

Data Mining: Foundation, Techniques and Applications

Lesson 3: Indexing



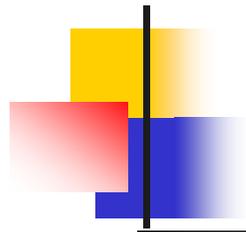
School of Computing

Anthony Tung(鄧錦浩)
School of Computing
National University of Singapore



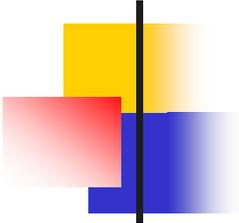
中國人民大學
RENMIN UNIVERSITY OF CHINA

Li Cuiping(李翠平)
School of Information
Renmin University of China



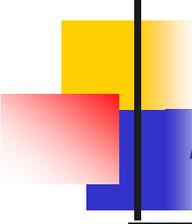
Examples

	Apriori	K-means	ID3
<i>task</i>	rule pattern discovery	<i>clustering</i>	<i>classification</i>
<i>structure of the model or pattern</i>	association rules	<i>clusters</i>	<i>decision tree</i>
<i>score function</i>	<i>support, confidence</i>	<i>square error</i>	<i>accuracy, information gain</i>
<i>search / optimization method</i>	<i>breadth first with pruning</i>	<i>gradient descent</i>	<i>greedy</i>
<i>data management technique</i>	<i>Our focus in the next two lectures</i>		



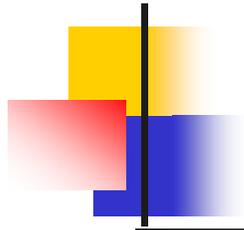
Overview

- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- String Indexing
 - Inverted Files
 - Suffix Tree/Array
 - Signature Files



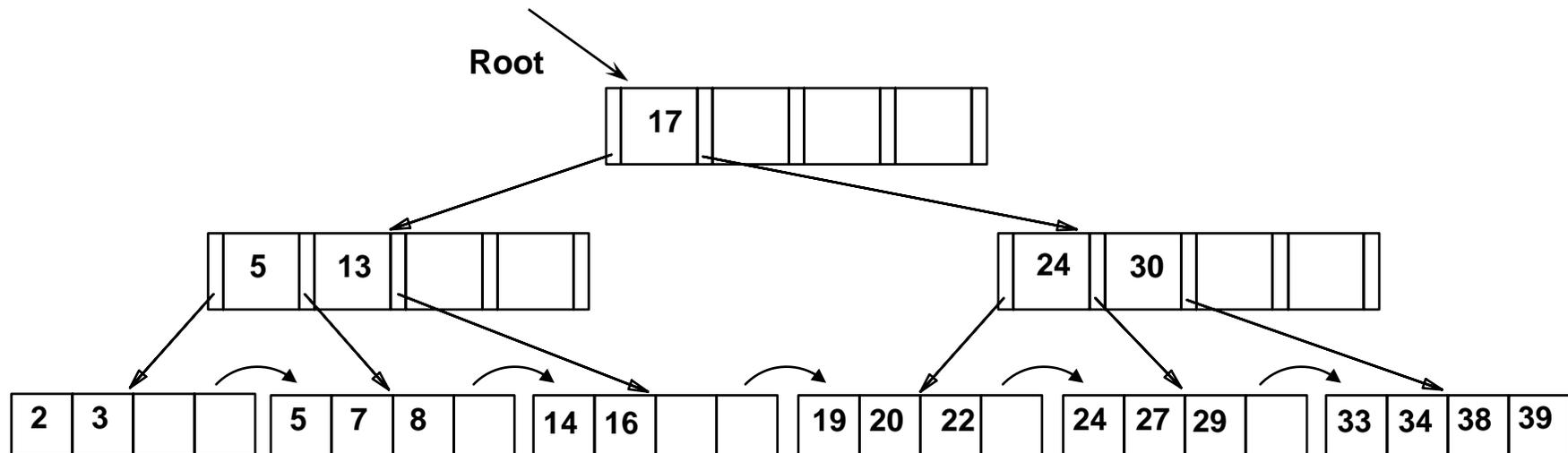
B⁺ Tree: The Most Widely Used Index

- *Height-balanced.*
 - Insert/delete at $\log_F N$ cost ($F = \text{fanout}$, $N = \# \text{ leaf pages}$);
- *Grow and shrink dynamically.*
- Minimum 50% occupancy (except for root).
 - Each node contains $\mathbf{d} \leq m \leq 2\mathbf{d}$ entries. The parameter \mathbf{d} is called the *order* of the tree.
- `next-leaf-pointer' to chain up the leaf nodes.
- Data entries at leaf are sorted.



Example B+ Tree

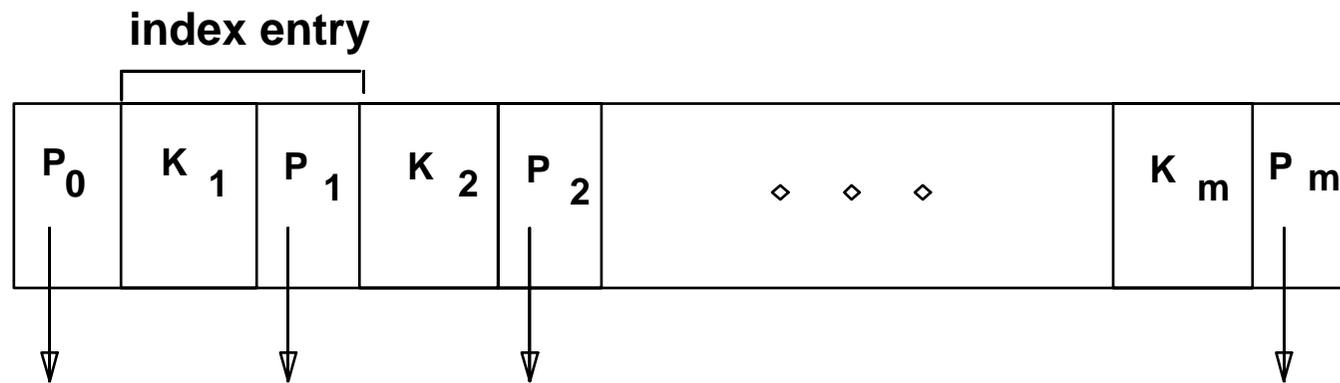
- Each node can hold 4 entries (order = 2)



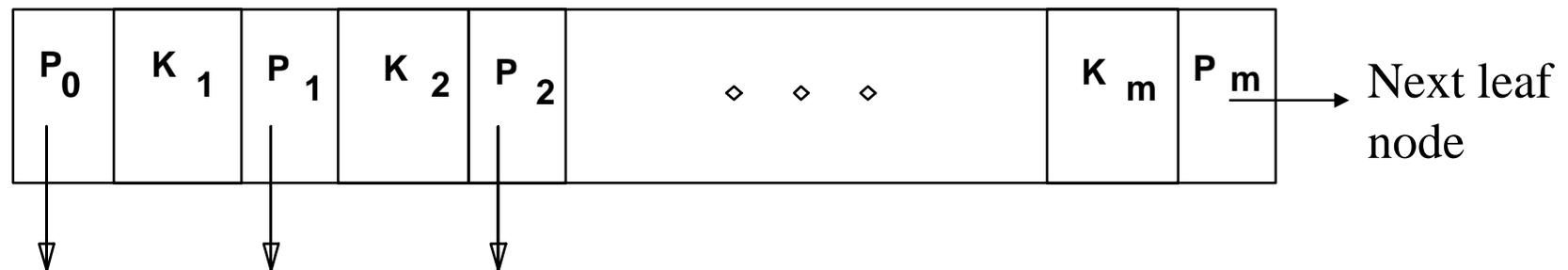


Node structure

- Non-leaf nodes

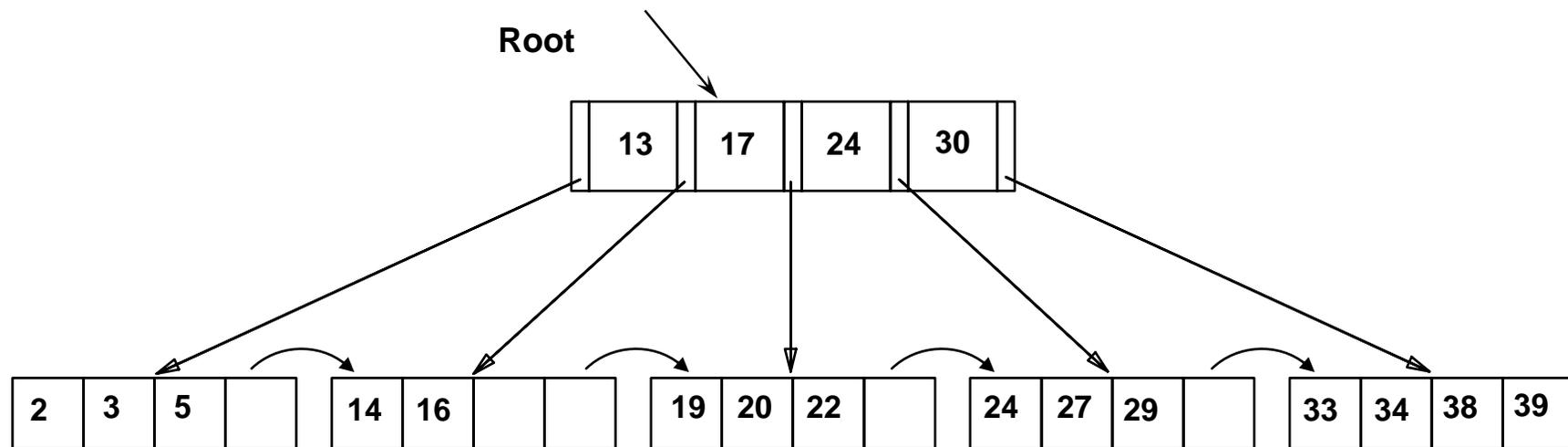


- ⌘ Leaf nodes

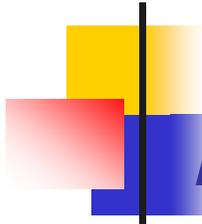


Searching in B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5, 15, all data entries ≥ 24 ...



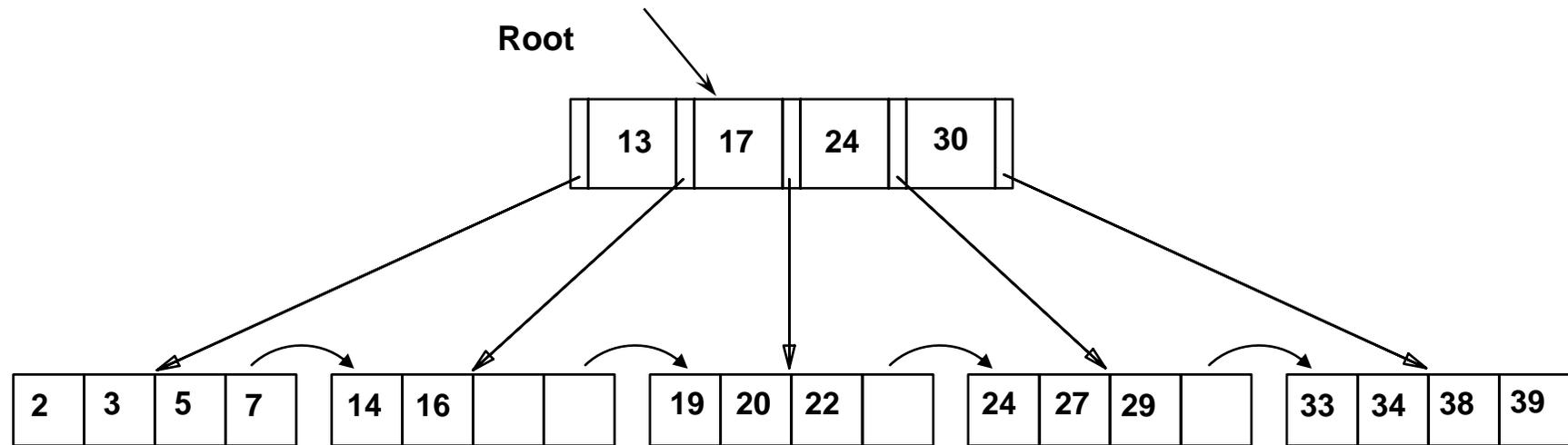
✉ *Based on the search for 15*, we know it is not in the tree!*



Inserting a Data Entry into a B+ Tree

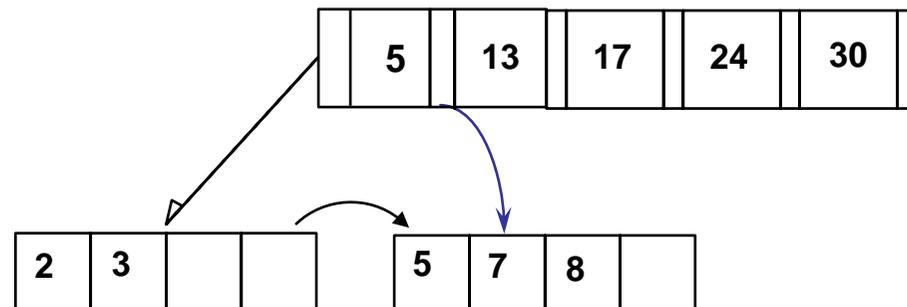
- Find correct leaf L .
- Put data entry onto L .
 - If L has enough space, *done!*
 - Else, must *split* L (into L and a new node $L2$)
 - Redistribute entries evenly, **copy up** middle key.
 - Insert index entry pointing to $L2$ into parent of L .
- This can happen recursively
 - To **split index node**, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits “grow” tree; root split increases height.
 - Tree growth: gets *wider* or *one level taller at top*.

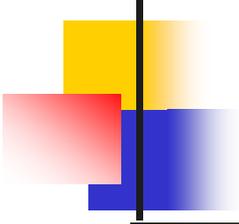
Inserting 7 & 8 into Example B+ Tree



(Note that 5 is copied up and continues to appear in the leaf.)

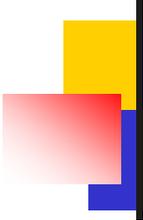
- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.





Overview

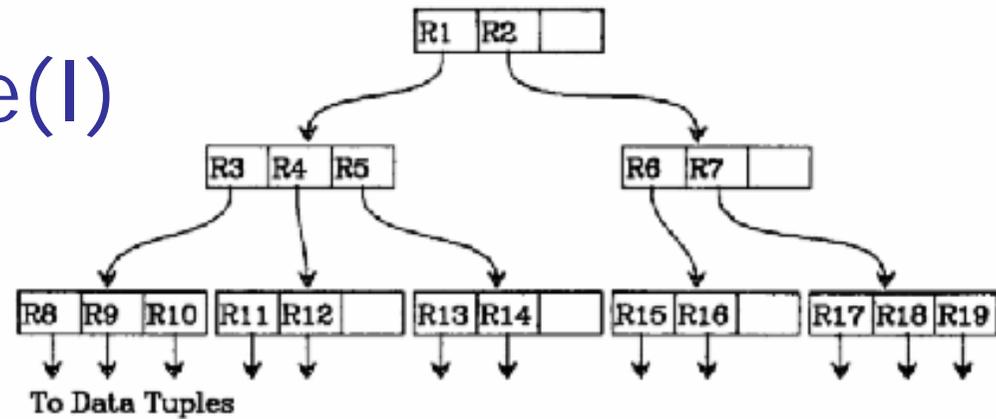
- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- String Indexing
 - Inverted Files
 - Suffix Tree/Array
 - Signature Files



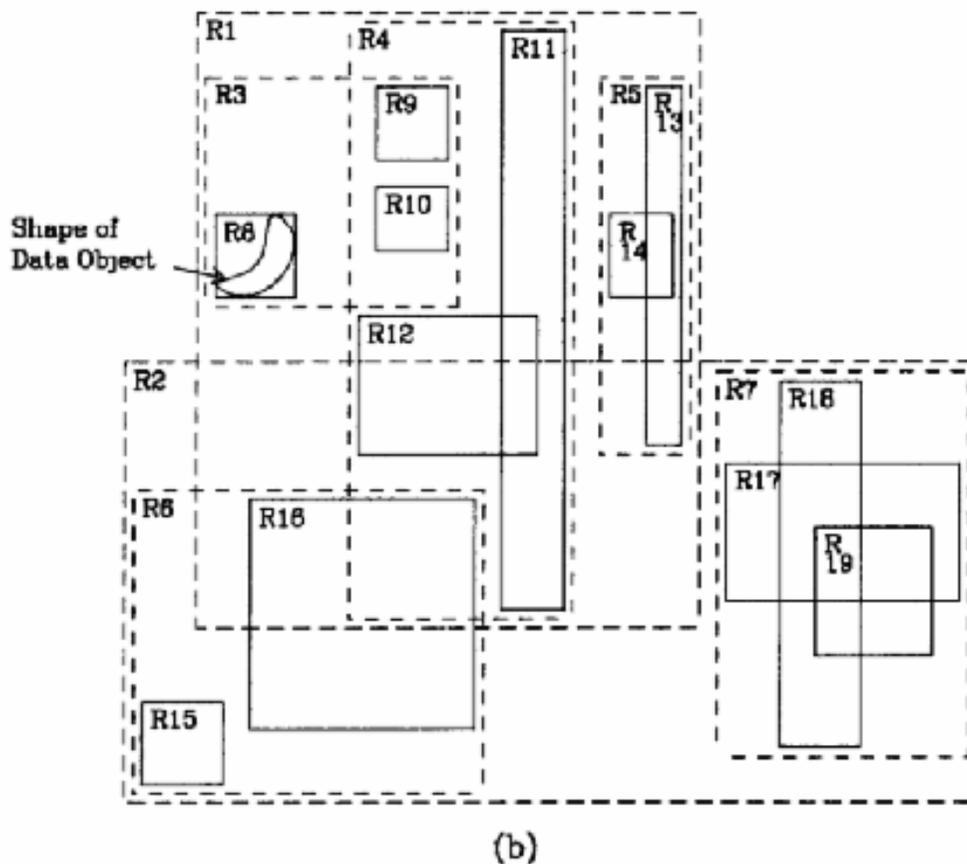
R-Tree Index Structure

- A spatial database consists of a collection of objects representing spatial objects, where each object has a unique identifier, which can be used to retrieve it.
- An R-tree is a height-balanced tree similar to a B-tree, with index records in its leaf nodes containing pointers to data objects.
- Nodes correspond to disk pages if the index is disk-resident.

