

4. Spatial Access Methods (SAMs)

(Spatial Indexing)

Overview

Learning Objectives

- Review one-dimensional access methods (B+-Trees)
- Understand the basic principles and ideas in SAM design (space versus data driven structures, cost factors and models, ...)
- Learn well-known SAMs, their pros & cons, and their cost models (Grid file, point quadtree, R-tree, R*-tree, region quadtree, k-d-b tree)

Literature

- [RSV02] Chapter 6 and/or [SC03] Chapter 4
- Volker Gaede, Oliver Günther: Multidimensional Access Methods. ACM Computing Surveys 30(2): 170-231 (1998).
<http://portal.acm.org/citation.cfm?id=280279>

Other papers of general interest:

- Lars Arge, *External Memory Data Structures*. In Handbook of Massive Data Sets, J. Abello, P.M. Pardalos, M.G.C. Resende (Eds.), Kluwer Academic Publishers, 2002, 313-357.
<http://www.daimi.au.dk/~large/Paperpages/dshandbook02.htm>
- E. Jeffrey Scott Vitter: *External memory algorithms and data structures: Dealing with MASSIVE Data*. ACM Computing Surveys 33(2): 209-271 (2001).
<http://portal.acm.org/citation.cfm?doid=384192.384193>

Review – Index Structures

Some notions and concepts to remember:

- Ordered Indexes
- Hash indexes
- Index entry
- Primary index, index-sequential file
- Dense and sparse indexes
- Multilevel indexes
- Secondary indexes
- Access, insertion, deletion time
- Space overhead

These notions are covered in any good textbook on database systems...

B⁺-Tree Index Files

B⁺-Tree indexes are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of the entire file is required.
- Advantage of B⁺-Tree index files: automatically reorganizes itself with small, local, changes, in the case of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B⁺-Trees: extra insertion and deletion overhead, space overhead.
- Advantages of B⁺-Trees outweigh disadvantages, and they are used extensively.

and.....

- a number of authors have proposed to map spatial index structures (primarily space-driven indexes) onto B⁺-Trees.

B+-Tree Index Files (2)

A B+-Tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length.
- Each node that is not a root has between $d \leq m \leq 2d$ entries. d is called the *order* of the tree, and it measures the capacity of a tree node.

Special case:

- The root can have between $1 \leq m \leq 2d$ entries.

Typical node with m search key entries, $d \leq m \leq 2d$:



- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_m$$

Non-Leaf and Leaf Nodes in B+-Trees

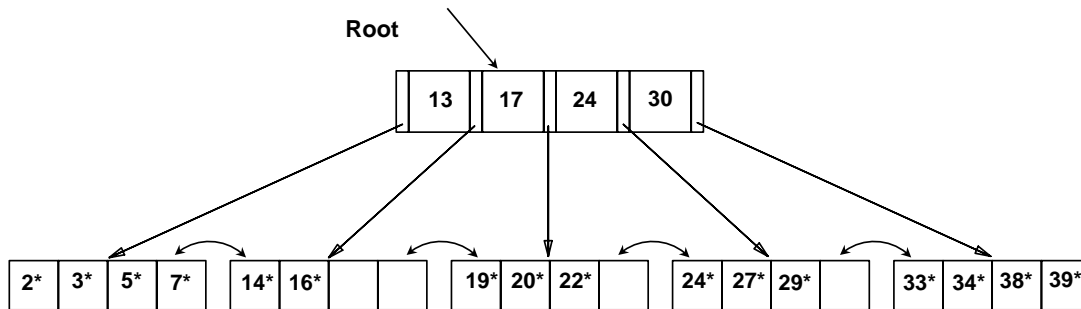
Non-leaf nodes form a *multi-level sparse index* on the leaf nodes. For a non-leaf node with m key value entries and $m+1$ pointers:

- All the search-keys in the subtree to which P_0 points are less than K_1 , and P_m points to a subtree in which all key values are greater than or equal to K_m .
- Pointer P_i points to a subtree in which all key values K are such that $K_i \leq K < K_{i+1}$.

Properties of leaf nodes

- Pointer P_i either points to a file record with search-key value K_i , or to a bucket of pointers to file records, each record having search-key value K_i . (there are some alternatives worth discussing.... ☞)
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than L_j 's search-key values.
- P_0 and P_m point to previous/next leaf node in search-key order.

Example of a B⁺-Tree



Note that each non-leaf and leaf node typically corresponds to one block (or page) on disk.

Observations about B⁺-Trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The B⁺-Tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (coming next).

Also

- Typical order: 100; typical fill-factor: 67%; average fan-out = 133
- Top level nodes can often be held in buffer:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = $133^2 = 17,689$ pages = 133 MBytes
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records

Queries on a B⁺-Tree

Find record(s) with a search-key value of k .

1. Start with the root node
 1. Examine the node for the smallest search-key value $> k$.
 2. If such a value exists, assume it is K_i . Then follow P_{i-1} to the child node
 3. Otherwise $k \geq K_m$, where there are m pointers in the node. Then follow P_m to the child node.
2. If the node reached by following the pointer above is not a leaf node, repeat step 1 on the node
3. Else we have reached a leaf node.
 1. If for some i , key $K_i = k$ follow pointer P_i to the desired record or bucket.
 2. Else no record with search-key value k exists.

If there are K search-key values in the file, the path is no longer than $\lceil \log_d(K) \rceil$. For example, with 1 million search key values and order $d = 100$, $\log_{100}(1,000,000) = 3$ nodes are accessed in a lookup. Compare this to a binary search tree!

Insertions into B⁺-Tree

Given a record with key value v

- Find correct leaf L for v .
- Put data entry onto L .
 - If L has enough space, *done!*
 - Else, *split* L (into L and a new node $L2$)
 - Redistribute entries evenly, *copy up* middle key.
 - Insert index entry pointing to $L2$ into parent of L .

