is $O((A+L)\log N)$. It is easy to see that $NP1$ is also sensitive to the position of the window.

4.3. Comparison

Tropf stated in [Trop81] that 1-tree is not the right choice for window search using the reason that "although only a few or even no points may be in the search range, it is expected that many points (having x values within the specified range) must be inspected". He further illustrates his points using Figure 5 to show the long strip covering the window that must be searched. The description is not quite accurate because what will be inspected in the worst case will be the shaded strip excluding the column just below the lower left corner of the window and the column just above the upper right corner. The strip will cover a comparatively larger area only when the window specified is small or elongated in the x direction.

In comparison, the range of codes that must be inspected in an N-tree is enclosed within the two staircases shown in Figure 6. It is difficult to tell how much more effort is required to search through the range of codes depicted in Figure 5 than that in Figure 6 by visual inspection, let alone to assert confidently that window search on N-tree performs better than that on 1-tree. To be certain about the truth of the myth, the complexity bounds of the two search algorithms have to be derived and compared.

In terms of big-Oh notation, the complexity bounds of the window search on N-tree and 1-tree are of the same category. This is contrary to our belief that using bit interleaving in transforming multiple keys into a superkey will make a window search more efficient than that using key concatenation.

When the window specified is a square, i.e., $W=L$, we have

$$NPN=2L^2/r^2+(2L-4)(1/r-4/r^2), \text{ and}$$

$$NP1=2L^2/r^2+2L(1-2/r^2).$$

Their difference $NP1-NPN$ is about $2L$ for sufficiently large r, say $r>5$. Therefore, window search on N-trees is more efficient in general.

Further study show that it is possible for the window search on 1-tree to outperform that on N-tree. For this to happen, we must have $NP1\leq NPN$, or

$$2A/r^2+2L(1-2/r^2)\leq 2A/r^2+(L+W-4)(1/r-4/r^2) \text{ ,or}$$

$$L(2-1/r)\leq(W-4)(1/r-4/r^2)\leq(W-4)/r \text{, or}$$

$$L\leq\frac{W-4}{2r-1}.$$

That is, when the window is elongated in the y direction, it is more likely that the window search operation on 1-tree to be more efficient.

5. Conclusion

In this paper, we study the bit interleaving method and key concatenation method used in mapping the points from k-space to 1-space. The range of the mappings are being organized into B-trees and are termed N-tree and 1-tree respectively.

Intuitively, bit interleaving used in N-tree seems to map points which are close in k-space to points which are also close in 1-space. Hence it is believed that a window search operation on an N-tree should be more efficient than that carried out on a 1-tree. Our study shows that it is not true because Their performance is comparable. It is shown that using a square window in the search on N-trees is more efficient in general. When the search window is elongated in the y direction, the latter may outperform the former.

REFERENCES

1. [Abel83] - D.J. Abel and J.L. Smith, A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Computer Vision, Graphics, and Image Processing 24*, 1(October 1983), 1-13.

2. [Dyer82] - C.R. Dyer, The space efficiency of quadtrees, *Computer Graphics and Image Processing 19*, 4(August 1982), 335-348.

3. [Garg82] - I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

4. [Ottm80] - T. Ottmann and D. Wood, 1-2 Brother trees or AVL trees revisited, *The Computer Journal 23*, 3(1980), 248-255.

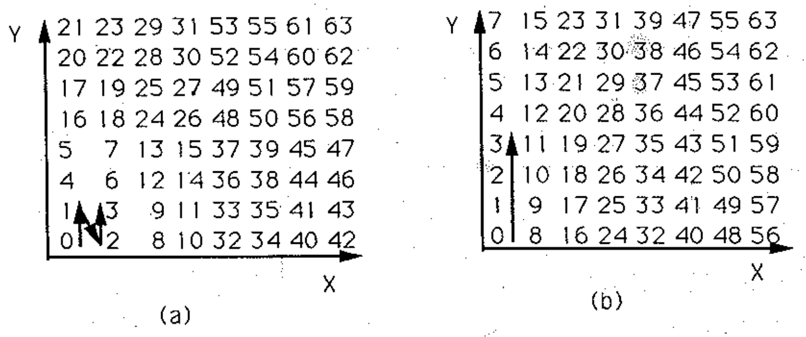
5. [Trop81] - H. Tropf and H. Herzog, Multidimensional range search in dynamically balanced trees, *Angewandte Informatik 23*, 2(February 1981), 71-77.