R-tree: Structure(I)



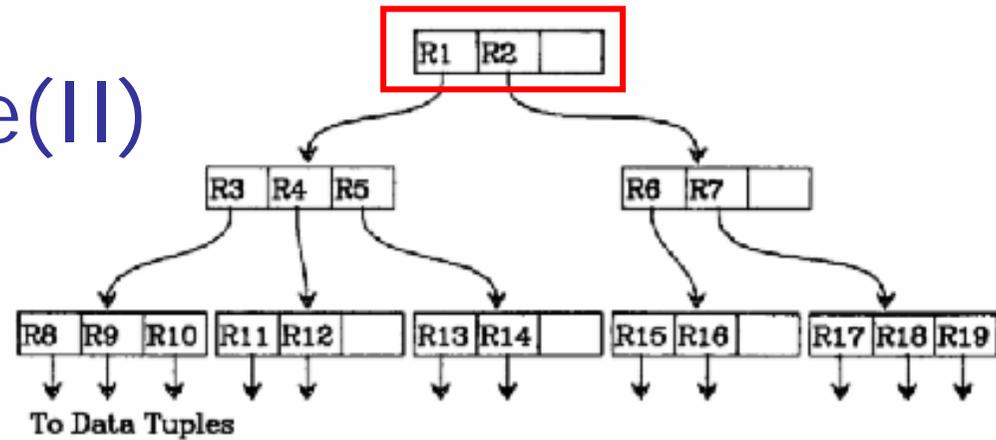
(a)



(b)

We will transverse through the R-Tree to give some idea of its structure

R-tree: Structure(II)

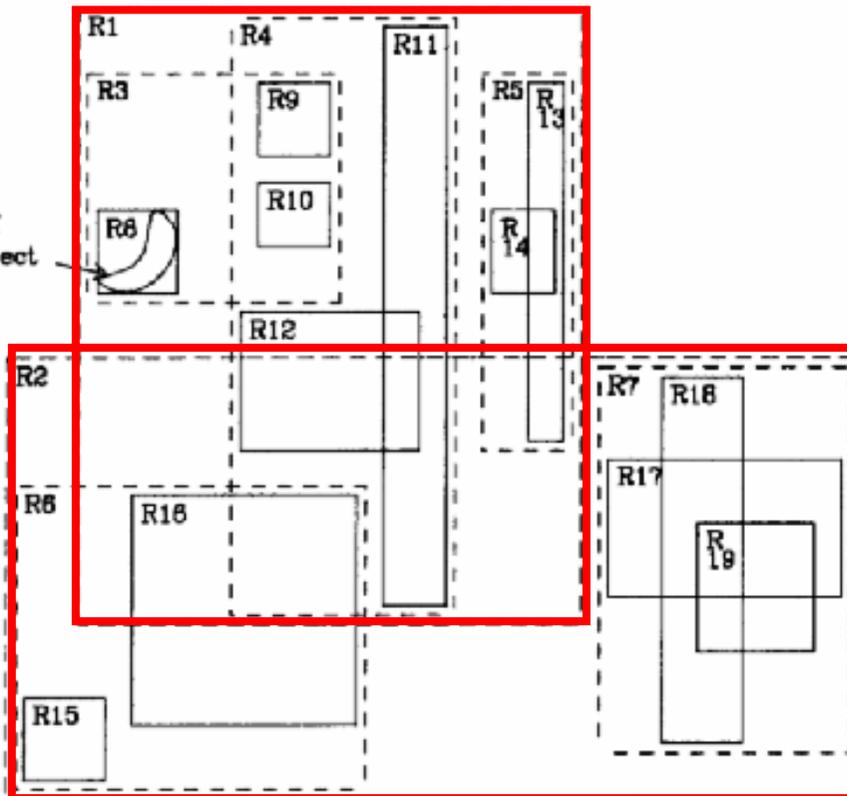


(a)

R1

Shape of Data Object

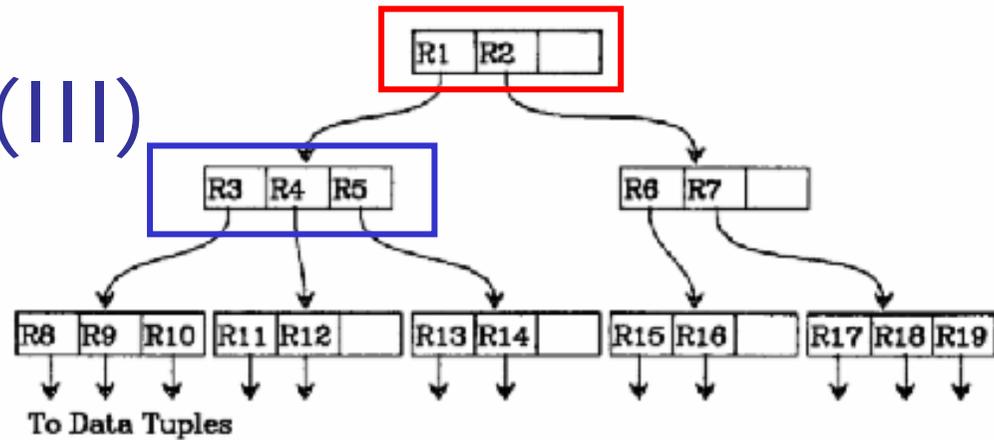
R2



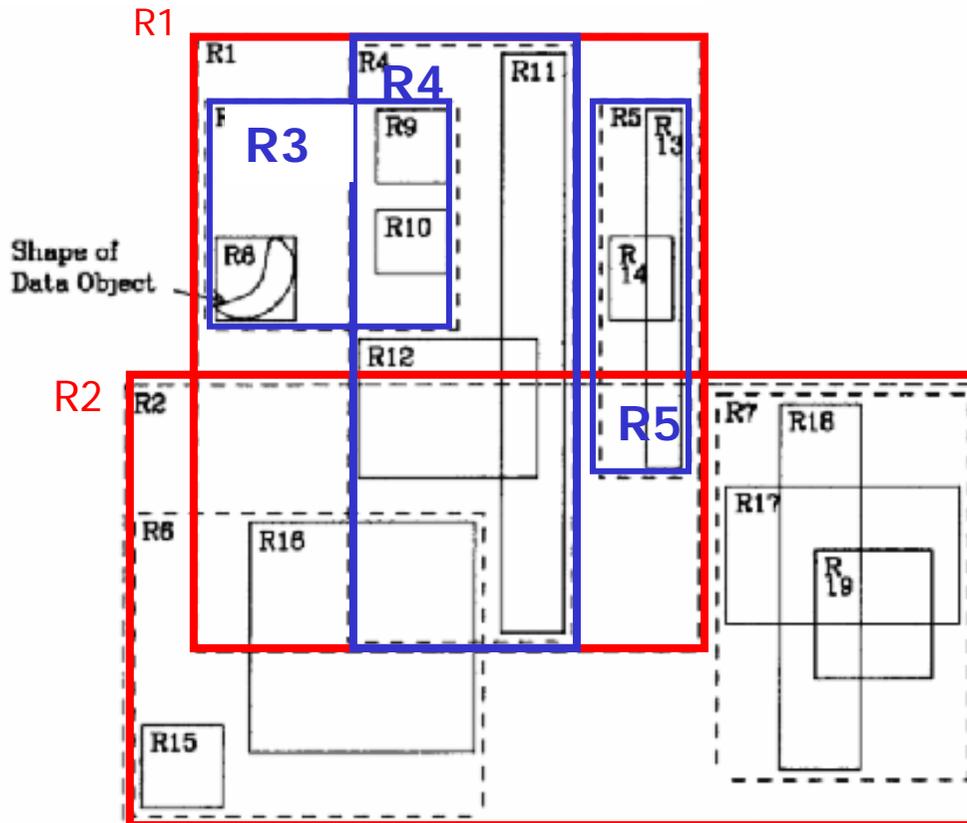
(b)

We first move to the first layer of the R-tree which consist of the two nodes **R1** and **R2**.

R-tree: Structure(III)



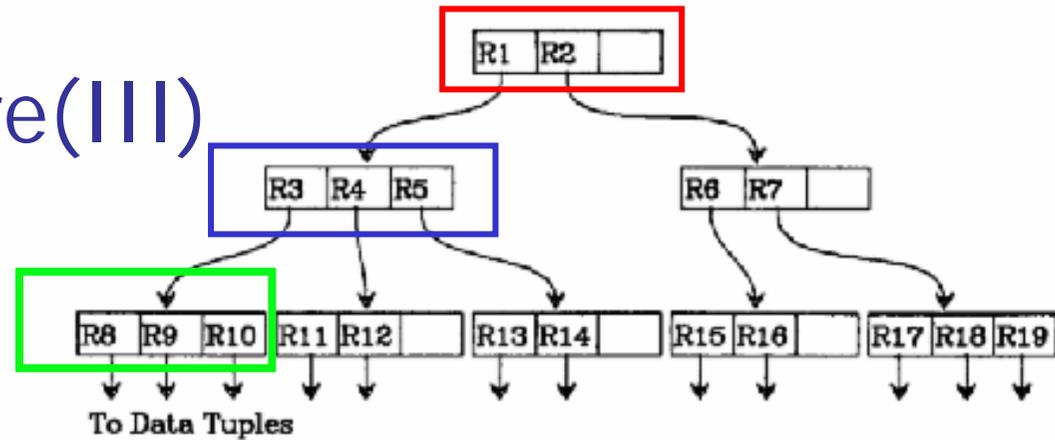
(a)



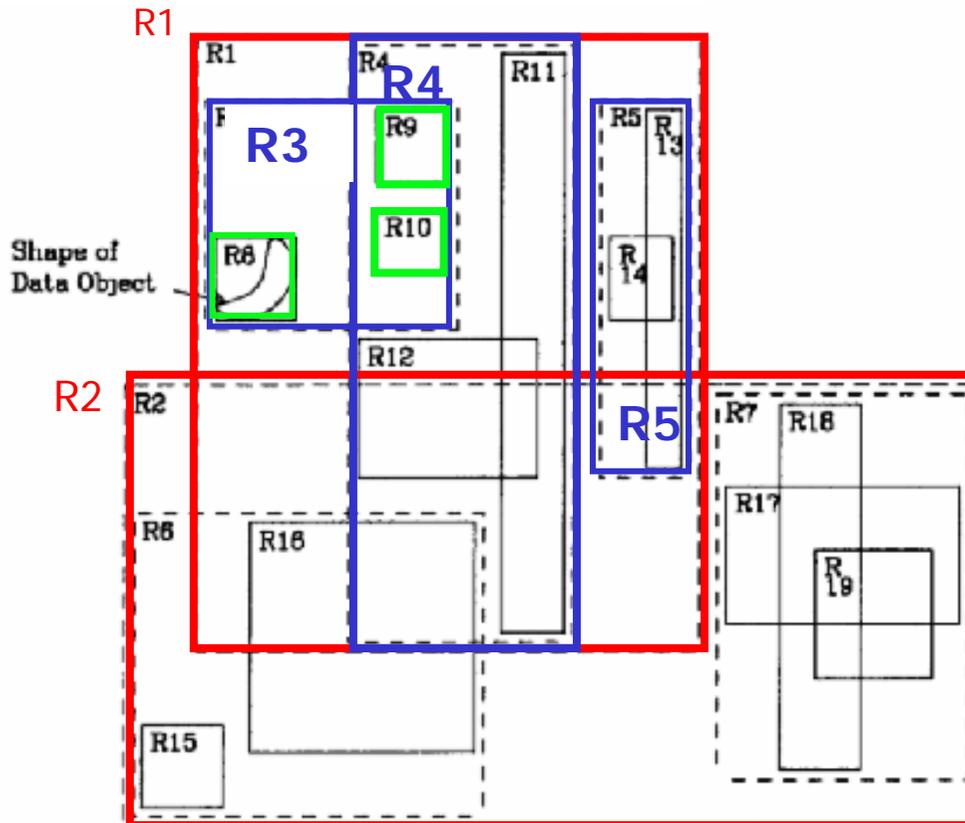
(b)

Then we move to the second layer under node **R1** which consist of node **R3**, **R4** and **R5**

R-tree: Structure(III)



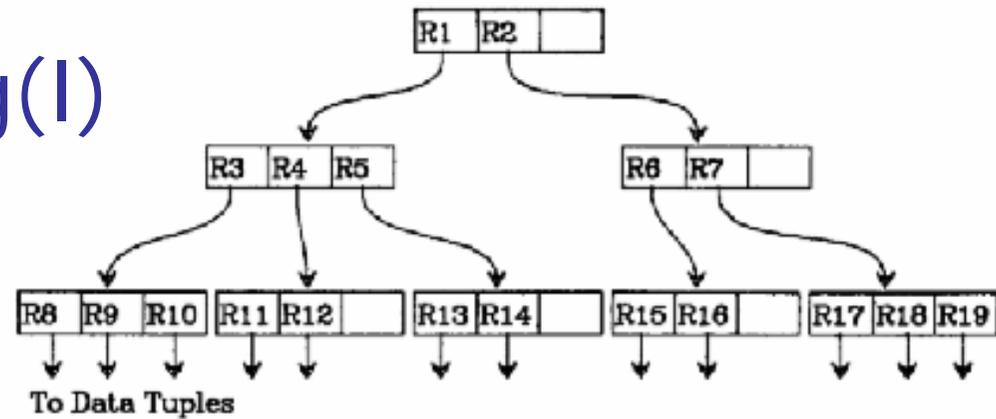
(a)



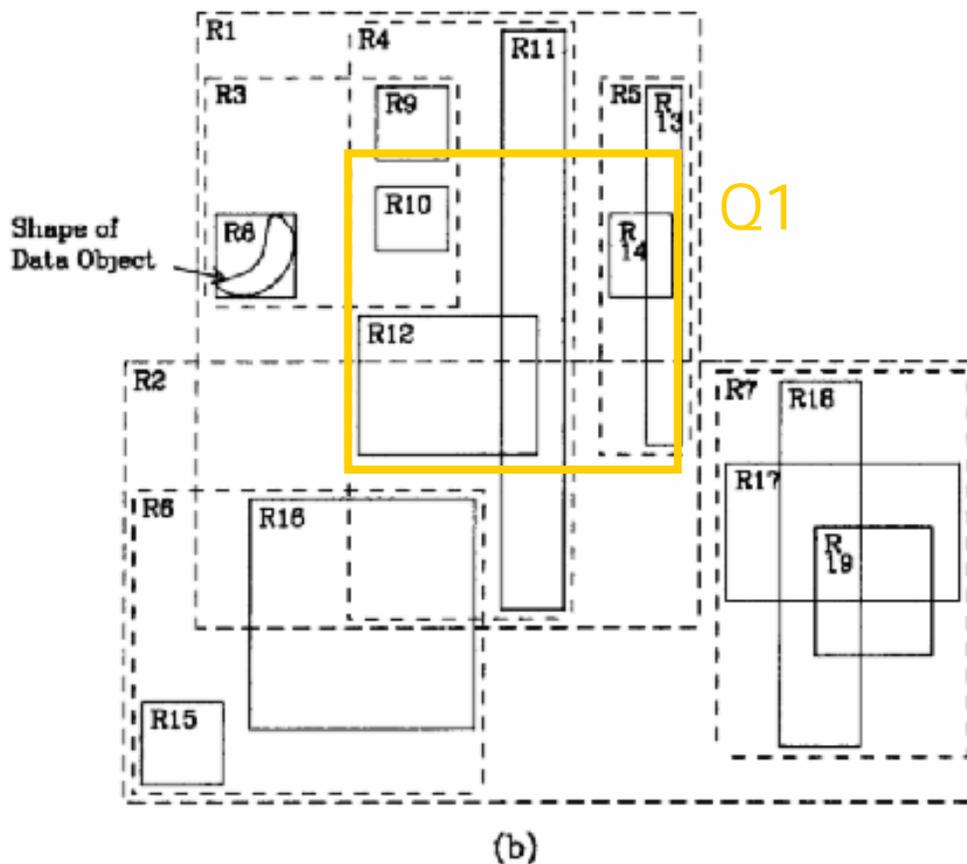
(b)