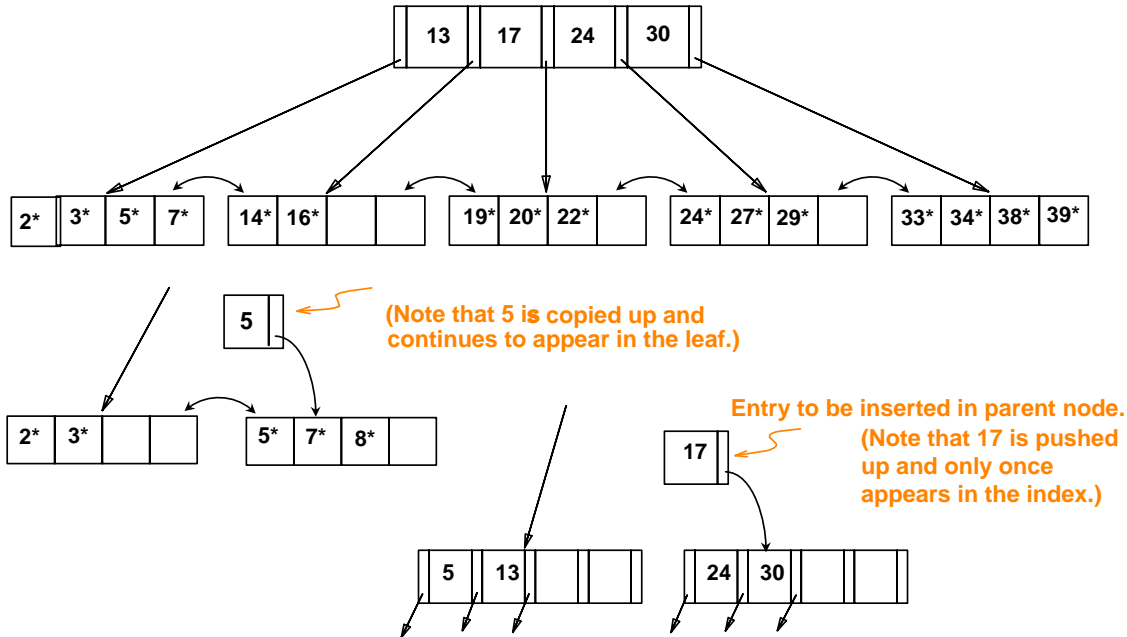
This split can happen recursively; to split index node, redistribute entries evenly, and *push up* middle key.

Splits “grow” the tree in width. A root split increases height of tree.

Tree growth: gets wider or one level taller at top.

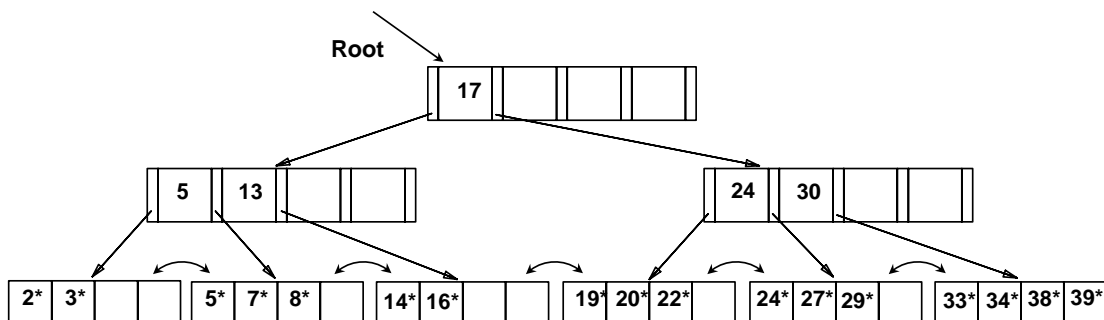
Insertions into B+-Tree (2)

Example: Insert record with key value "8" into tree on slide 7.



Insertions into B+-Tree (3)

Result tree after insertion of entry with key value 8



Note that the root was split, leading to an increase in the tree's height.

In this example, one could avoid split by re-distributing entries; however, this is usually not done in practice.

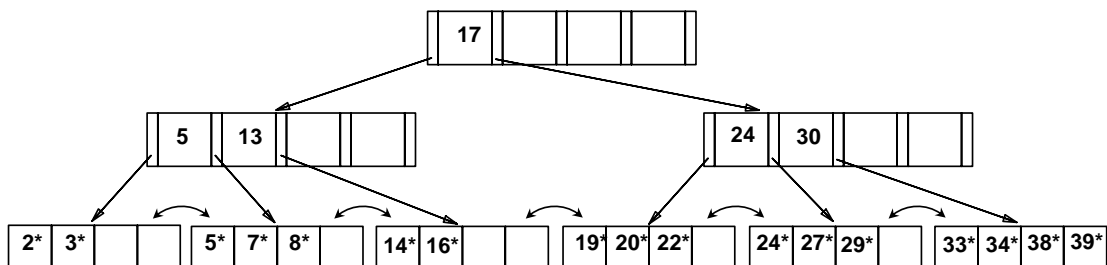
Deletions from B+-Tree

Given a B+-Tree and key value v :

- Start at root (search), find leaf L where entry v belongs.
- Remove the entry.
- If L is at least half-full, *done!*
- If L has only $d-1$ entries,
 - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
 - If re-distribution fails, merge L and sibling.
- If merge occurred, then one must delete the entry (pointing to L or sibling) from parent of L .
- Merge could propagate to root, decreasing the height of the tree.

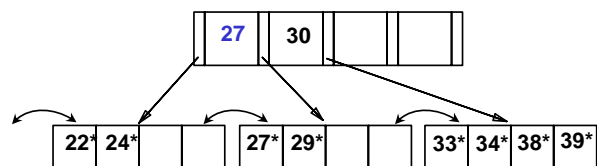
Deletions from B+-Tree (2)

Example: first, delete key value 19, and then key value 20



Deleting 19 is easy...

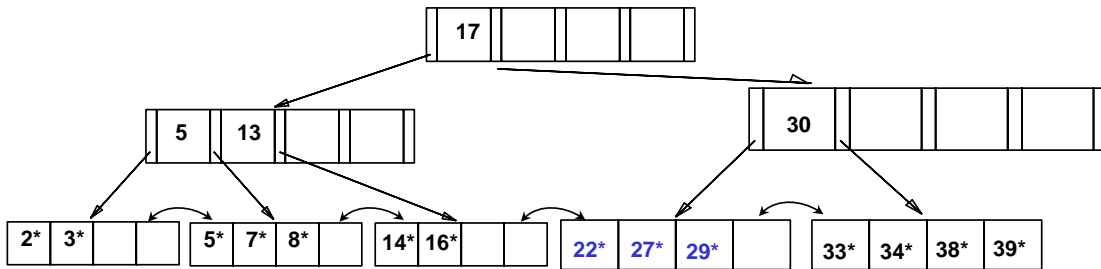
For the deletion of 20,
one can borrow from
sibling....and copy up
middle key.