6. [Same90a] - H. Samet, *The Design and Analysis of Spatial Data Strucures*, Addison Wesley, Reading, Massachusetts, 1990.

7. [Same90b] - H. Samet, *Applications of Spatial Data Structures*, Addison Wesley, Reading, Massachusetts, 1990.

8. [Shaf90] - C.A. Shaffer, H. Samet, and R.C. Nelson, QUILT: A geographic information system based on quadtrees, *International Journal of Geographical Information Systems 4*, 2(April-June 1990), 103-131.

9. [Whit82] - M. White, N-trees: large ordered indexes for multi-dimensional space, US Bureau of the Census, Statistical Research Division, Washington, DC, 1982.



21	23	29	31	53	55	61	63
20	22	28	30	52	54	60	62
17	19	25	27	49	51	57	59
16	18	24	26	48	50	56	58
5	7	13	15	37	39	45	47
4	6	12	14	36	38	44	46
1	3	9	11	33	35	41	43
0	2	8	10	32	34	40	42

(a)

7	15	23	31	39	47	55	63
6	14	22	30	38	46	54	62
5	13	21	29	37	45	53	61
4	12	20	28	36	44	52	60
3	11	19	27	35	43	51	59
2	10	18	26	34	42	50	58
1	9	17	25	33	41	49	57
0	8	16	24	32	40	48	56

(b)

Figure 1. (a) Codes under bit interleaving IK()
(b) Codes using key concatenation CK().

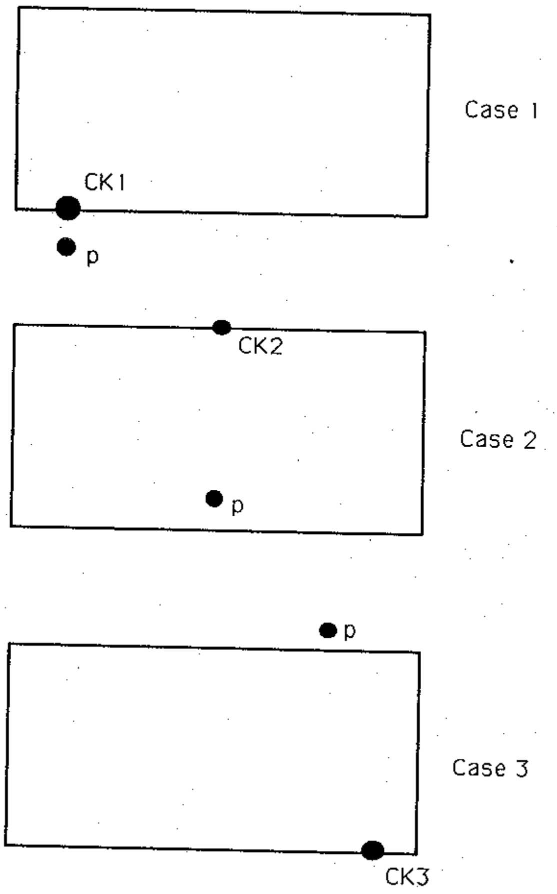


Figure 2. The relationship between p and CK1, CK2, and CK3 with respect to the window.

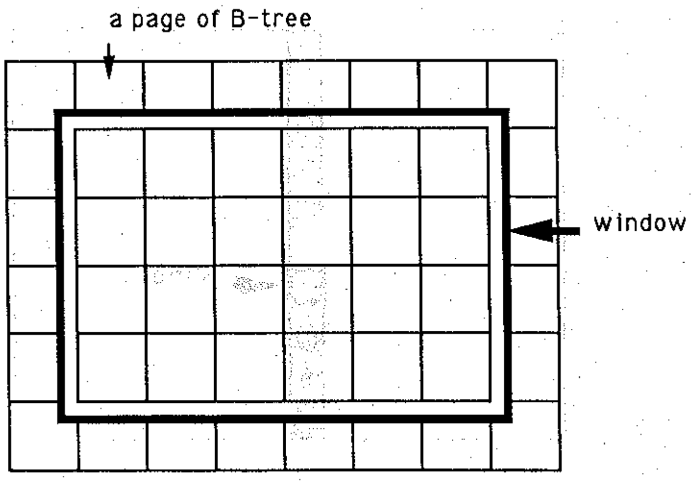


Figure 3. Leaf pages required to cover the window in N-tree.