Then we move to the third layer under node **R3** which consist of node **R8**, **R9** and **R10**

R-tree: Searching(I)

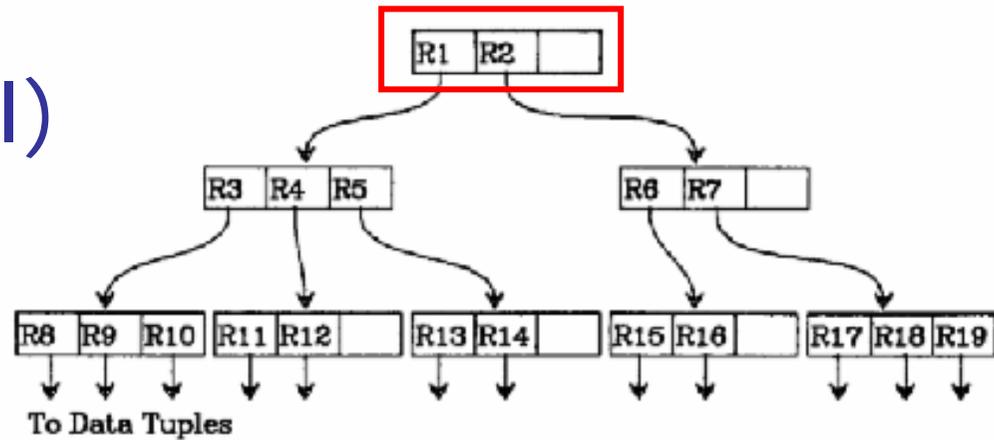


(a)



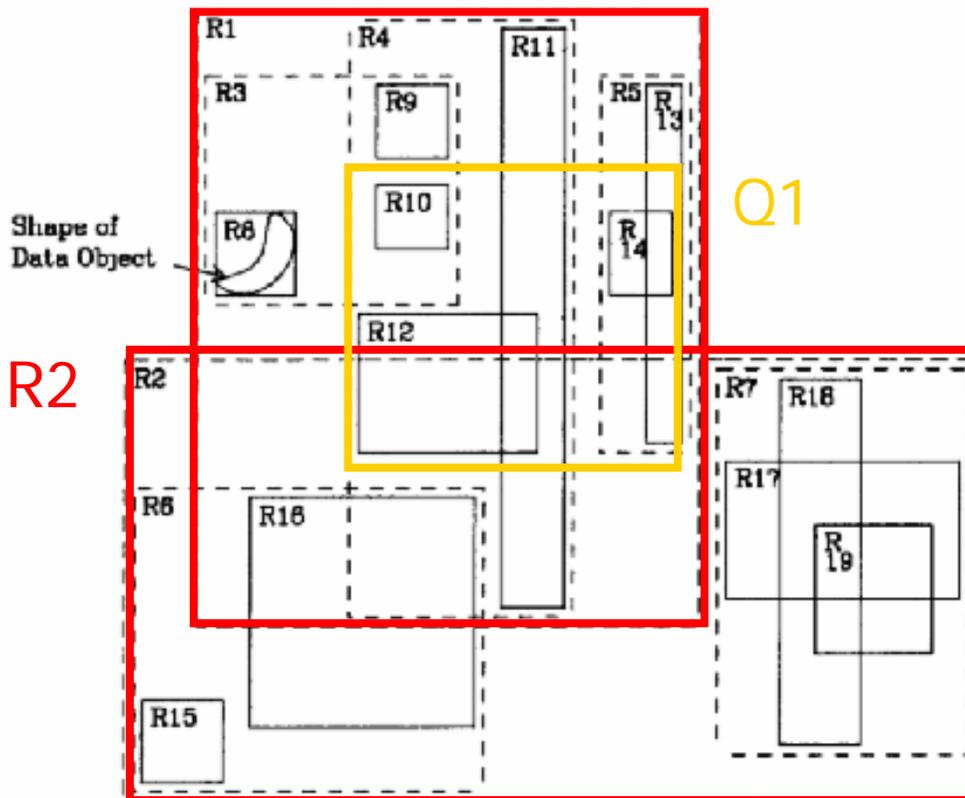
Assuming we want to find all objects lying within the yellow query box Q1

R-tree: Search(II)



(a)

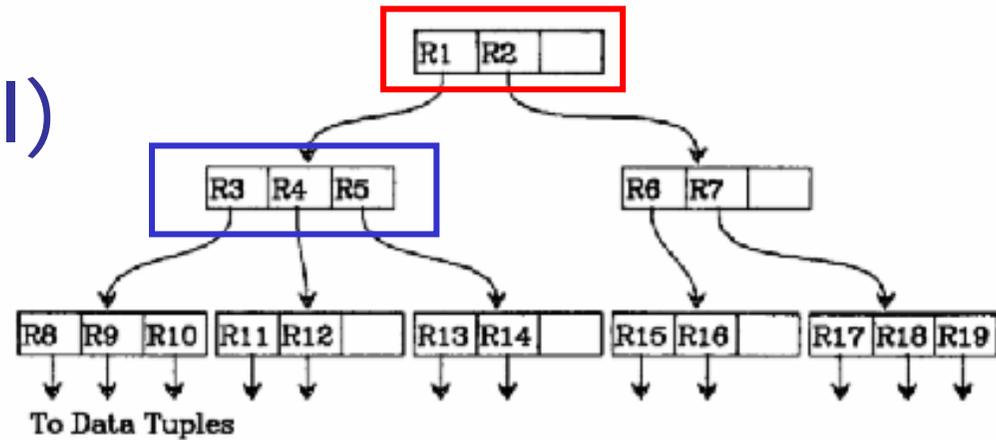
R1



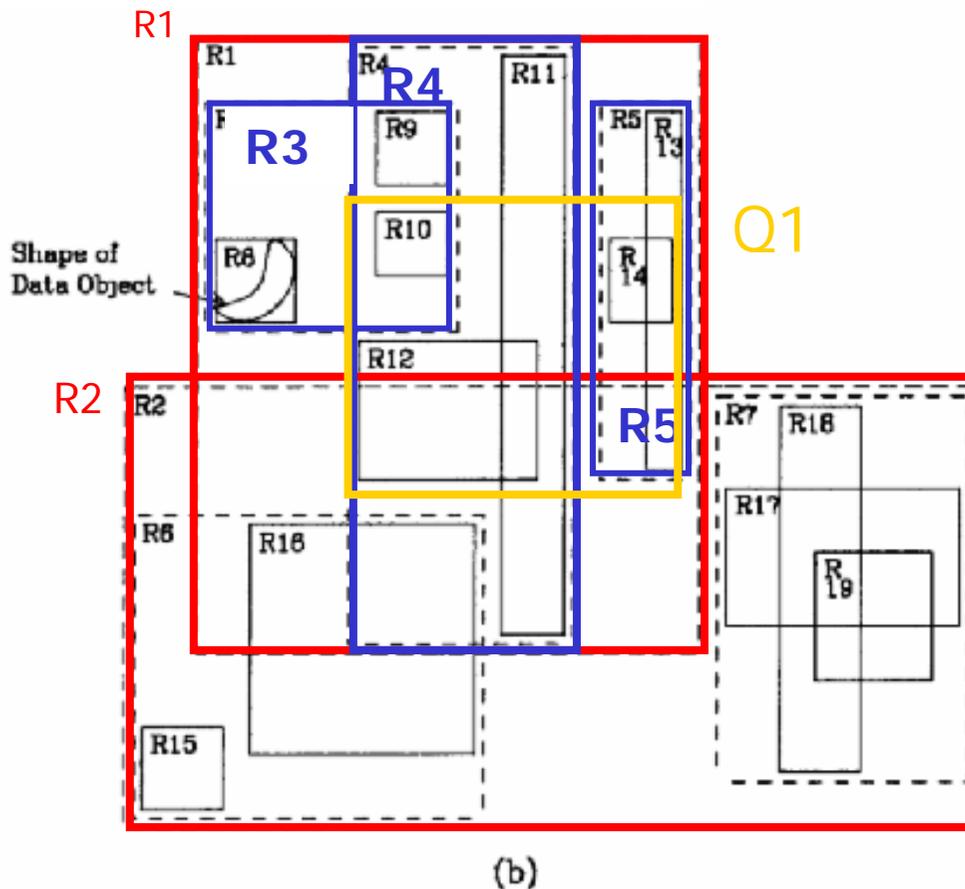
(b)

We first move to the first layer of the R-tree and observe that the Q1 intercept both R1 and R2.

R-tree: Search(III)

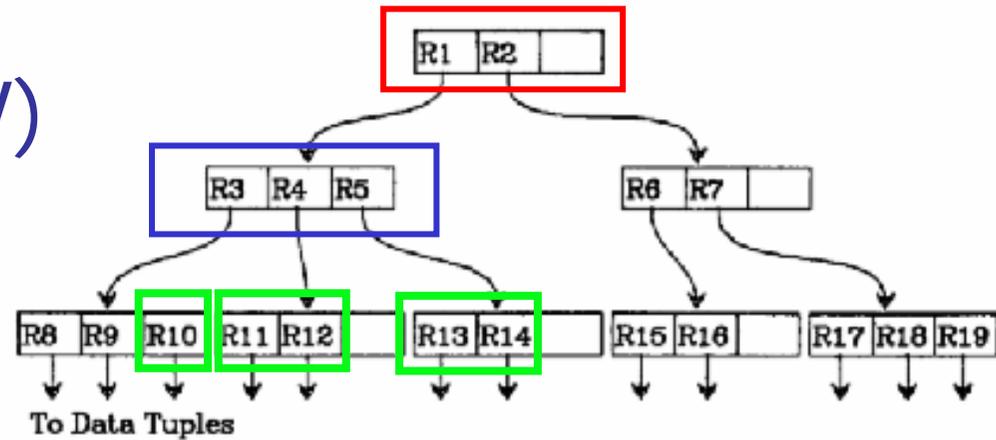


(a)

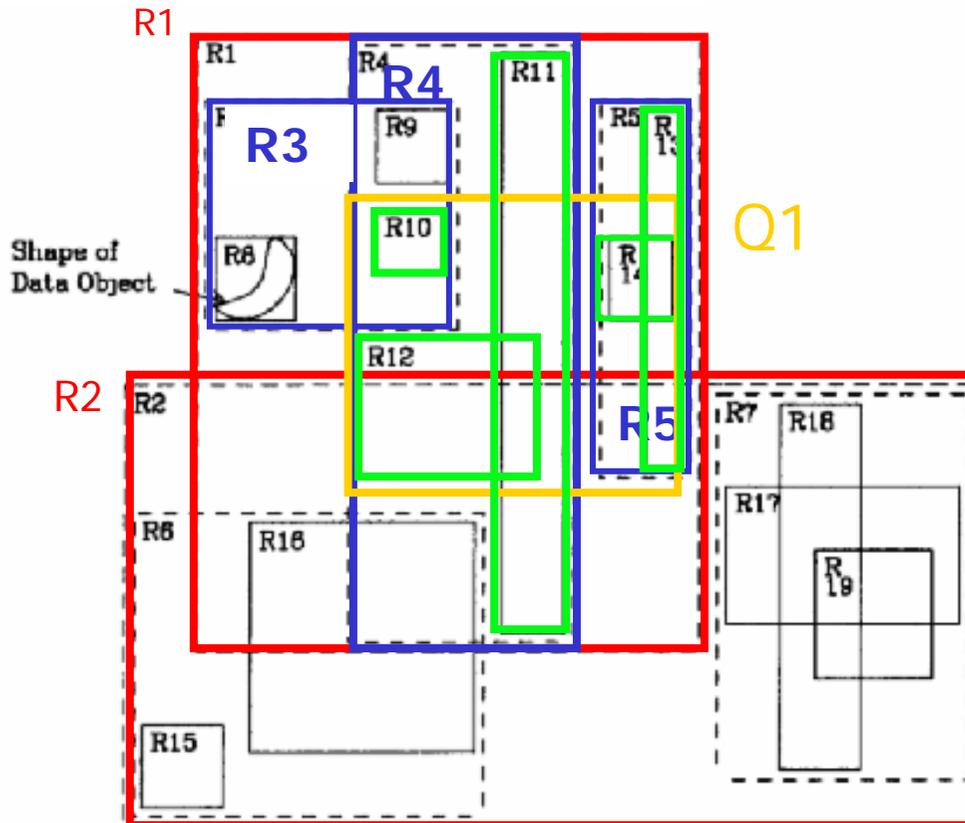


Then we move to the second layer of both **R1** and **R2** and observe that the query box only intercept node **R3**, **R4** and **R5**. **Thus we will not access the children of R6 and R7.**

R-tree: Search(IV)



(a)

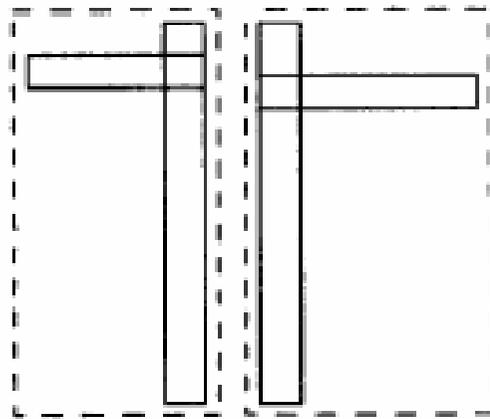


(b)

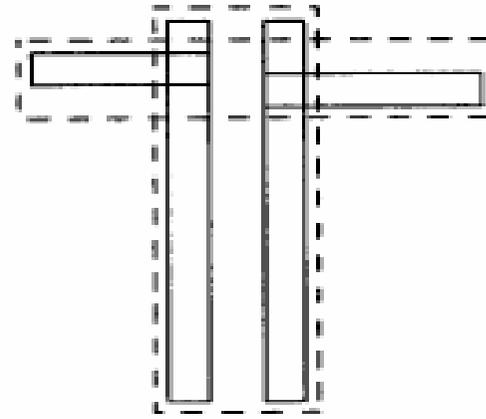
Then we move to the third layer under R3, R4, R5 and find the only R10, R11, R12, R13 and R14 intercept Q1. Thus we will only access the data objects under these five nodes

Maintenance of R-tree

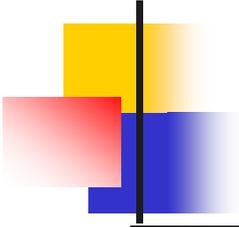
- Very much like the maintenance of B+ Tree. Take average $O(\log n)$ to search, insert and delete.
- Concern when splitting nodes: Need to ensure that the bounding box have the smallest area



Bad split

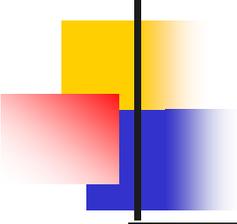


Good split



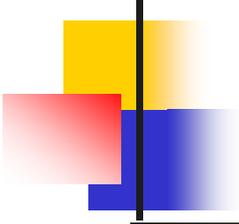
NN Search Problem

- Given a point p , find its nearest neighbor from all points of database.
- Eg:
 - Find the nearest hotel to some theater
 - Find the nearest star to some space point
 - Find the most similar picture to some given picture



Naive Algorithm

- Compare point by point
 - $\text{min} = \infty$
 - Pick a random point q from the data set, return the distance between them
 - If the distance is less than min , update min
- Obviously, not efficient



Improved algorithm

- Construct an index such as **R-tree** for the data
- To prune off those MBRs which can never contain a nearest neighbor.