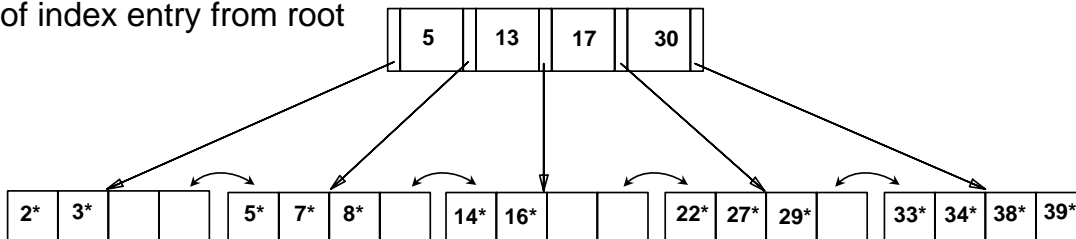
...and now delete value 24!

Deletions from B⁺-Tree (3)

....done via merging



...and then “pull-down”
of index entry from root



Comments on B-Tree Index Files

Insertion/deletion can be done with $O(\log_F(N/B))$ block accesses, with F being fan-out factor, N number of key value entries, and B number of entries per page. For search, we have to add factor a/B for reporting result.

Wait!! What about B-Trees....

- Advantages of B-Tree indexes:
 - May use less tree nodes than a corresponding B⁺-Tree.
 - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indexes:
 - Only small fraction of all search-key values are found early
 - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B⁺-Tree
 - Insertion and deletion more complicated than in B⁺-Trees
 - Implementation is harder than B⁺-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

Spatial Access Methods (SAMs)

Overview

- Point and window queries look for objects whose geometry contains a point or overlaps a rectangle.
- *Spatial joins*: given two sets of geometric objects, report pairs that satisfy some spatial relationships (intersection, containment etc.)
- With SAMs, one expects a search time for an object in a set S logarithmic in the size of S .
- In the following, the focus will be on 2-dimensional space.
- Often, instead of indexing the geometric objects themselves, a corresponding *minimum bounding box (mbb)* as geometric key for constructing spatial indexes is used.
- A spatial index is built on a collection of $[mbb, oid]$ entries.
- Operations on collection of objects indexed on their mbb are done in two steps: (1) filtering: select objects whose mbb satisfies predicate; these build a superset of the solution; (2) refinement: spatial test is done on actual objects (as has been discussed in Chapter 3).

Spatial Access Methods (2)

....overview (continued)

- B/B+-Trees rely on the order of the key domain, which is 1-dimensional and efficiently supports interval queries.
- A convenient order for geometric objects in 2D would be one that preserves object proximity. There is no such an order....(in fact, the same is true if one wants to use hashing...)
- In the following, for the different SAMs, we will focus on:
 - (1) *Index construction*, i.e., the insertion of a single object or collection of objects (bulk loading).
 - (2) *Search operations*, i.e., point and window queries where the search space is assumed to be a rectangular subset of the 2D plane with sides parallel to the x and y axes.

Issues in SAM Design

Expectations of a SAM:

- *Time complexity*: it should support exact (point) and range (window) search in sublinear time.
- *Space complexity*: its size should be comparable to that of the indexed collection.
- *Dynamicity*: During insertion/deletions, it must adapt to any growth or shrink of the indexed collection (or to a non-uniform distribution of objects) without performance loss. But, it is difficult to get a structure robust enough to support any statistical distribution of object location, size, shape etc.

Space-driven SAMs: Based on the partitioning of the embedding 2D space into rectangular cells, independent of distribution of the objects (e.g., Grid File, Region Quadtree)

Data-driven SAMs: Structures are organized by partitioning the set of objects; adapts to the objects' distribution in space (e.g., R-Tree, k-d-b-Tree)

The Grid File

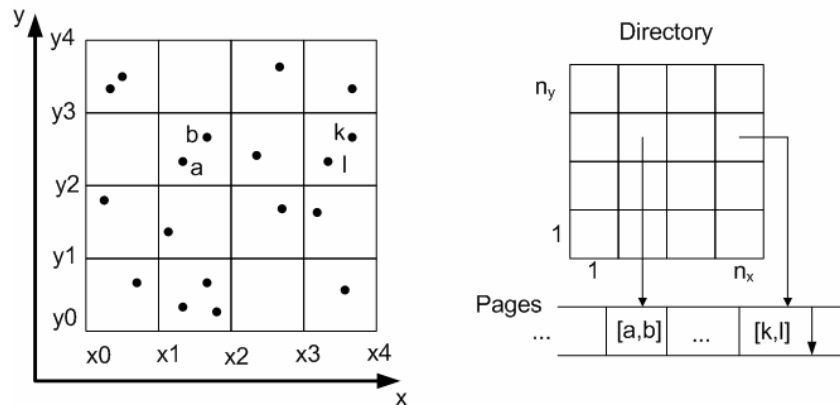
The Grid file is based on a space partitioning scheme and provides an improvement over the *fixed grid* initially designed for indexing points.

- Jon Louis Bentley, Jerome H. Friedman: Data Structures for Range Searching. ACM Computing Surveys 11(4): 397-409 (1979).
- Jürg Nievergelt, Hans Hinterberger, Kenneth C. Sevcik: The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transactions on Database Systems 9(1): 38-71 (1984).
- Michael Freeston: The BANG File: A New Kind of Grid File. SIGMOD Conference 1987: 260-269. (Balanced and Nested Grid)

In the following, we first look at the *fixed grid* structure for point indexing, followed by the *Grid file* for point indexing, and then the adoption of the Grid file for indexing mbbs (rectangles).