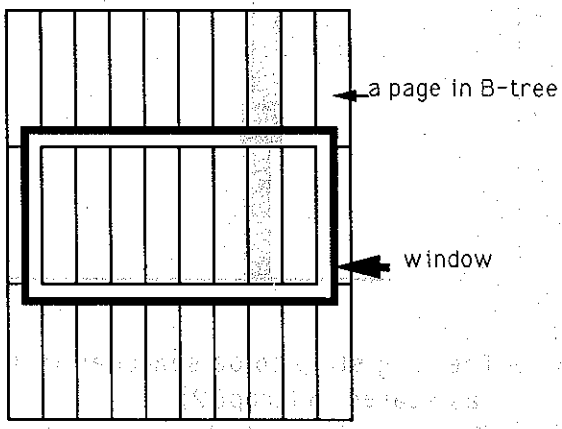


Figure 4. Leaf pages required to cover the window in 1-tree. .

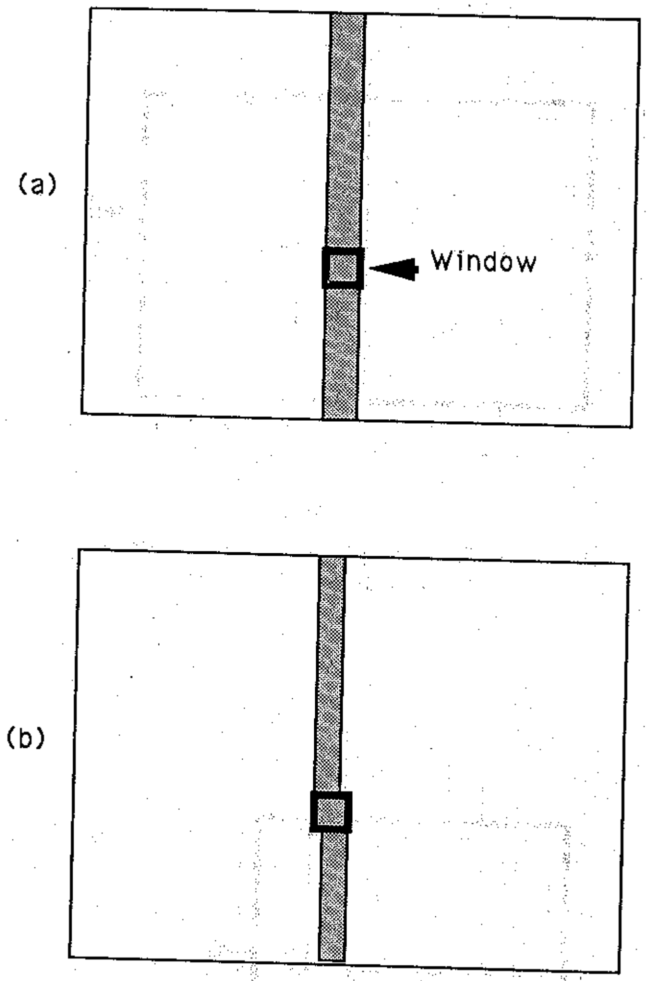
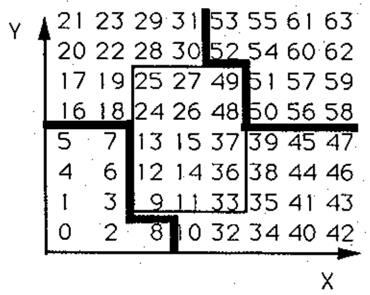


Figure 5. (a)The long strip to be searched in 1-tree as stated in [trop82].
(b)The actual area to be searched.



21	23	29	31	53	55	61	63
20	22	28	30	52	54	60	62
17	19	25	27	49	51	57	59
16	18	24	26	48	50	56	58
5	7	13	15	37	39	45	47
4	6	12	14	36	38	44	46
1	3	9	11	33	35	41	43
0	2	8	10	32	34	40	42

Figure 6. Region between the staircases are to be searched.