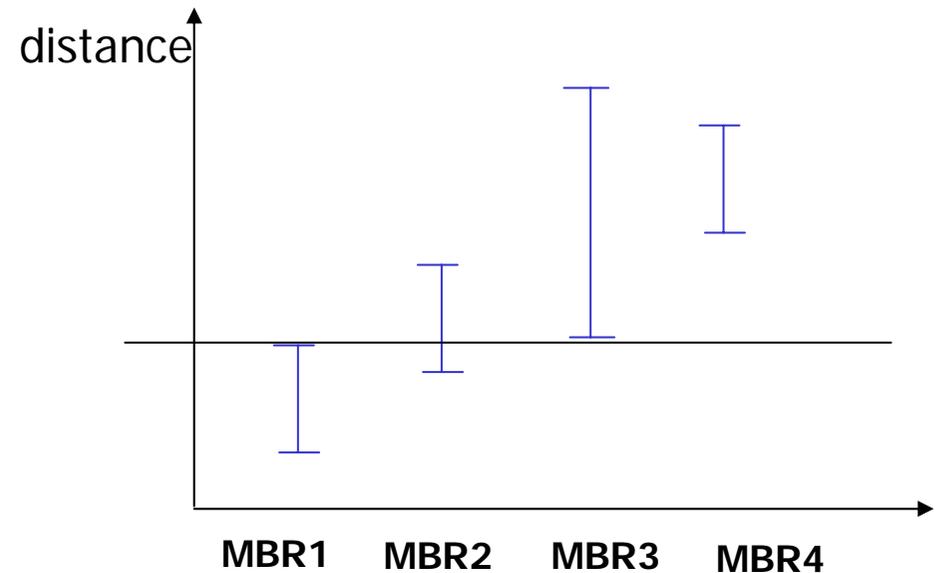
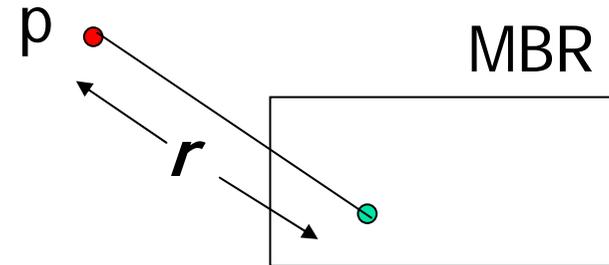
R-tree: Finding Nearest Neighbor

- First problem to be solve:

- Given a query point p and a minimum bounding rectangle (MBR), find an **upper bound** and **lower bound** for the distance r such that there **exists at least one point in the MBR that is within a distance of r from p .**

- Why ?

- To facilitate some form of ranking to prune off uninteresting MBRs
- Eg. we can prune off MBR3 and MBR4.

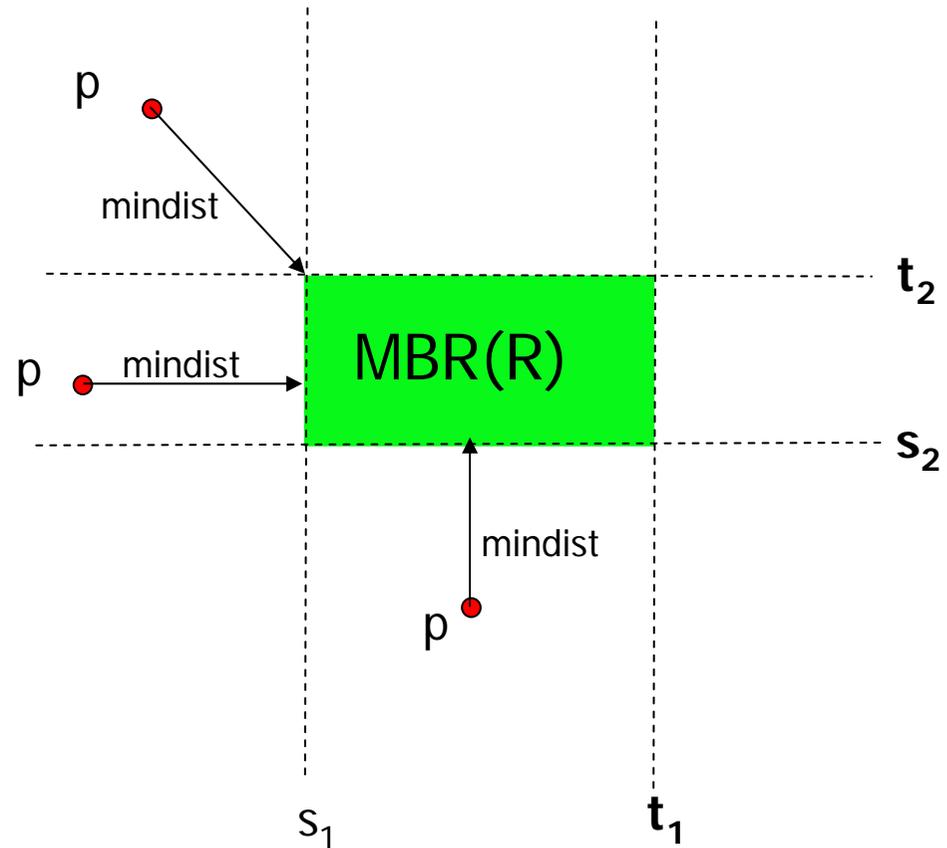


R-tree: Finding Lower Bound(MINDIST)

$$MINDIST(p, R) = \sum_{i=1}^n |p_i - r_i|^2$$

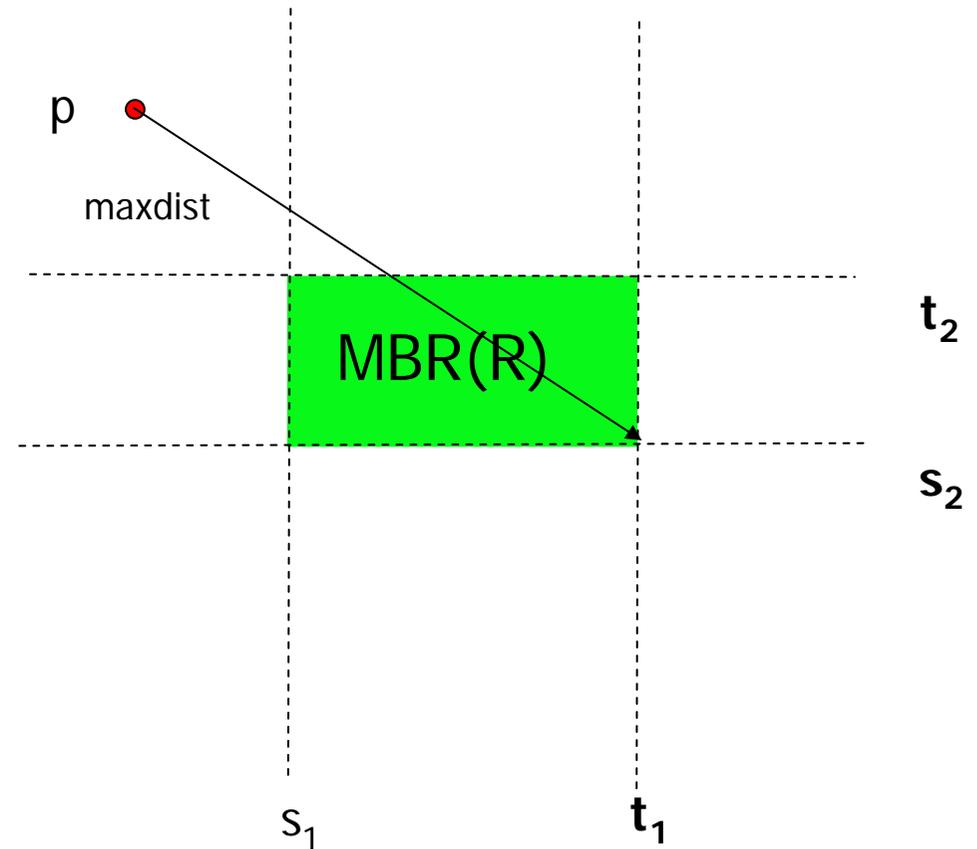
$$r_i = \begin{cases} s_i & \text{if } p_i < s_i; \\ t_i & \text{if } p_i > t_i; \\ p_i & \text{otherwise;} \end{cases}$$

Note: p is located at (p_1, p_2) i.e p_1 and p_2 are the x-y coordinates!



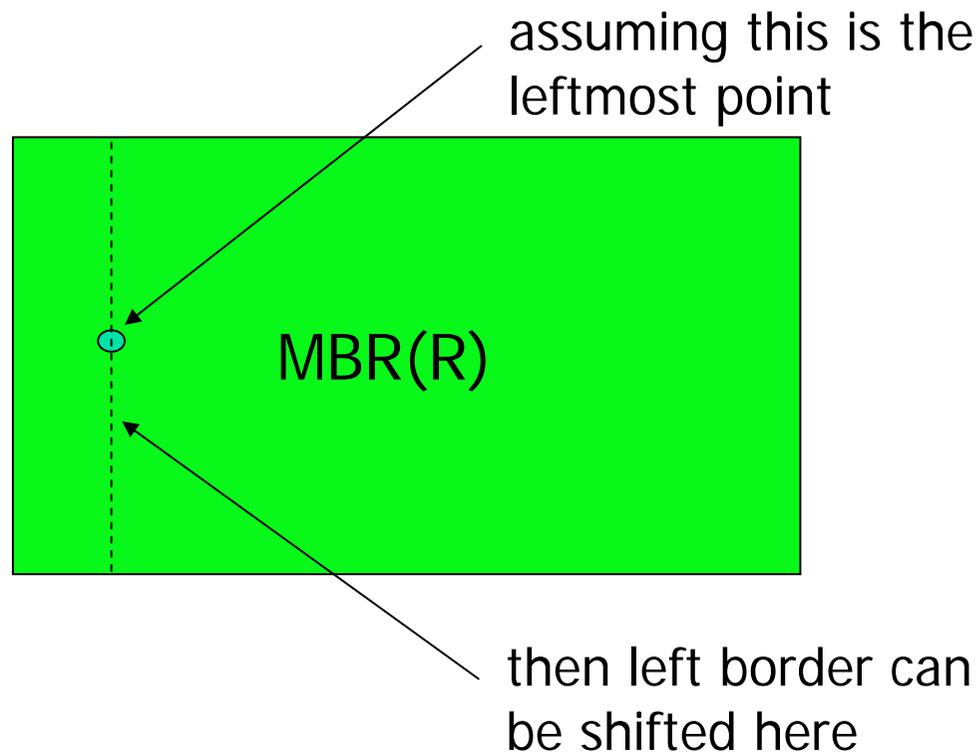
R-tree: Finding Upper Bound(I)

- **Most trivial solution:** Find the point in MBR which is furthest away
- ***But we can do better with a certain observation!***



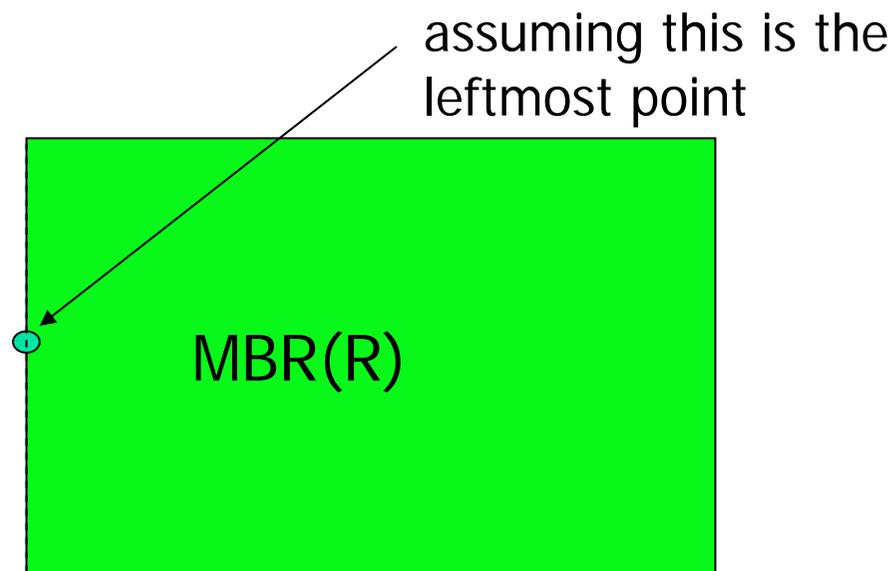
R-tree: Finding Upper Bound(II)

- **Important observation:** Every face of any MBR should contains at least one point in the DB.



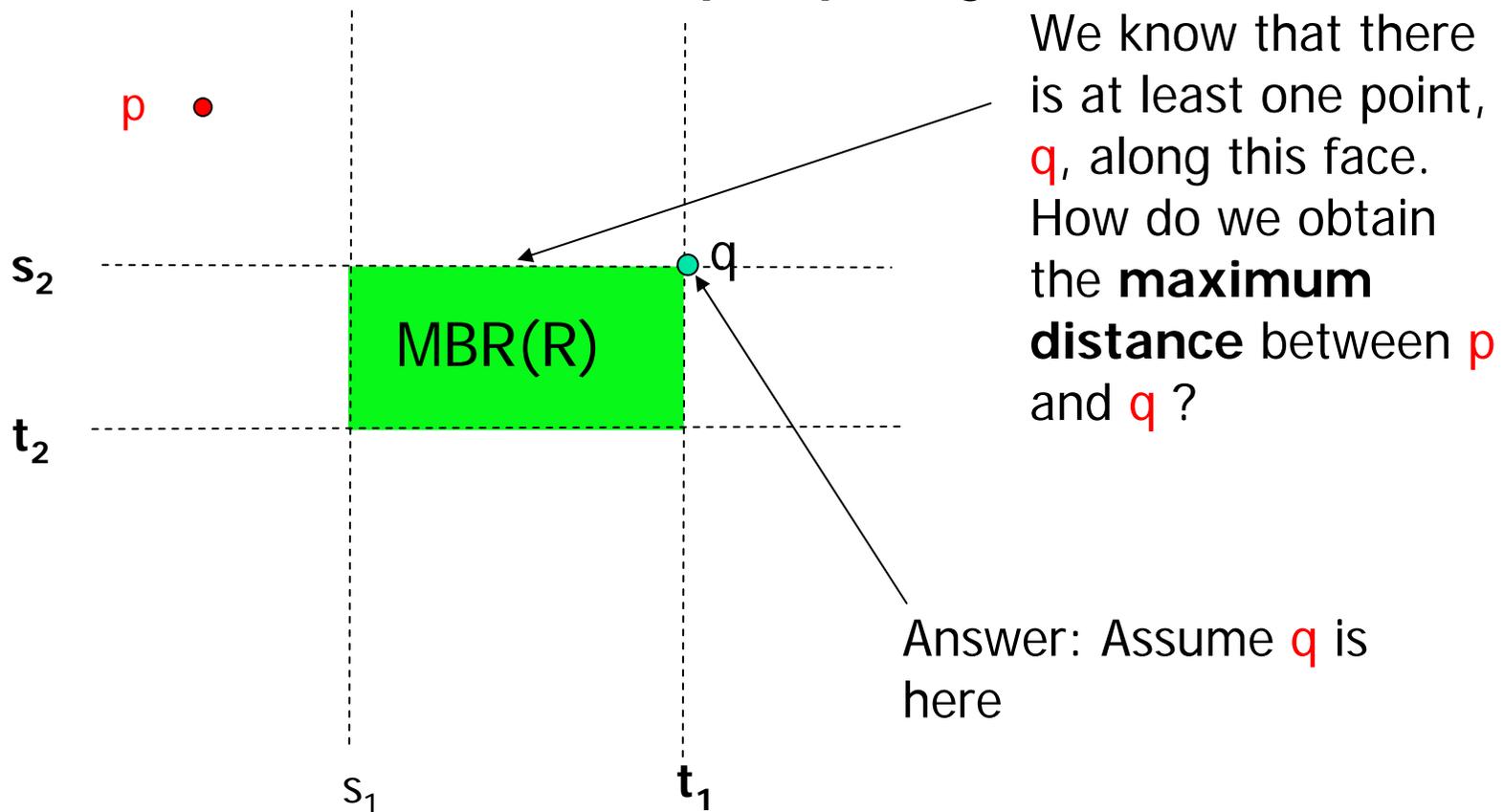
R-tree: Finding Upper Bound(II)

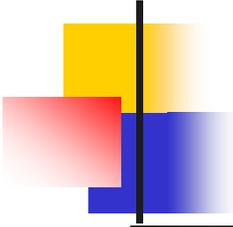
- **Important observation:** Every face of any MBR should contains at least one point in the DB.



R-tree: Finding Upper Bound(III)

- We can now find a better upper bound based on this property.