The Fixed Grid

- Search space is decomposed into regular grid of an $n_x \times n_y$ array of equal-size cells, each cell corresponding to one disk page.
- Point P is assigned to a cell c if c .rectangle contains P ; points associated with a cell c are stored sequentially in a page.
- The index requires a 2D array $[1:n_x, 1:n_y]$ as *directory*. Element $DIR[i, j]$ contains address PageID of the page that stores the points associated with cell $c_{i,j}$.



The Fixed Grid (2)

If $[S_x, S_y]$ is the size of the 2D search space, each cell's rectangle has the size $[S_x/n_x, S_y/n_y]$.

Operations:

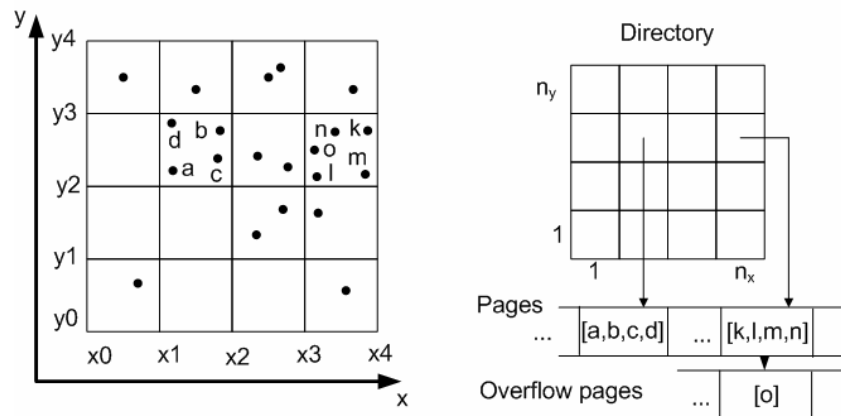
- Inserting $P(a,b)$:
 - compute $i = (a-x_0)/(S_x/n_x) + 1$ and $j = (b-y_0)/(S_y/n_y) + 1$
 - read page $DIR[i,j].PageID$ and insert P
- Point query: given a point $P(a,b)$, get the page as for insertion, read the page, scan entries, and check whether a point is equal to P .
- Window query: compute the set S of cells c such that c .rectangle overlaps with the query window W ; for each cell $c_{i,j}$ in S , read page $DIR[i,j].PageID$ and return the points in the page that are contained in the argument window W .

Point queries require single I/O (assuming directory is in main memory)

Number of I/Os for window query depends on number of cells intersection with window W .

The Fixed Grid (3)

- Grid resolution depends on the number N of points to be indexed; given a cell capacity of M , one can create a fixed grid with at least N/M cells.
- Points that do not fit into cell anymore are placed into *overflow pages*; thus, point queries might take more than just one I/O.
- Fixed Grid is very sensitive to point distribution; (very) skewed point distribution can result in linear search time.



Point Indexing with Grid File

Grid file approach is almost the same as for fixed grid, except that in case of a cell overflow, the cell is split into two cells

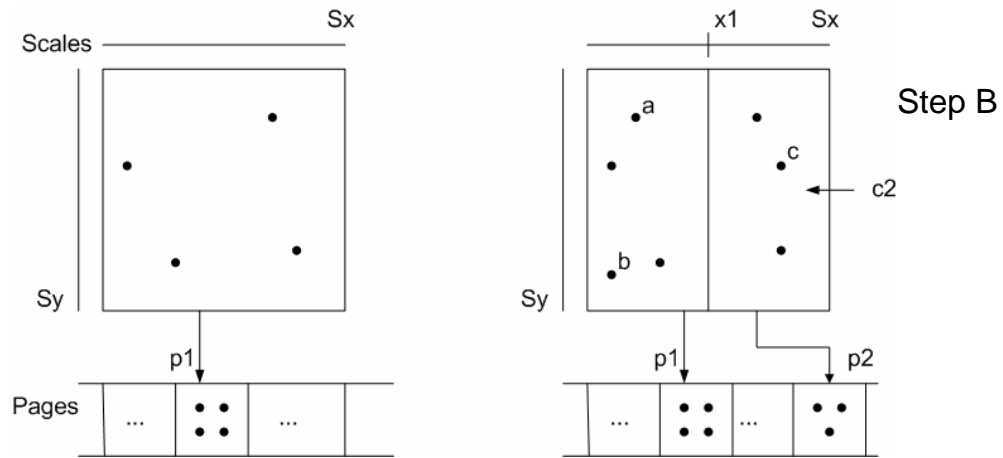
⇒ *Cells can be of different size and partition adapts to point distribution.*

Three data structures are necessary for the Grid File:

1. *Directory* is a 2D array that references pages associated with cells. Difference to the directory for a fixed grid is that *two adjacent cells can reference the same page*.
2. *Two scales* S_x and S_y are linear arrays describing the partition of a coordinate axis into intervals. Each value in one of the scales (say, S_x) represents a boundary in the partition of the search space along the related dimension (here, the x axis).

Point Indexing with Grid File (2)

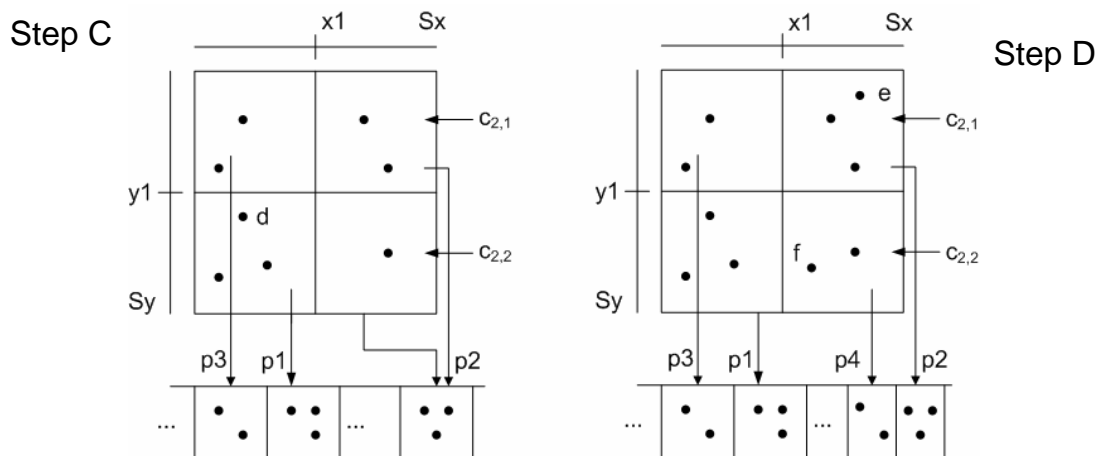
Example: assume capacity of a page is 4, directory is a single cell with associated page p1. In Step B, points a, b, and c are inserted.