R-tree: Finding Upper Bound(IV)

- Find the upper bound MINMAXDIST by performing the following steps:
 - For each face of the MBR, pick the location \mathbf{q} that is furthest away from \mathbf{p} and insert $\text{dist}(\mathbf{p}, \mathbf{q})$ into a set QSET
 - Pick the *minimum* out of QSET to be MINMAXDIST, the upper bound for r such that there exist at least one point q from the MBR such that $\text{dist}(\mathbf{p}, \mathbf{q}) \leq r$

Actual Calculation of MINMAXDIST

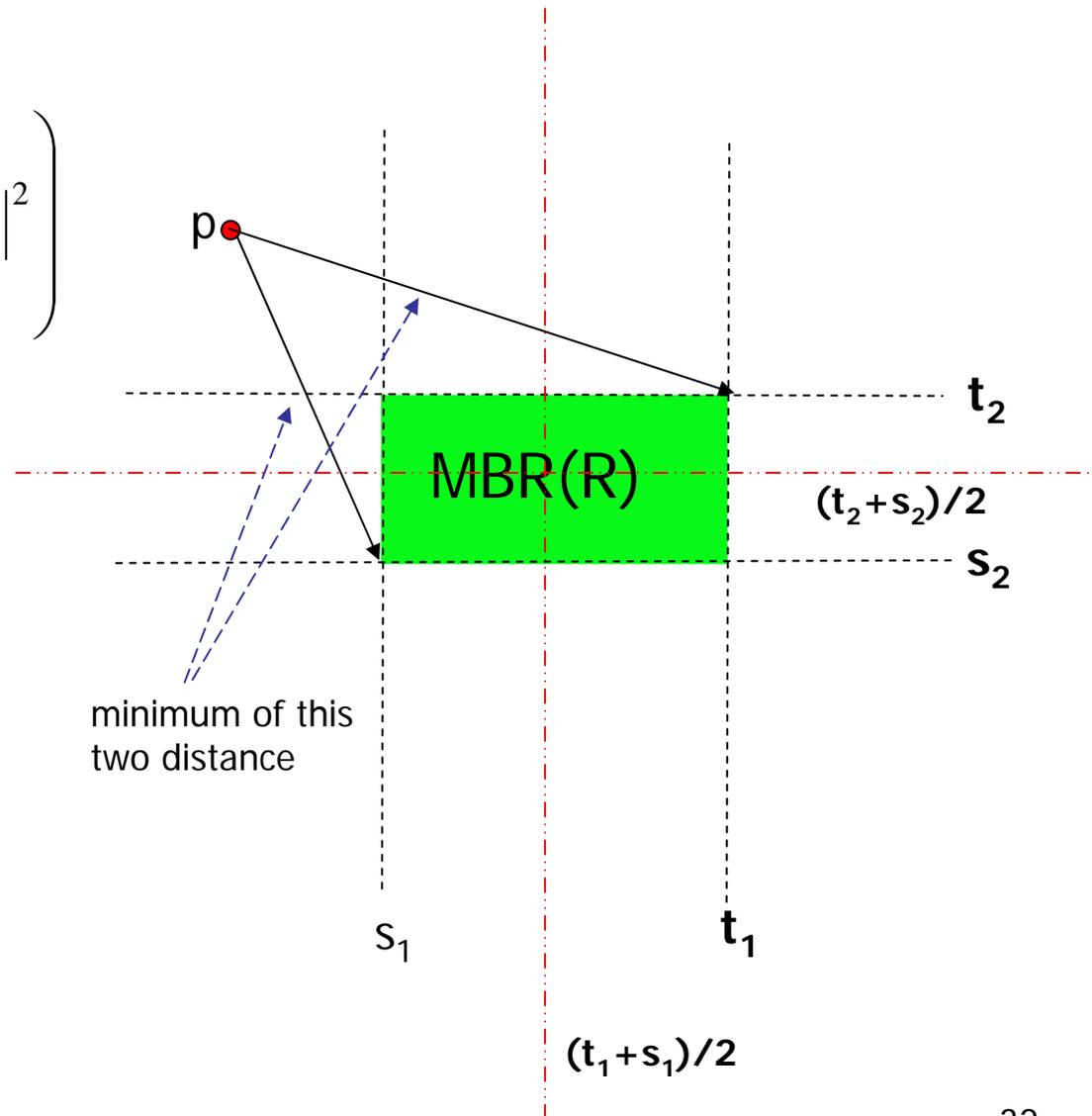
$$\min_{1 \leq k \leq n} \left(|p_k - rm_k|^2 + \sum_{\substack{i \neq k \\ 1 \leq i \leq n}} |p_i - rM_i|^2 \right)$$

$$rm_k = s_k \text{ -if : } p_k \leq \frac{(s_k + t_k)}{2}$$

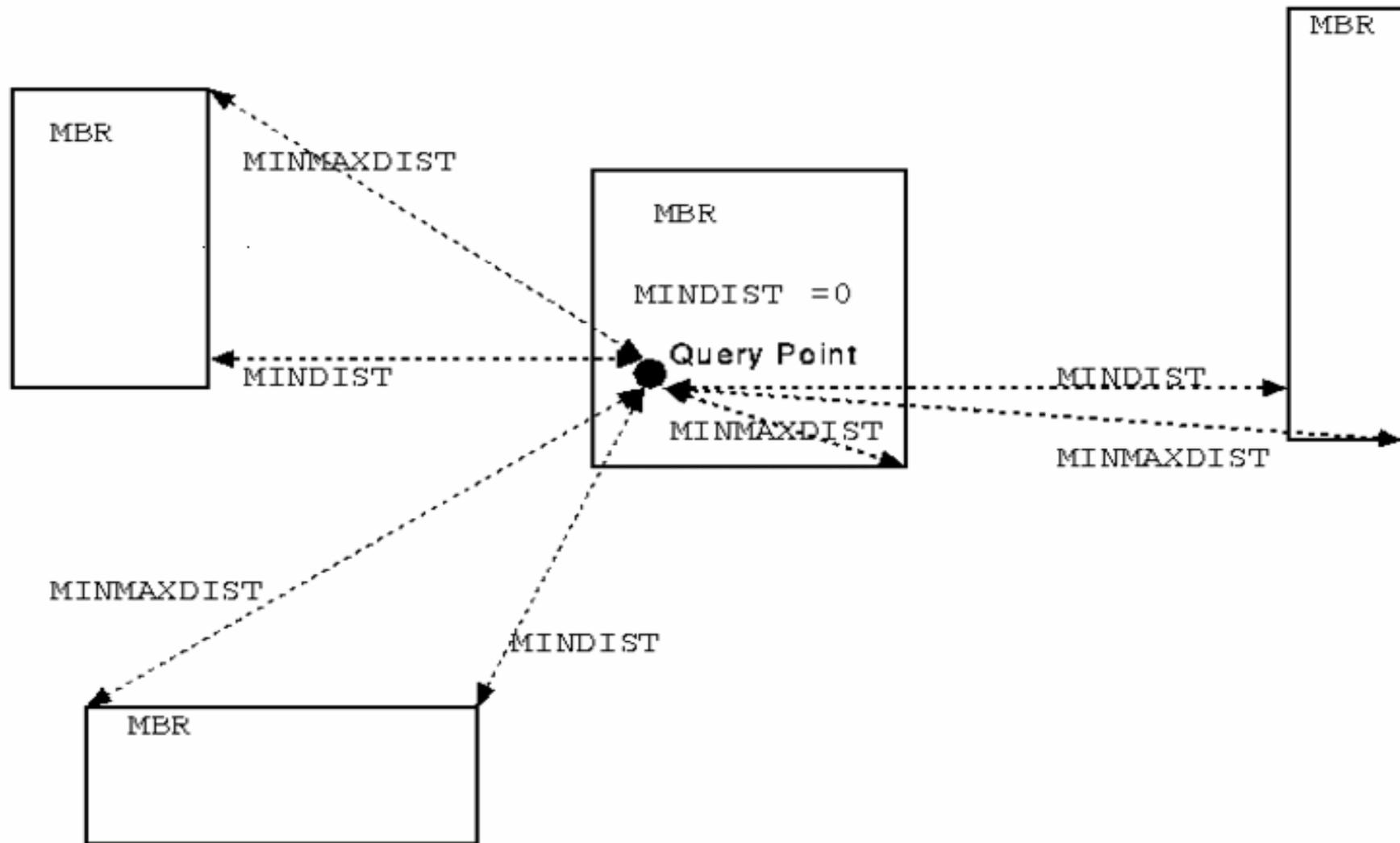
$$rm_k = t_k \text{ -otherwise}$$

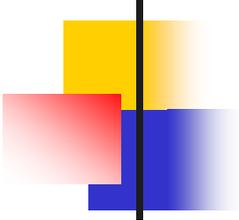
$$rM_i = s_i \text{ -if : } p_i \geq \frac{(s_i + t_i)}{2}$$

$$rM_i = t_i \text{ -otherwise}$$



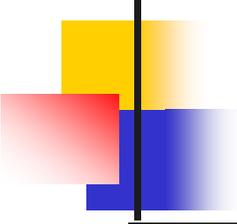
R-tree: Example





Overview

- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- String Indexing
 - Inverted Files
 - Suffix Tree/Array
 - Signature Files



1-NN Algorithm for R-tree(I)

- **Search ordering:**

- **MINDIST**: optimistic
- **MINMAXDIST**: pessimistic.

- **Search pruning:**

- **Downward pruning**: An MBR R is discarded if there exists another R' such that

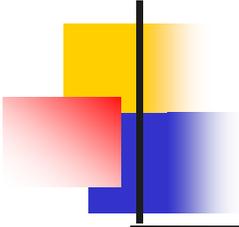
$$\text{MINDIST}(P,R) > \text{MINMAXDIST}(P,R')$$

- **Downward pruning**: An object O is discarded if there exists an R such that

$$\text{ACTUAL-DIST}(P,O) > \text{MINIMAXDIST}(P,R)$$

- **Upward pruning**: An MBR R is discarded if an object O is found such that

$$\text{MINDIST}(P,R) > \text{ACTUAL_DIST}(P,O)$$



1-NN Algorithm for R-tree(II)

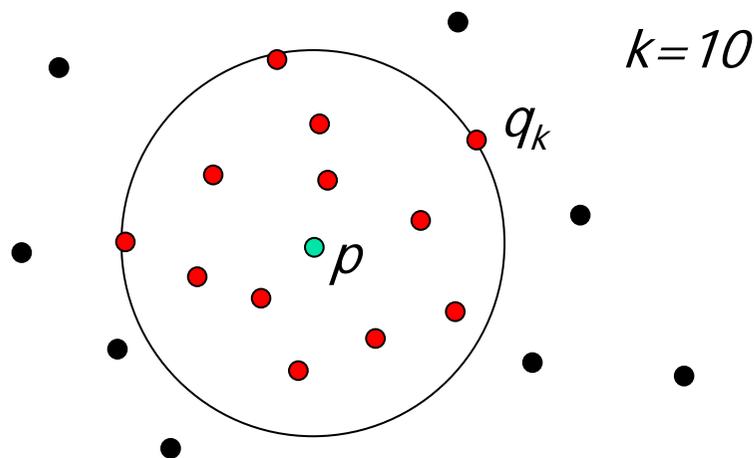
- **best first traversal** of the nodes in the R-tree.
- A heap is maintained for storing every MBR
- The algorithm maintains a variable **Best** which is initially set to ∞ and are updated later

k-Nearest Neighbors

- Definition: Given a query point p , and a distance function $dist()$, let q_k be a point in the database such that

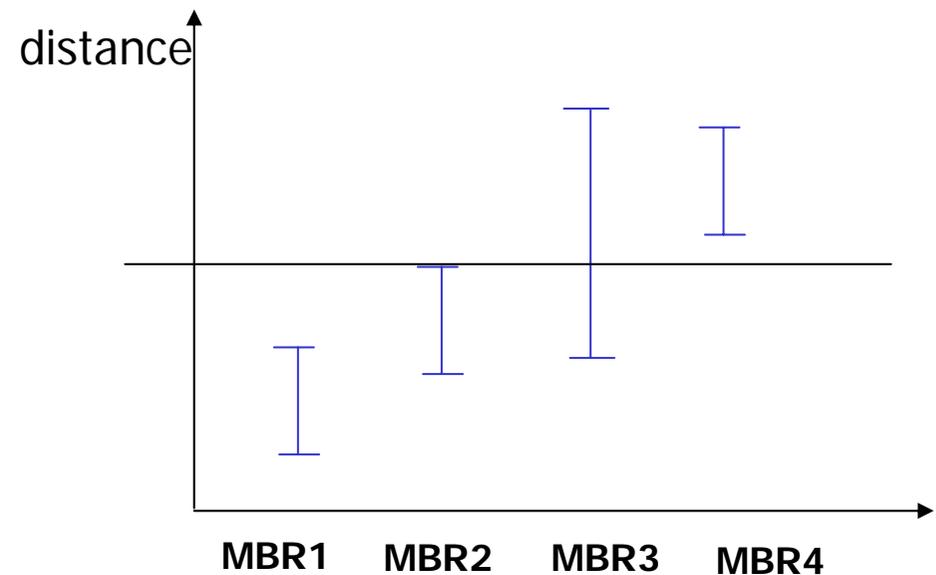
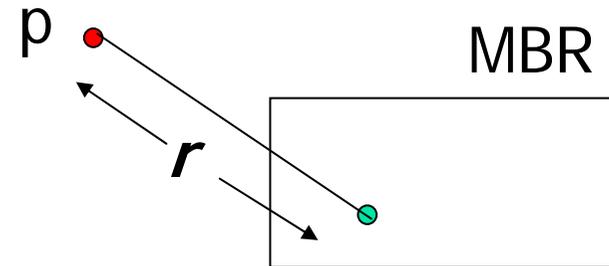
$$\text{count}(\{q \mid \text{dist}(p,q) < \text{dist}(p,q_k), q \in D\}) = k-1$$

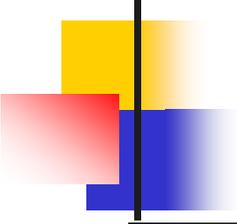
The k -nearest neighbors of p are all points q such that $\text{dist}(p,q) \leq \text{dist}(p,q_k)$



R-tree: Finding k-Nearest Neighbors

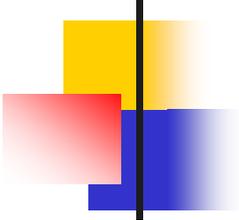
- First problem to be solve:
 - Given a query point p and a minimum bounding rectangle (MBR), find an **upper bound** and **lower bound** for the distance r such that there **exists at least one point in the MBR that is within a distance of r from p** .
- Why ?
 - To facilitate some form of ranking to prune off uninteresting MBRs
 - Eg. if $k=2$, we can prune off MBR4.





k-NN Algorithm

- Keep a sorted buffer of at most k current nearest neighbors
- Pruning is done according to the distance of the furthest nearest neighbor in this buffer

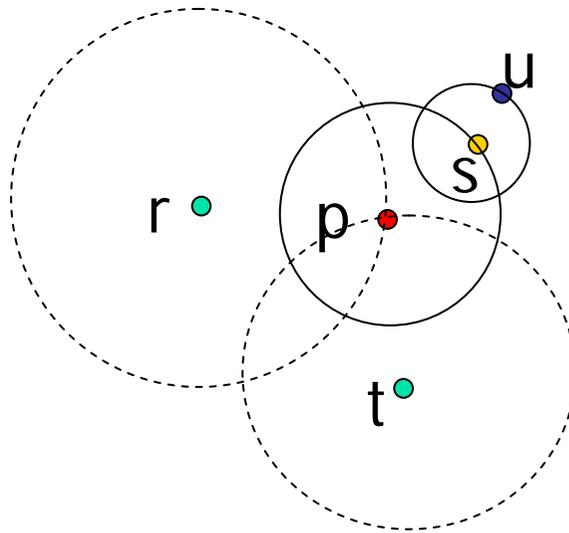


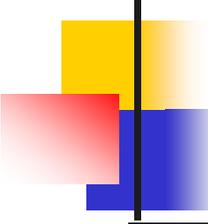
Overview

- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- String Indexing
 - Inverted Files
 - Suffix Tree/Array
 - Signature Files

Reverse Nearest Neighbors

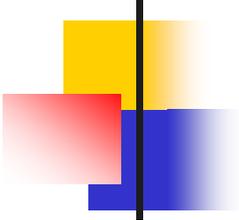
- The reverse nearest neighbors of a point p , $RNN(p)$ are those points which have p as their 1-nearest neighbor
- Example: $RNN(p) = \{r, t\}$, $RNN(s) = \{p\}$ i.e. s is the nearest neighbor of p , $RNN(u) = \{s\}$





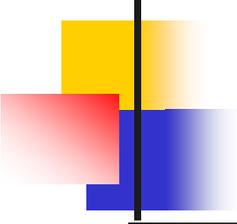
RNN: Potential Applications

- **FedEx Drop-off Points:** Who are the customers who have a particular FedEx drop-off point as their nearest drop-off point.
- **Competitors Analysis:** Does my video shop have a lot of reverse neighbors ?
- **Data Mining:** Spatial reasoning
 - Which location in the data space have the most RNN ?
=> most dense point
 - What does it mean if $NN(p)$ **is not in** $RNN(p)$? What does it mean if $NN(p)$ **is in** $RNN(p)$?



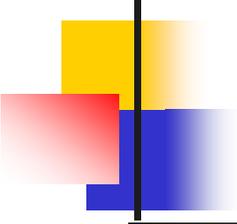
Overview

- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- **String Indexing**
 - Inverted Files
 - Suffix Tree/Array
 - Signature Files



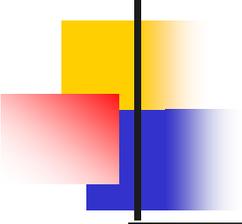
Introduction(I)

- On-line text searching(=sequential searching)
 - involves finding the occurrences of a pattern in a text when the text is not preprocessed.
 - is appropriate when the text is small
 - is the only choice if the text collection is very volatile (i.e. undergoes modifications very frequently), or the index space overhead cannot be afforded.



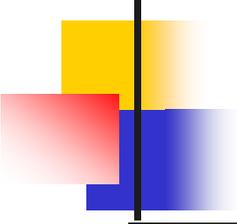
Introduction(II)

- Indexed searching
 - builds data structures over the text(called indices) to speed up the search.
 - is appropriate when the text collection is large and semi-static
 - Semi-static collection : is updated at reasonably regular intervals but are not deemed to support thousands of insertion of single words per second.
- Indexing techniques
 - inverted files, suffix arrays, and signature files
 - Consider search cost, space overhead, and cost of building and updating indexing structures



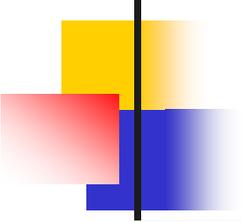
Introduction(III)

- Indexing technique
 - Inverted files
 - Word oriented mechanism for indexing a text collection
 - Composed of *vocabulary* and *occurrences*
 - are currently the best choice for most application.
 - Suffix arrays
 - are faster for phrase searches and other less common queries.
 - are harder to build and maintain.
 - Signature files
 - Word oriented index structures based on hashing
 - were popular in the 1980s



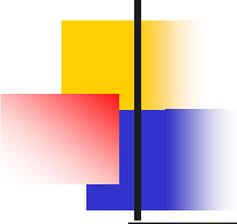
Introduction(IV)

- Indexed structure
 - Trie
 - sorted arrays, binary search tree, B-tree, hash table, etc.
- Notations
 - n : the size of the text database
 - m : pattern length
 - M : amount of main memory available
- Stop Words
 - a list of words considered to have no indexing value
 - Ex. a, an, any, the, to, with, from, for, of, that, who



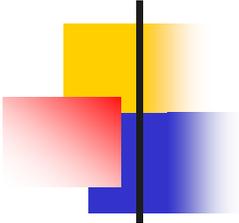
Trie

- The term trie comes from the word "retrieval".
- Is pronounced, "tree."
- Tries were introduced in the 1960's by Fredkin.



Structure of a trie

- A trie is a k-ary position tree.
- It is constructed from input strings, i.e. the input is a set of n strings called S_1, S_2, \dots, S_n , where each S_i consists of symbols from a finite alphabet and has a unique terminal symbol which we call \$.



Kinds of tries

- 1 Non compact tries.
- 2 Compact tries.
- 3 "PATRICIA" Tries: even more compact.
- 4 Suffix tries.
- 5 Suffix trees.

Non compact tries

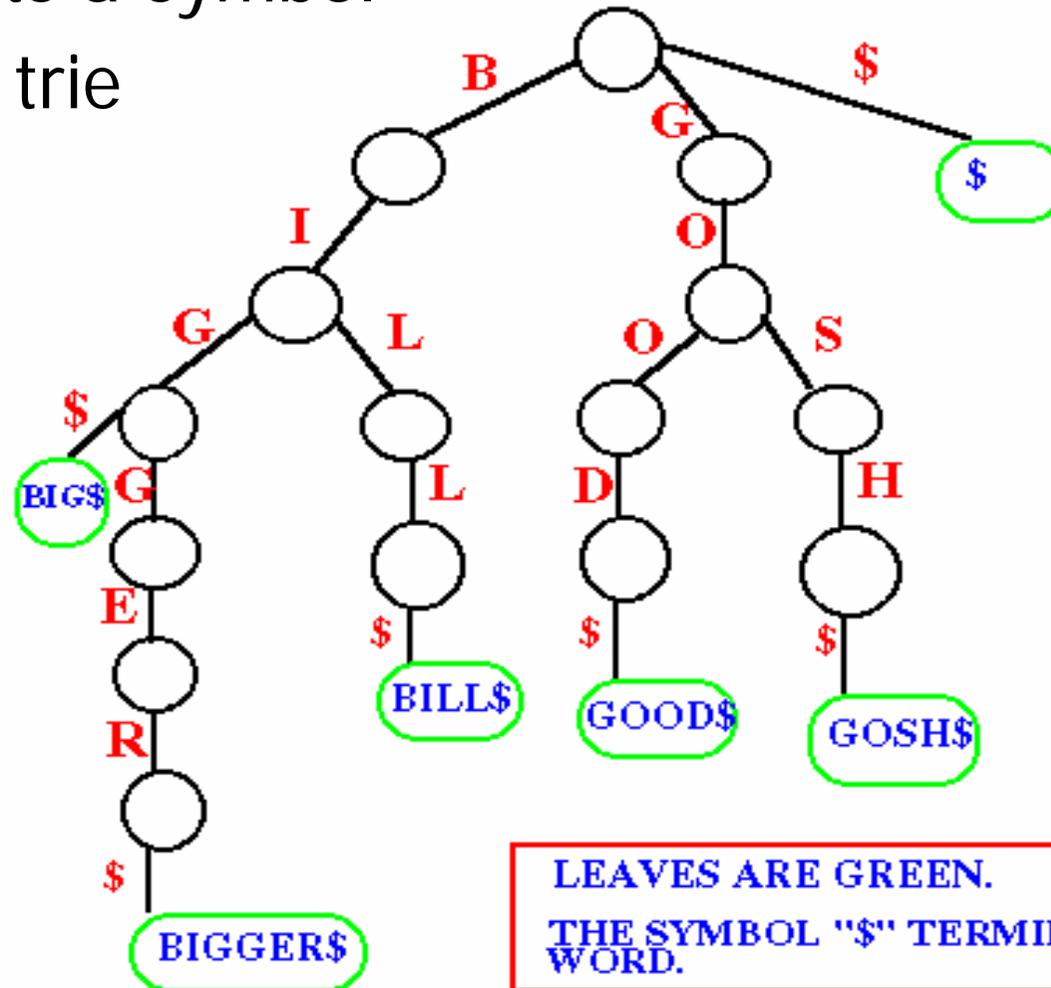
- every edge of the underlying tree represents a symbol

- construct the trie from:
the following 5 strings:

- BIG,
- BIGGER,
- BILL,
- GOOD,
- GOSH.

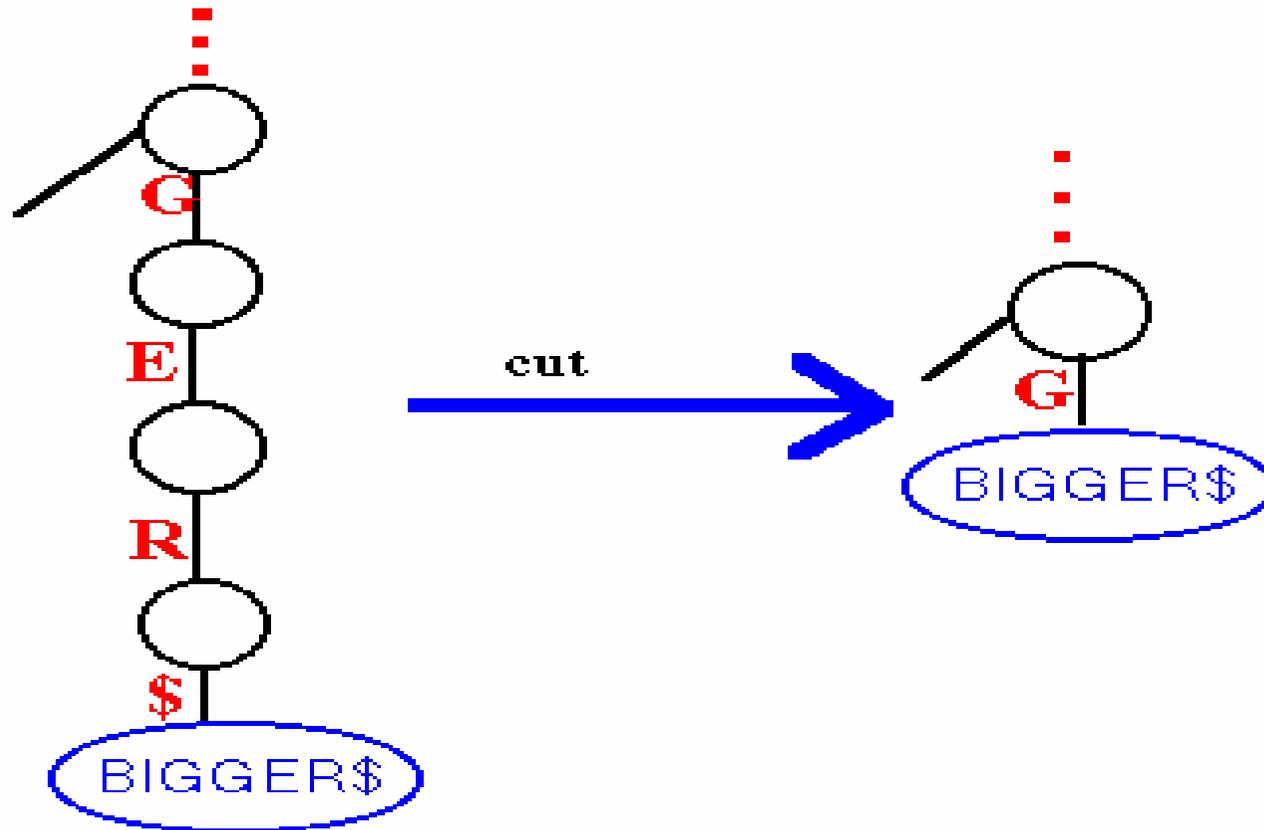
- Look for:

- GOOD
- BAD



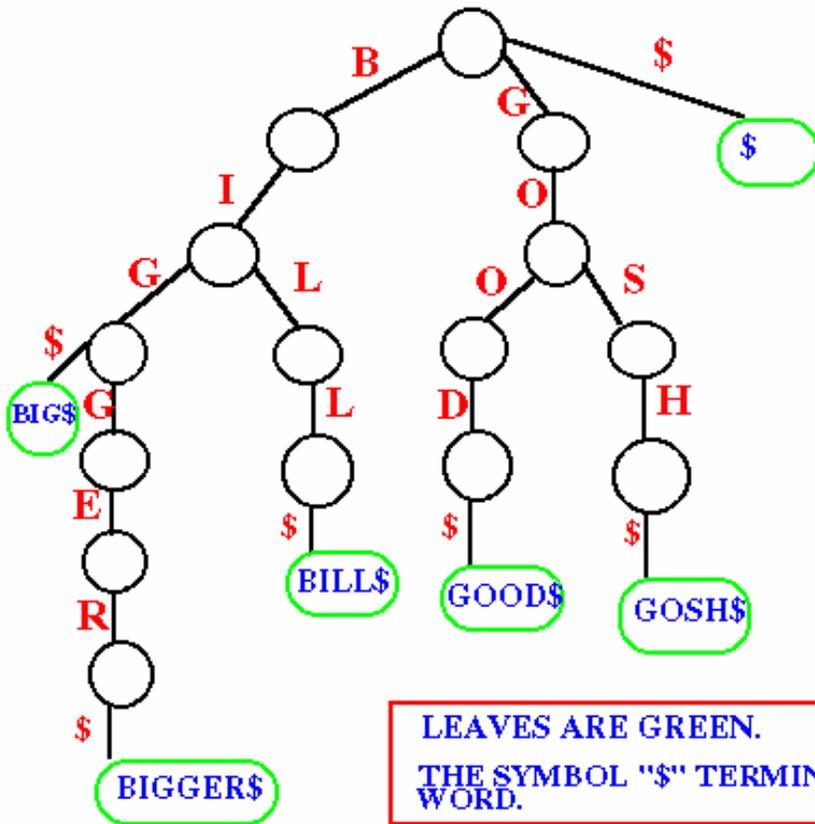
Compact tries

Trim away all chains which lead to leaves

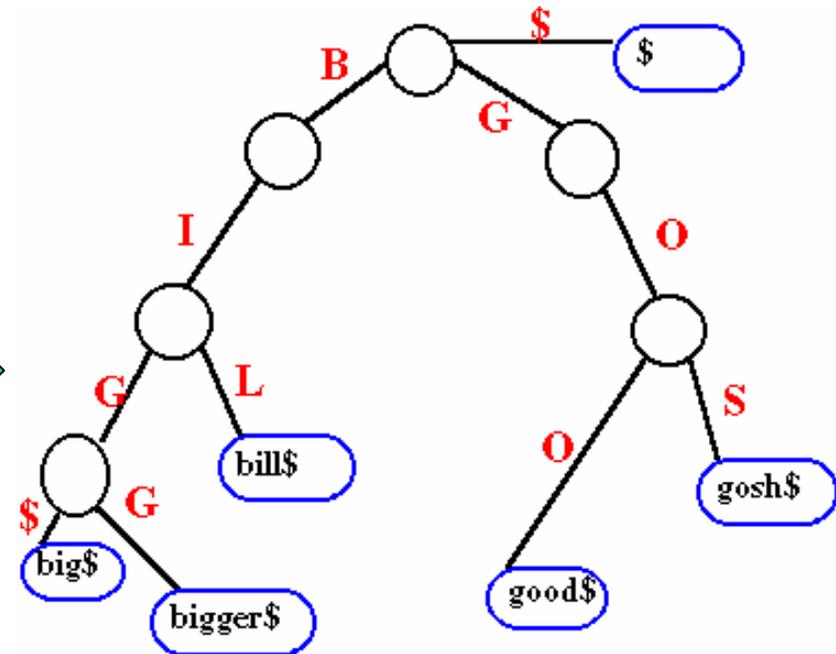


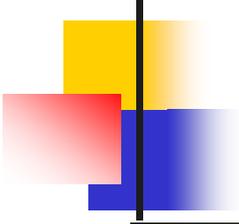
Compact tries

Non compact trie



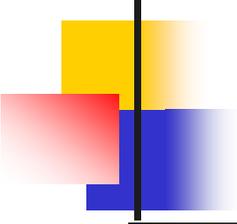
compact trie





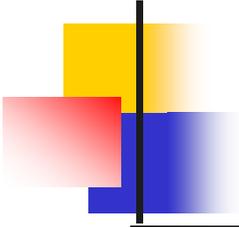
Overview

- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- String Indexing
 - **Inverted Files**
 - Suffix Tree/Array
 - Signature Files



Inverted Files(I)

- Definition
 - A word-oriented mechanism for indexing a text collection in order to speed up the searching task.
- Two elements
 - Vocabulary
 - The set of all different words in the text.
 - Occurrence
 - For each word a list of all the text positions where the word appears.



Inverted Files(II)

- A sample text and an inverted index built on it

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters

text

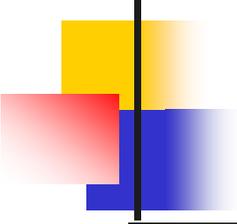
Vocabulary

letters
made
many
text
words

Occurrence

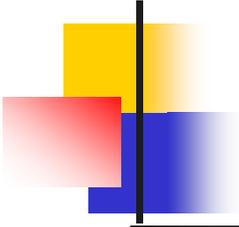
60...
50...
28...
11, 19...
33, 40...

inverted index



Inverted Files(III)

- Required space
 - The space required for the vocabulary is rather small.
 - The occurrences demand much more space.
 - Block addressing
 - reduces space requirements.
 - The text is divided in blocks, and the occurrences point to the blocks where the word appears (instead of the exact position).
 - Block division
 - The division into blocks of fixed size improves efficiency at retrieval time.
 - The division using natural cuts (files, documents, web pages) may eliminate the need for online traversal.



Inverted Files(IV)

- The sample text split into four blocks

block 1	block 2	block3	block 4
This is a text.	A text has many	words. Words are	made from letters

text

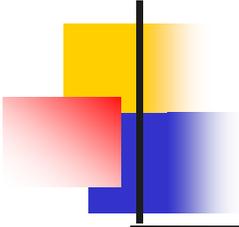
Vocabulary

letters
made
many
text
words

Occurrence

4...
4...
2...
1, 2...
3...

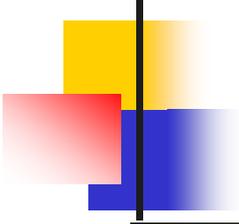
inverted index



Inverted Files(V)

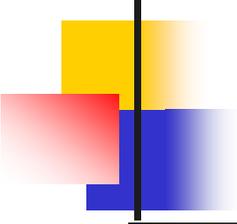
- Sizes of an inverted file:

Index	Small collection (1 Mb)		Medium collection (200 Mb)		Large collection (2 Gb)	
Addressing words	45%	73%	36%	64%	35%	63%
Addressing documents	19%	26%	18%	32%	26%	47%
Addressing 64K blocks	27%	41%	18%	32%	5%	9%
Addressing 256 blocks	18%	25%	1.7%	2.4%	0.5%	0.7%



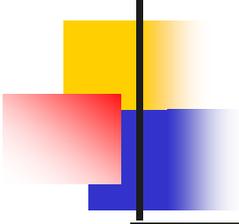
Searching(I)

- Three General search steps
 - Vocabulary search
 - The words and patterns present in the query are isolated and searched in the vocabulary.
 - Retrieval of occurrences
 - The lists of the occurrences of all the words found are retrieved.
 - Manipulation of occurrences
 - The occurrences are processed to solve phrases, proximity, or Boolean operations.
 - If block addressing is used, it may be necessary to directly search the text to find the information missing from the occurrences.