Point Indexing with Grid File (3)

Example (cont.): In Step C, point d is inserted;

In Step D, points e and f are inserted.



Point Indexing with Grid File (4)

In summary, when a point P is inserted, the following cases can occur:

– *No cell split:*

P is located in a cell that has not reached maximum capacity.

– *Cell split and no directory splits:*

P falls into cell c , and page p associated with c is full but referenced by at least two cells. A new page p' is allocated and assigned to c , and objects in p contained in c .rectangle as well as point P are moved to p' .

– *Cell split and directory split:*

Page p referenced by cell $c_{i,j}$ into which P falls is full, and there are no other cells referencing p .



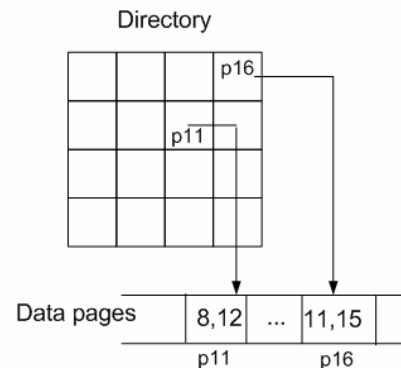
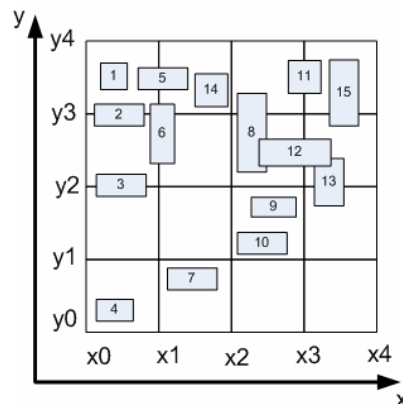
Pros & cons of the Grid file?

Rectangle Indexing with Grids

Assign rectangle (mbb) to the cells that overlap the rectangle.

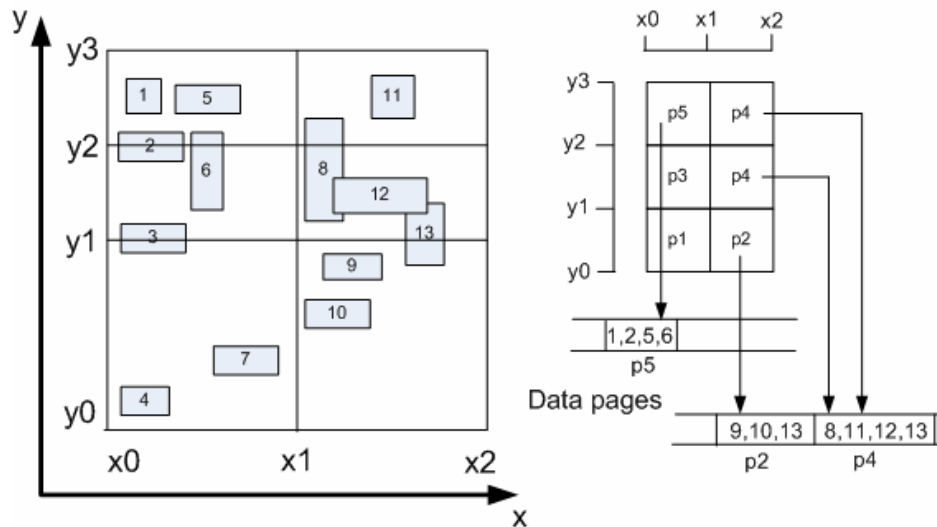
A *fixed grid*
for rectangle
indexing.

Page capacity
is 4.



Rectangle Indexing with Grids (2)

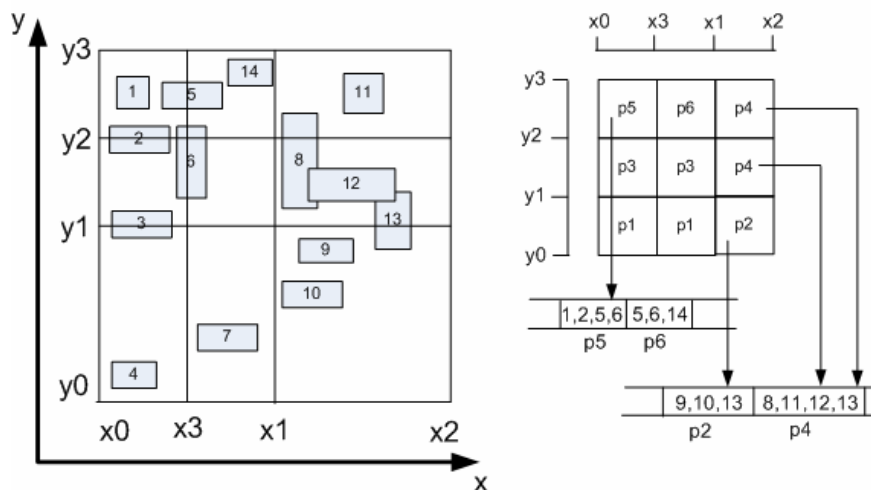
Grid file for rectangle indexing



Rectangle Indexing with Grids (3)

As consequence of object duplication in neighboring cells, a cell split is likely to occur more often.

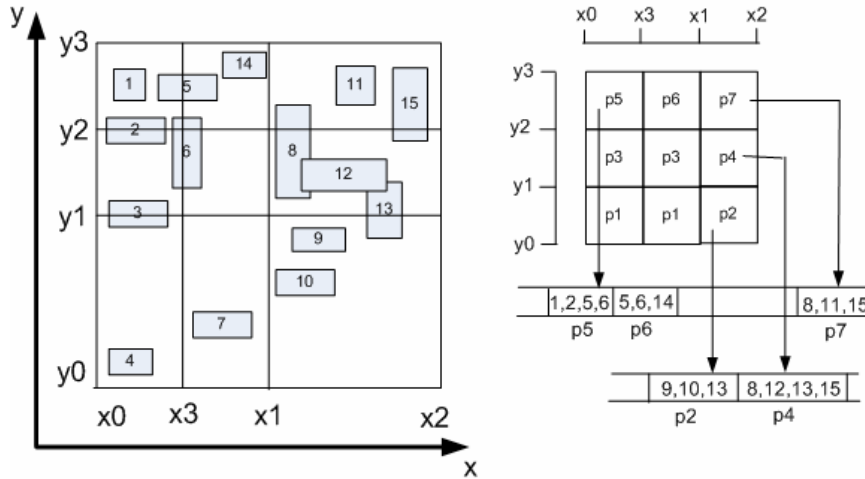
Assume set of mbbs from previous slide, insertion of mbb 14 into DIR[1,3] triggers overflow of page p5; a split routine is initiated ☞



Rectangle Indexing with Grids (4)

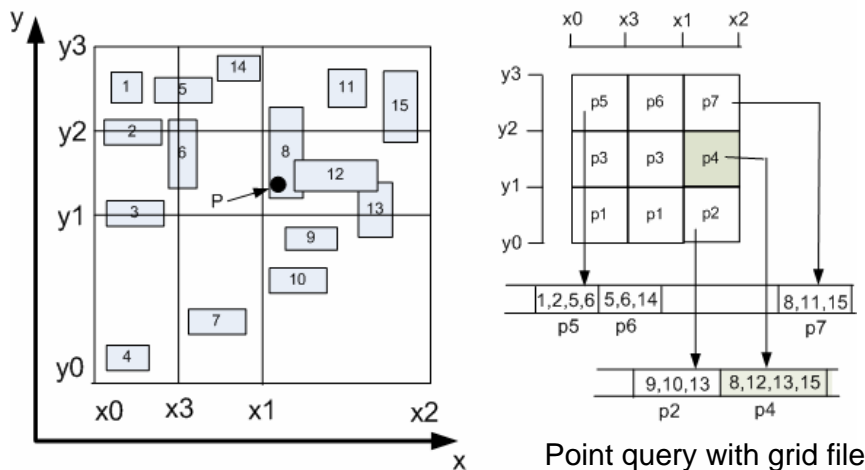
Assume set of mbbs from previous slide. A cell split without directory split is initiated when mbb 15 is inserted.

Page p4 overflows, page p7 is allocated, and the cells DIR[3,2] and DIR[3,3] are updated in the directory.



Point and Window Queries

Given a point $P(a,b)$, first determine single cell containing P . Access corresponding page and obtain collection E of entries for which $P \in e.mbb$ (results in set of $[e.mbb, e.oid]$ pairs). Then, fetch resulting objects based on $e.oid$ and test whether e contains P (refinement step).



Point and Window Queries (2)

The window query algorithm works as follows:

1. compute all cells that overlap the window W
2. each of the cells is scanned (as in point query algorithm)
3. because result can have duplicates, these need to be removed

Notes:

- Removing duplicates can be costly and space consuming. Its time complexity is superlinear in the size of the result.
- Loading several pages can be accelerated if they are consecutive on disk (sequential scan of n pages is far more efficient than random access to n pages).
- Point query can be carried out in constant I/O time (two disk access principle).
- Number of I/Os for window query is proportional to the window area.

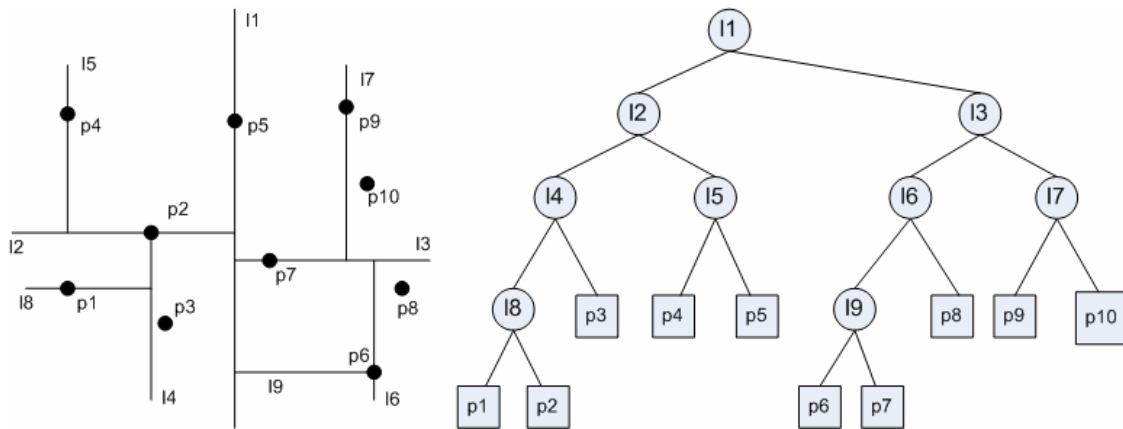
K-d Tree

- Prominent d -dimensional data structure that represents a recursive subdivision of the space into subspaces by means of $d-1$ dimensional hyperplanes; “k” stands for the dimension of the space.
- K-d tree is a binary search tree.
- Hyperplanes are iso-oriented, and their direction alternates among the d possibilities. For $d = 3$, splitting hyperplanes are alternately perpendicular to the x -, y -, and z - axes.
- Each splitting plane has to contain at least one data point, which is used for its representation in the tree.
- Insertions and search are simple, but deletions are not.
- There are many extensions to and variations of the k-d Tree:
adaptive k-d Tree, hB-Tree, Quadtree, k-d-B Tree

Jon Louis Bentley: Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM 18(9): 509-517 (1975)
[BK00] Section 5.2

K-d Tree (2)

Example of a 2D tree that indexes 10 points.