Searching(II)

- Single-word queries
 - Be searched using any suitable data structure to speed up the search, such as hashing, tries $O(m)$, or B-trees.
 - Prefix and range queries can be solved with binary search, tries, or B-trees, but not with hashing.
- Context queries
 - Each element of query must be searched separately and a list generated for each one.
 - The lists of all elements are traversed to find places where all the words appear in sequence(for a phrase) or appear close enough(for proximity).



Searching(III)

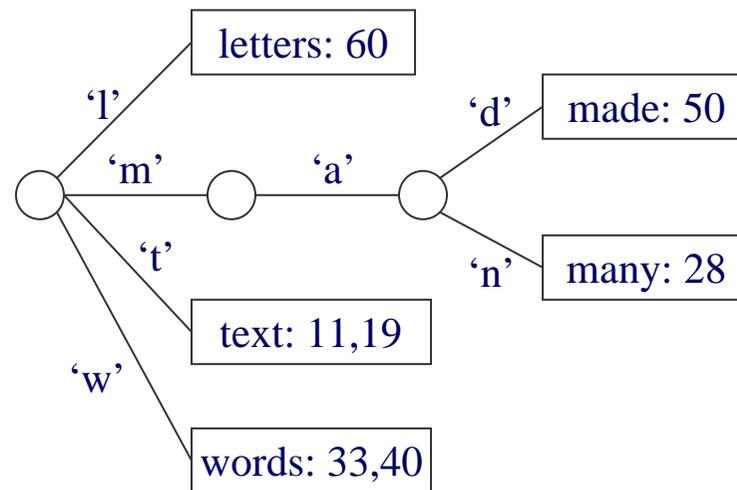
- Block addressing
 - It is necessary to traverse the blocks for these queries, since the position information is needed.
 - It is better to intersect the lists to obtain the blocks which contain all the searched words and then sequentially search the context query in those blocks.

Construction(I)

- Building an inverted index for the sample text (Fig 8.3)

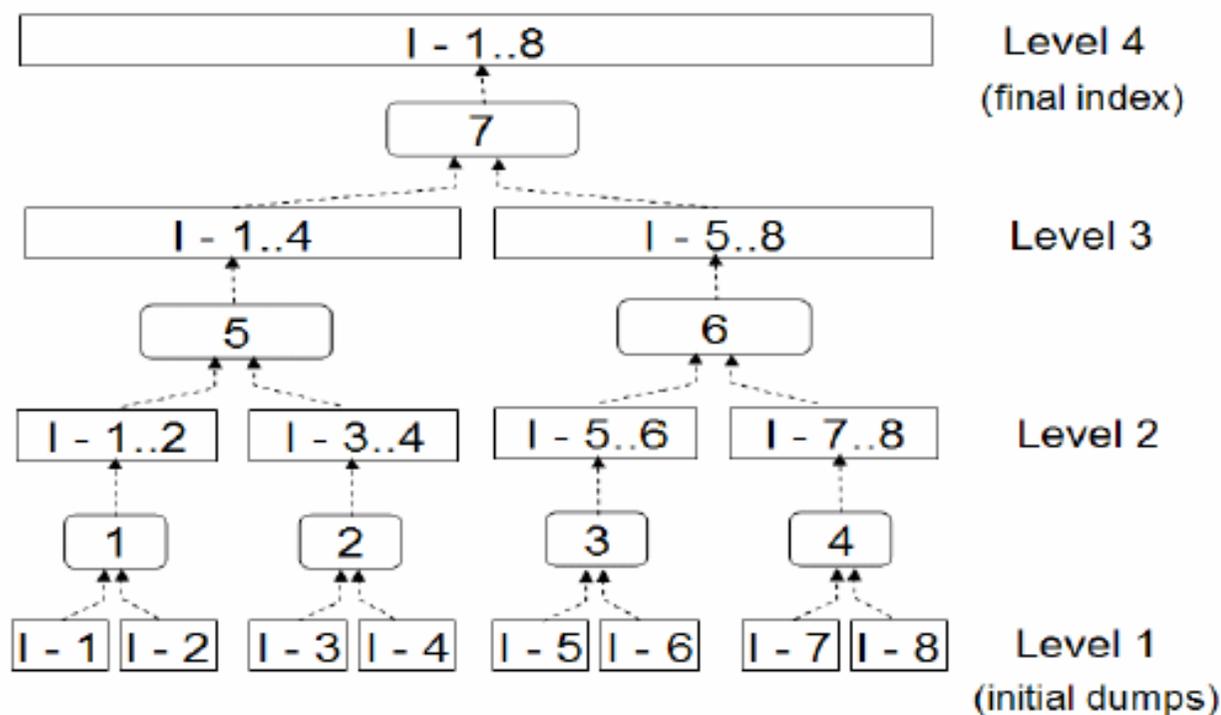
1 6 9 11 17 19 24 28 33 40 46 50 55 60

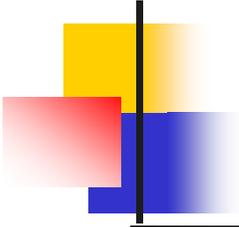
This is a text. A text has many words. Words are made from letters



Construction(II)

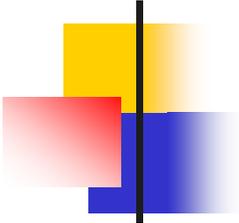
- Merging of the partial indices
 - Merge the sorted vocabularies
 - Merge both lists of occurrences if a word appears in both indices





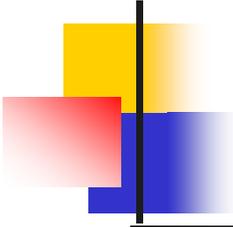
Other Indices for Text

- Suffix trees and suffix arrays
 - Suffix tree is a trie data structure built over all the suffixes of the text (a string that goes from one text position to the end of the text)
 - Suffix arrays are a space efficient implementation of suffix trees
- Signature file
 - Word-oriented index structures based on hashing
 - Low space overhead, search complexity is linear
 - Problem: false drop



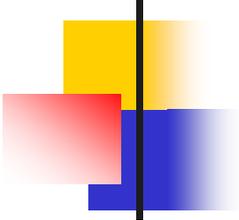
Overview

- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- String Indexing
 - Inverted Files
 - Suffix Tree/Array
 - Signature Files



Suffix Trees and Suffix Arrays

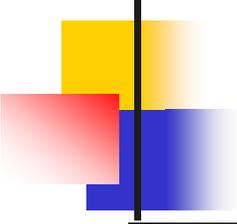
- Suffix
 - Each position in the text is considered as a text suffix.
 - A string that does from that text position to the end to the text
 - Each suffix is uniquely identified by its position
- Advantage
 - They answer efficiently more complex queries.
- Drawback
 - Costly construction process
 - The text must be readily available at query time
 - The results are not delivered in text position order, but in a lexicographical order



Suffix tree

- **Structure**

- The suffix tree is a trie structure built over all the suffixes of the text
 - Points to text are stored at the leaf nodes
- The suffix tree is implemented as a **Patricia tree** (or **PAT tree**), i.e., **a compact suffix tree**
 - Unary paths (where each node has just one child) are compressed
 - An indication of next character (or bit) position to check are stored at the internal nodes
- Each node takes 12 to 24 bytes
- A space overhead of 120%~240% over the text size

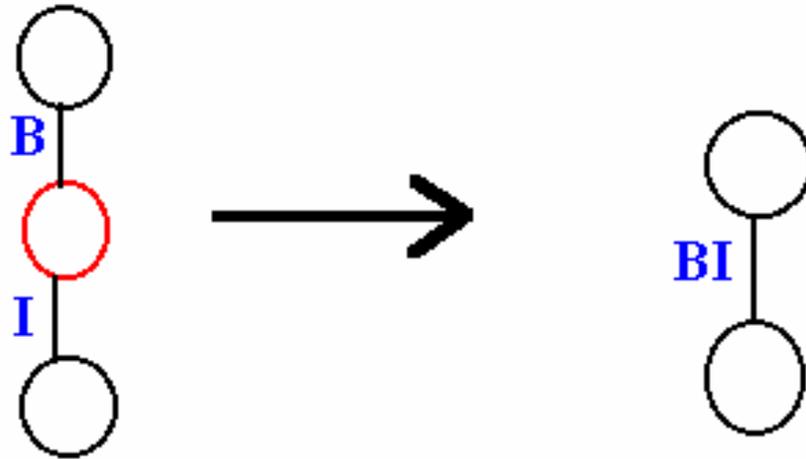


Tries called "PATRICIA"

- The compact trie can be even more compacted.
- Tries called "PATRICIA"
 - "PATRICIA" stands for "practical algorithm to retrieve information coded in alphanumeric".
 - different from the previous trie, an edge can be labeled with more than one character.
 - Hence, all the unary nodes will be collapsed.

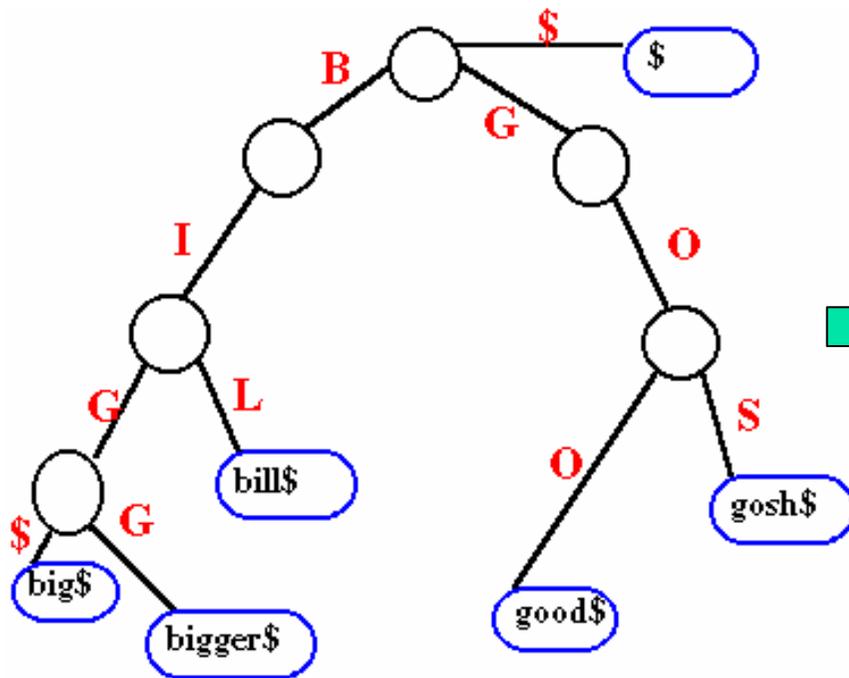
Tries called "PATRICIA"

- Collapsing process

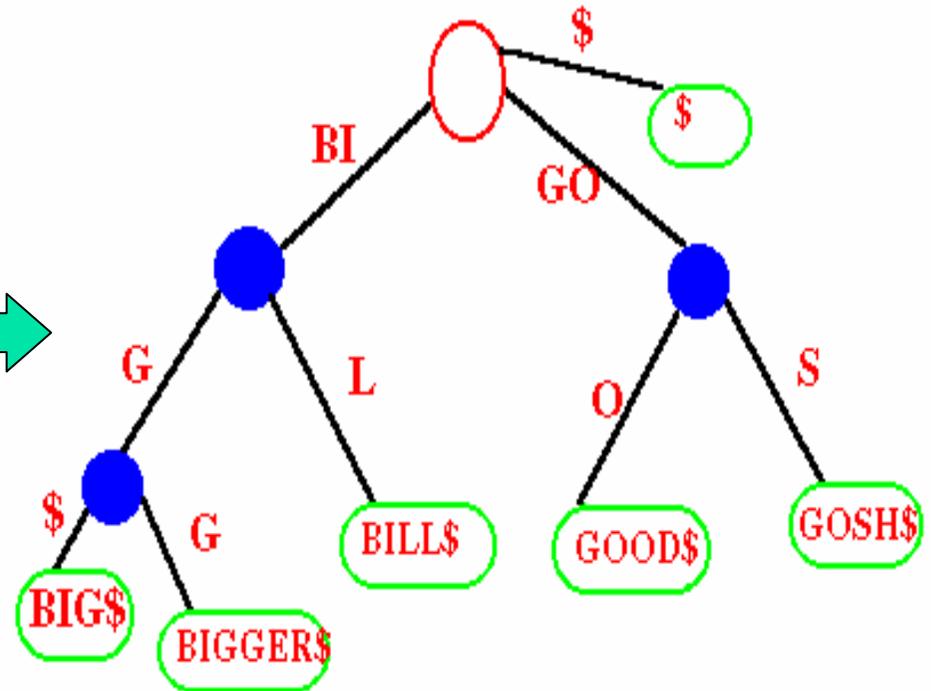


Tries called "PATRICIA"