A k-d tree is constructed using a recursive procedure with two parameters:

- a set of points to be indexed
- depth of the root the recursive call constructs (=0 for the first call)

K-d Tree (3)

Algorithm BuildKDTree(P , depth): root of a k-d tree storing P
if P contains only one point **then** return a leaf storing that point
else if depth is even

then split P into two subsets with a vertical line ℓ through the median
x-coordinate of the points in P ;

Let P_1 points on the left of or on ℓ , P_2 points on the right of ℓ

else split P into two subsets with a horizontal line ℓ through the
median y-coordinate of the points in P ;

Let P_1 points below or on ℓ , P_2 points above ℓ

$v_{\text{left}} \leftarrow \text{BuildKDTree}(P_1, \text{depth}+1)$

$v_{\text{right}} \leftarrow \text{BuildKDTree}(P_2, \text{depth}+1)$

create a node v storing ℓ , make v_{left} left child of v , v_{right} right child of v

return v

(convention: point on splitting line ℓ belongs to subset left/below ℓ)

K-d Tree (4)

Construction time:

- Most expensive step at every recursive call is finding the splitting line; this requires finding the median x- or y-coordinate.
- Median line finding can be done in linear time.
- There is a better approach.... ☞

The building time $T(n)$ for a set of n points satisfies the recurrence

$$T(n) = \begin{cases} O(1), & \text{if } n = 1 \\ O(n) + 2T(\lceil n/2 \rceil), & \text{if } n > 1 \end{cases}$$

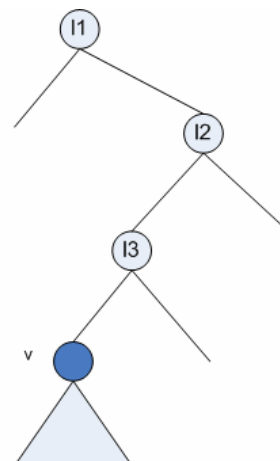
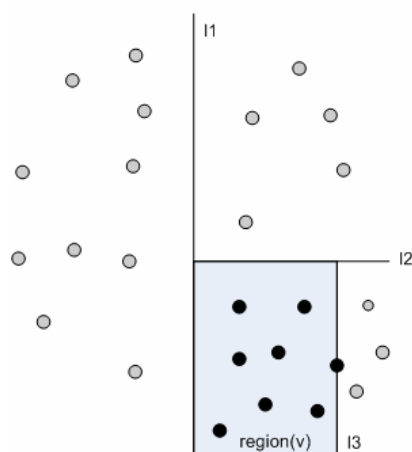
Each leaf on the tree stores one distinct point of $P \Rightarrow O(n)$ storage

Lemma: A k-d tree for a set of n points uses $O(n)$ space and can be constructed in $O(n \log n)$ time.

Querying a k-d Tree

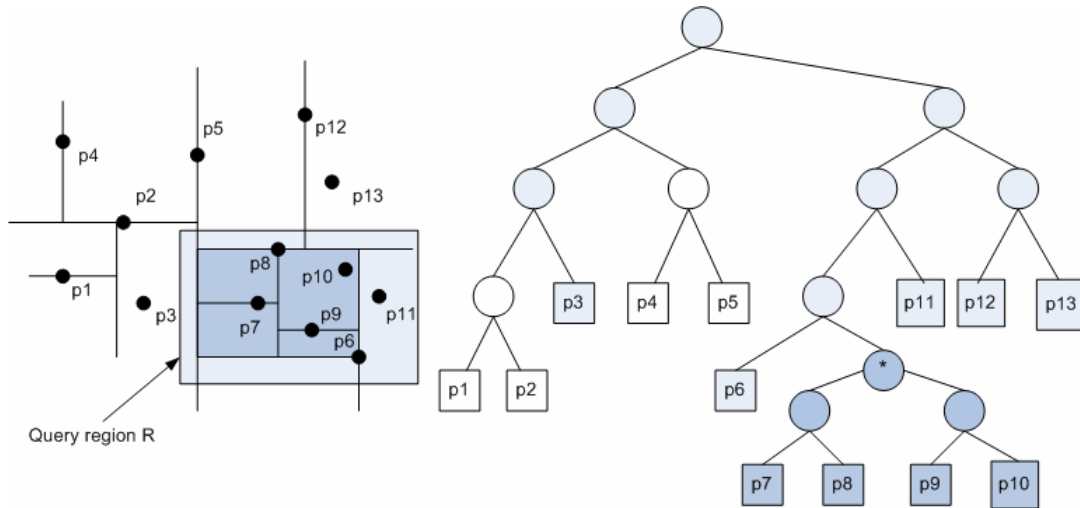
Observations:

- a point is stored in a subtree rooted at node v iff it lies in $\text{region}(v)$.
- for a given rectangle R (query region), one only has to search subtree rooted at v if R intersects $\text{region}(v)$.



Querying a k-d Tree (2)

- Traverse the k-d tree, but only visit the nodes whose region is intersected by the query region R.
- When a region is fully contained by R, all points can be reported.
- When a leaf is reached, check containment of point in R individually.



Querying a k-d Tree (3)

Algorithm SearchKDTree(v , R): all points below v that lie in R

if v is a leaf

then report the point stored at v if it lies in R

else if region($lc(v)$) is fully contained in R

then report SubTree($lc(v)$)

else if region($lc(v)$) intersects R

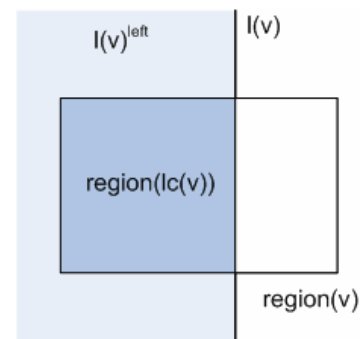
then SearchKDTree($lc(v)$, R)

if region($rc(v)$) is fully contained in R

then report SubTree($rc(v)$)

else if region($rc(v)$) intersects R

then SearchKDTree($rc(v)$, R)



The subroutine SubTree(v) traverses a subtree rooted at node v and reports all its points stored at the leaves.

Main test: does R intersect region corresponding to some node v ?

Querying a k-d Tree (4)

Theorem: A query with an axis-parallel rectangle in a k-d tree storing n points can be performed in $O(\sqrt{n} + k)$ time, where k is the number of reported points. ☞

Concluding remarks:

- Insertions ?
- k-d trees can also be constructed for point sets in d-dimensional space, where $d > 2$:
 - at the root, the set of points is split into two subsets by a hyperplane perpendicular to the x_1 -axis (root has depth 0)
 - at the children of that node, the partition is based on the x_2 -axis (depth 1)
 - at nodes at depth two on the x_3 -axis and so on, until at depth $d-1$, partitioning is based on the last axis.
 - at depth d , we start all over again...
- Construction time for $d > 2$ again $O(n \log n)$
- Query time is bounded by $O(n^{1-1/d} + k)$

Adaptive k-d Tree

One disadvantage of the k-d tree is that it is sensitive to the order in which points are inserted.

The adaptive k-d tree tries to solve this problem by choosing a split such that the same number of points can be found on both sides. Split lines are still parallel to x- and y-axes, but

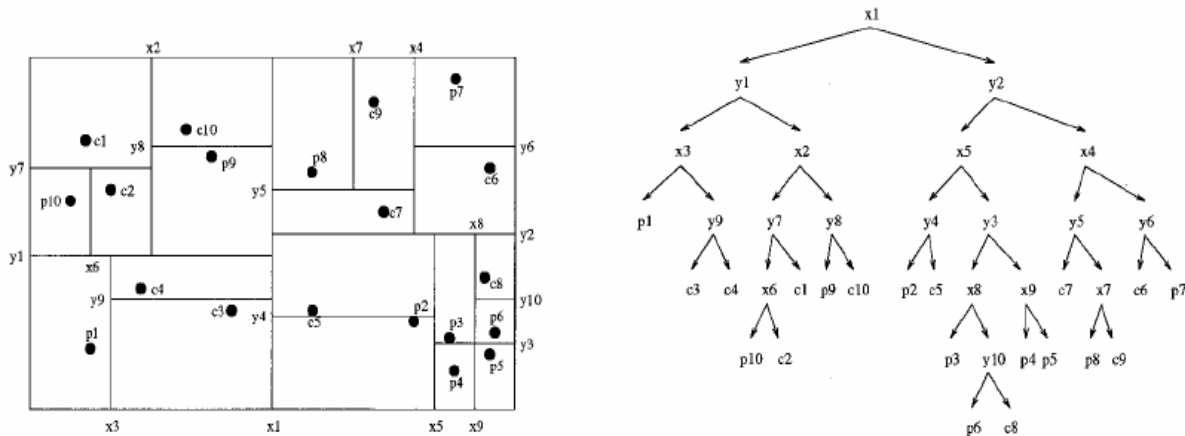
- they do not need to contain data point, and
- their directions need not be strictly alternating

The adaptive k-d tree is of a rather static nature and works best if all points are known in advance. It is obviously difficult to keep the tree “balanced” in case of many insertions/deletions.

A problem common to all k-d trees is that for certain point distributions, no hyperplane can be found that splits the data points evenly.

Adaptive k-d Tree (2)

Example of an adaptive k-d Tree [Gaede & Günther 98]



K-d-B Tree

Both the k-d Tree and adaptive k-d Tree are designed as main memory structures. They do not account for I/Os and are thus not appropriate for large spatial data sets.

The k-d-B Tree combines some of the properties of the adaptive k-d tree and the B-tree to handle points in multidimensional space.

It partitions the space in the manner of an adaptive k-d tree and associates resulting subspace with tree nodes, each interior node corresponding to an interval shaped region.

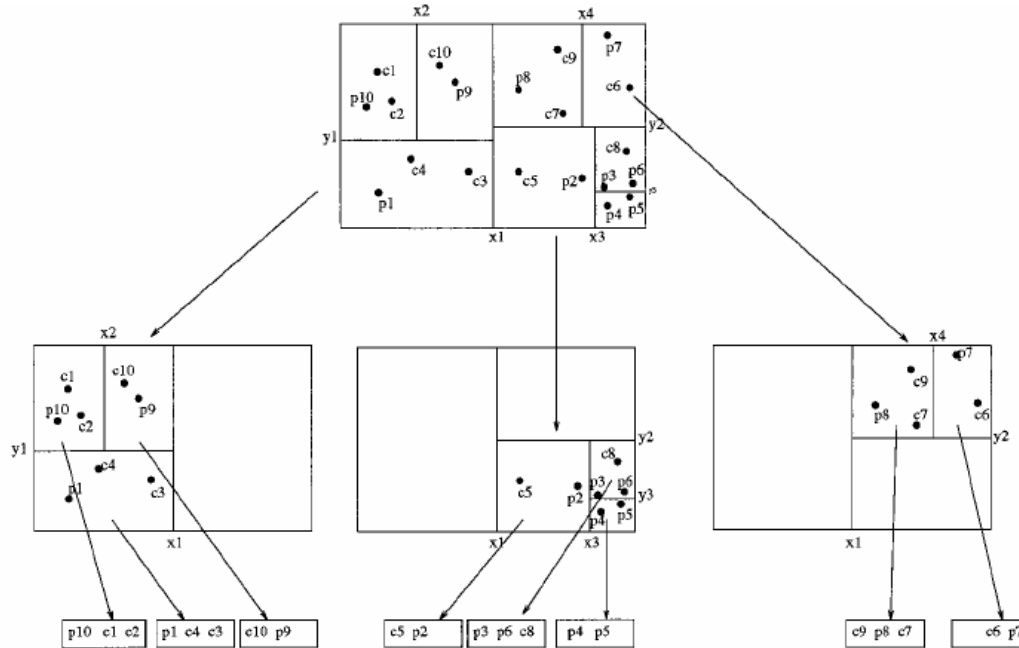
All nodes of the tree correspond to disk pages:

- A leaf node stores the data points that are located in the respective partition of the leaf.

Like the B-tree, the k-d-B tree is perfectly balanced. However, it cannot ensure minimum space utilization for nodes.

K-d-B Tree (2)

Example of a k-d-B Tree [Gaede & Günther 98]



K-d-B Tree (3)

k-d-B tree can be constructed in $O(n/B \log_B n)$ I/Os

Queries:

- Analogous to k-d tree algorithm, except that each leaf now contains B points
- $O(\sqrt{n/B} + k/B)$ I/Os

Insertions:

- First, perform point search to locate matching partition
- If not full, insert
- Otherwise split and move half of the entries to a new node
- Split can propagate up to root
- $O(\log_B^2 n)$ I/Os

Deletions are straightforward but may require merging of sibling nodes.

John T. Robinson: The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. SIGMOD Conference 1981: 10-18