compact trie

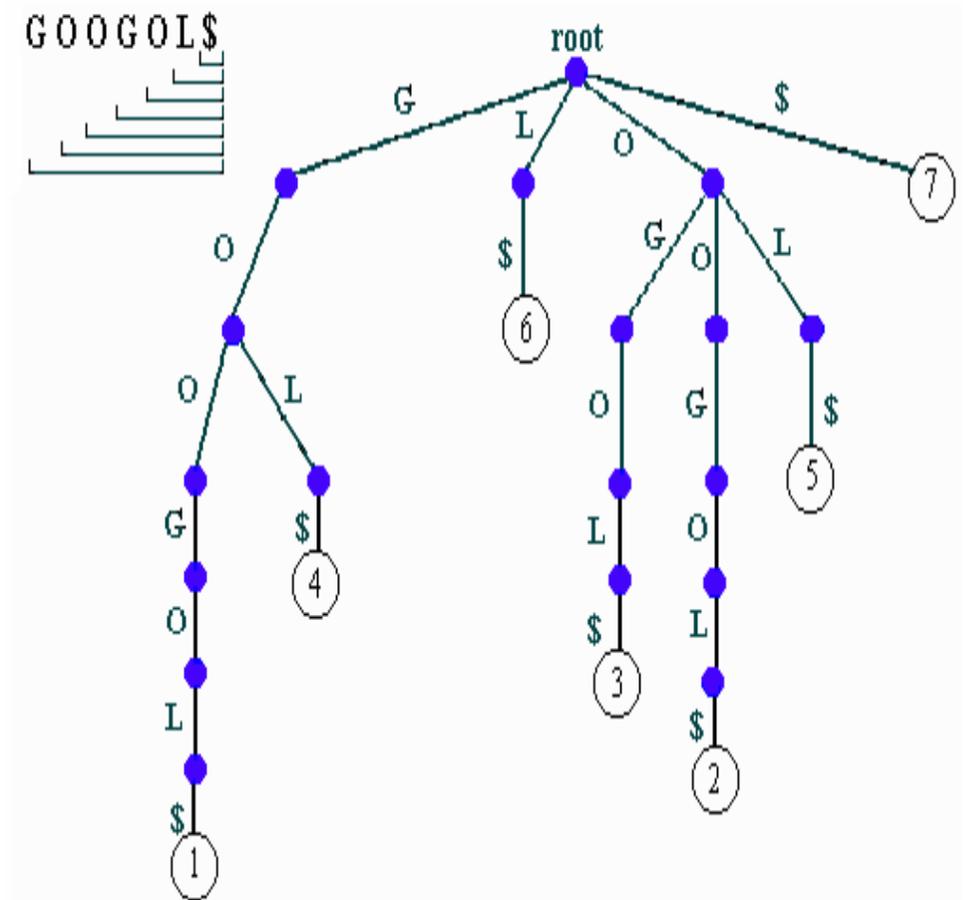


PATRICIA trie



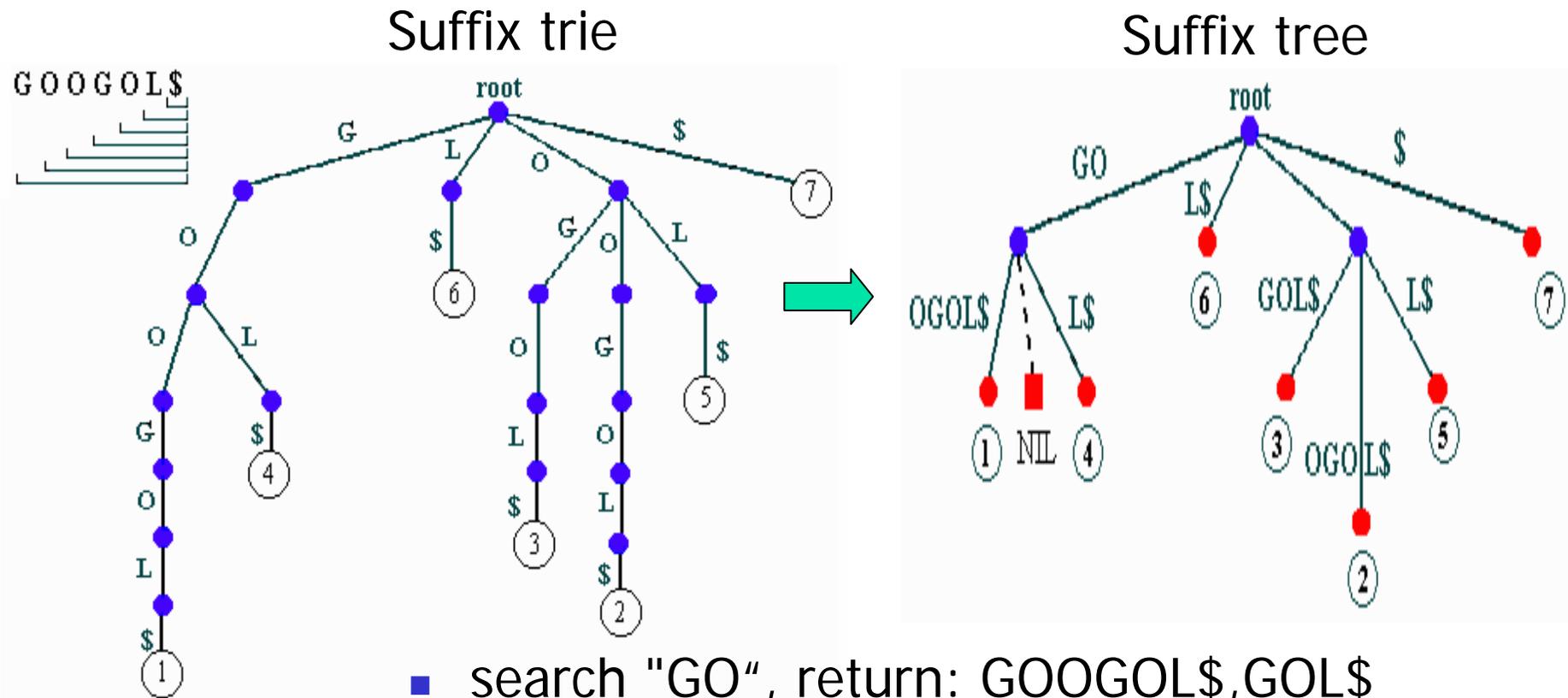
Suffix trie

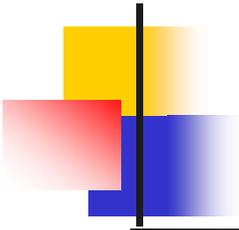
- The idea behind suffix TRIE is:
 - assign to each symbol in a text an index corresponding to its position in the text.
 - For example:
TEXT: G O O G O L \$
POSITION: 1 2 3 4 5 6 7
 - To build the suffix TRIE we use these indices instead of the actual object.



Suffix tree

- The suffix tree is created by TRIMMING (compacting + collapsing every unary node) of the suffix TRIE



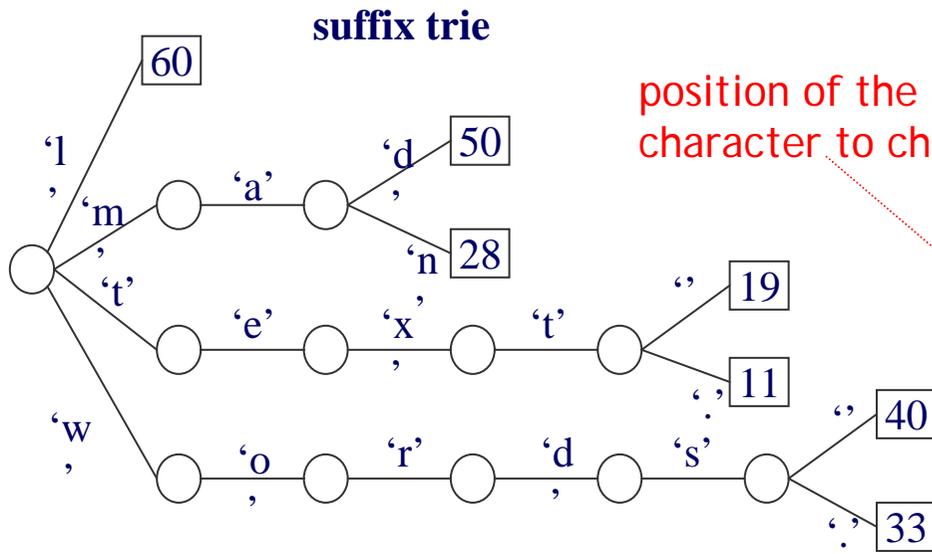


Suffix tree

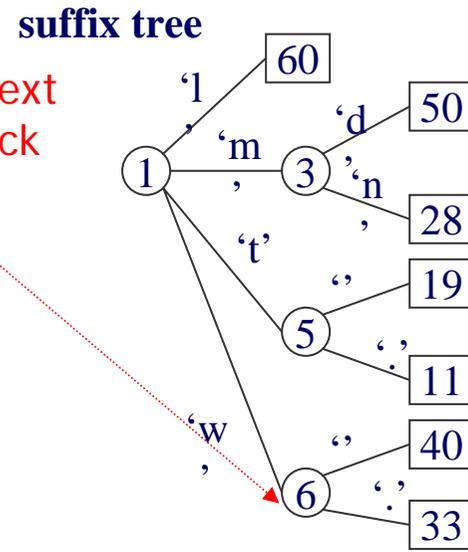
- The suffix trie and suffix tree for the sample text

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters



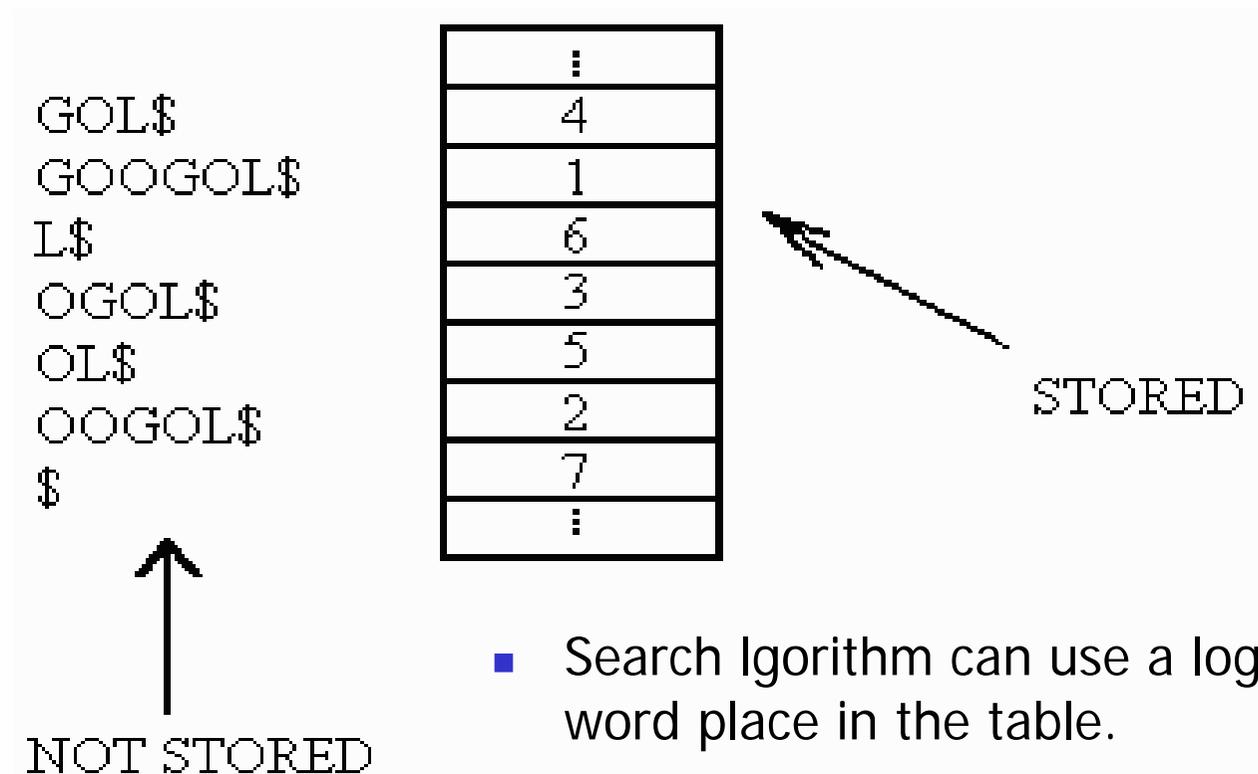
position of the next character to check



one pointer stored for each indexed suffix

Suffix array

Sort the suffixes & store in a table all the indices.



- Search algorithm can use a $\log_2 n$, to find word place in the table.
- For example, if "GOOD" is in the text then it is between 4 and 1. Looking up the string corresponding to suffixes 4 and 1 in the text we see that "GOOD" is not in it.

Suffix Arrays

- **Basic Ideas**

- Provide the same functionality as suffix trees with much less space requirements
- The leaves of the suffix tree are traversed in left-to-right (or top-to-down) order, i.e. lexicographical order, to put the pointers to the suffixes in the array
 - The space requirements the same as inverted files
- Binary search performed on the array
 - Slow when array is large $O(n)$, n is the size of indices

1	6	9	11	17	19	24	28	33	40	46	50	55	60
This is a text. A text has many words. Words are made from letters.													

Suffix array

60	50	28	19	11	40	33
----	----	----	----	----	----	----

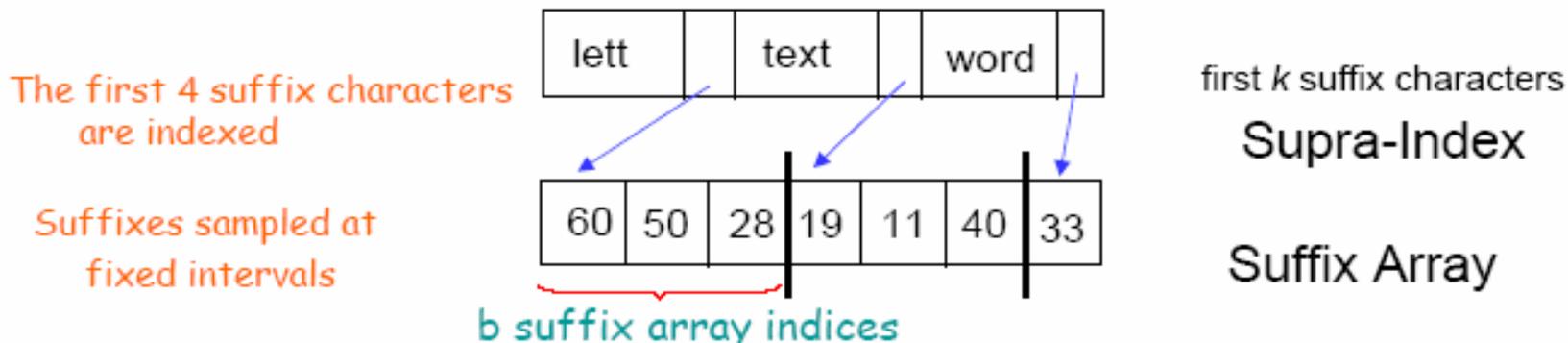
one pointer stored for each indexed suffix

(~40% overhead over the text size)

Suffix Arrays: Supra indices

- Divide the array into blocks (may with variable length) and make a sampling of each block
 - Use the **first k suffix characters**
 - Use the **first word of suffix changes** (e.g., “text” (19) in the next example for nonuniformly sampling)
- Act as a first step of search to reduce external accesses (supra indices kept in memory!)

1	6	9	11	17	19	24	28	33	40	46	50	55	60
This is a text. A text has many words. Words are made from letters.													



Suffix Arrays: Supra indices

- Compare word (vocabulary) supra-index with inverted list

1 6 9 11 17 19 24 28 33 40 46 50 55 60
This is a **text**. A **text** has **many** **words**. **Words** are **made** from **letters**.

letter	made	many	text	word
--------	------	------	------	------

Vocabulary
Supra-Index

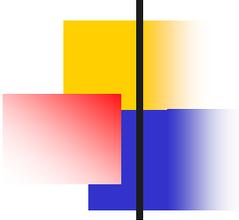
60	50	28	19	11	40	33
----	----	----	----	----	----	----

Suffix Array

60	50	28	11	19	33	40
----	----	----	----	----	----	----

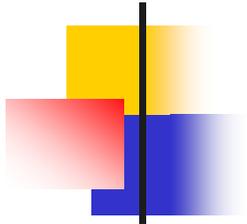
Inverted List

- major difference
- Word occurrences in suffix array are sorted lexicographically
 - Word occurrences in inverted list are sorted by text positions



Overview

- Introduction
- Spatial Indexing
 - R-Tree
 - Finding K-nearest Neighbors
 - The Reverse Nearest Neighbors
- String Indexing
 - Inverted Files
 - Suffix Tree/Array
 - Signature Files

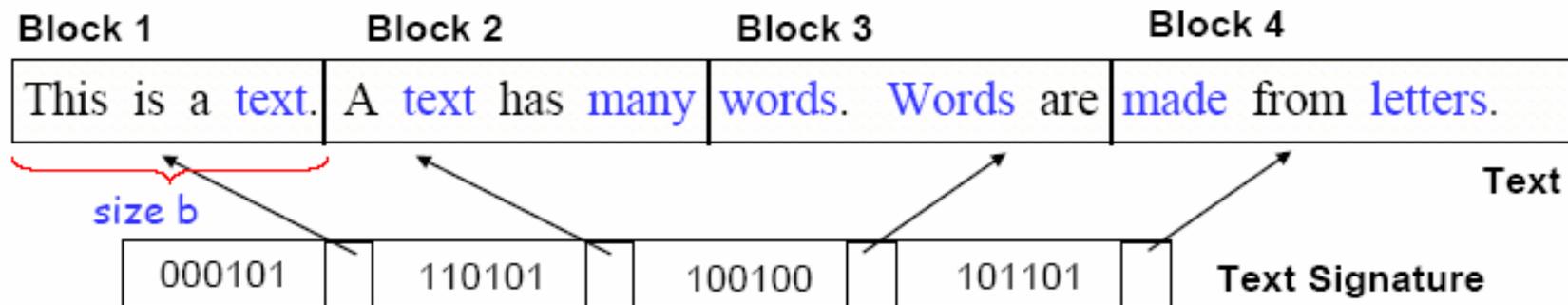


Signature files(I)

- **Basic Ideas**

- **Word-oriented index structures based on hashing**
 - A hash function (signature) maps words to bit masks of B bits
- Divide the text into **blocks of b words** each
 - **A bit mask of B bits** is assigned to each block by bitwise **ORing** the signatures of all the words in the text block
- A word is presented in a text block if all bits set in its signature are also set in the bit mask of the text block

Signature files(II)

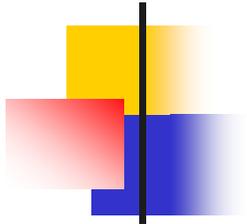


Signature functions	
$h(\text{text})$	= 000101
$h(\text{many})$	= 110000
$h(\text{words})$	= 100100
$h(\text{made})$	= 001100
$h(\text{letters})$	= 100001

size B

Stop word list
this
is
a
has
are
from
.....

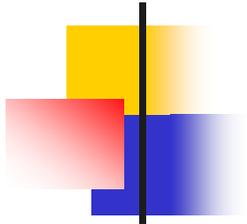
- The text signature contains
 - Sequences of bit masks
 - Pointers to blocks



Signature files(III)

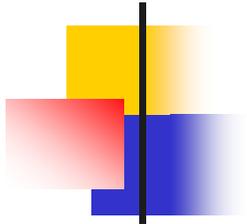
- **False Drops or False Alarms**
 - All the corresponding bits are set in the bit mask of a text block, but the query word is not there
 - E.g., a false drop for the index “letters” in block 2
- **Goals of the design of signature files**
 - Ensure the probability of a false drop is low enough
 - Keep the signature file as short as possible

tradeoff



Signature files(IV)

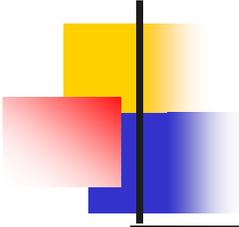
- **Single word queries**
 - Hash each word to a bit mask W
 - Compare the bit mask B_i of all text block (linear search) if they contain the word ($W \& B_i == W$?)
 - **Overhead:** online traverse candidate blocks to verify if the word is actually there
- **Phrase or Proximity queries**
 - The bitwise OR of all the query (word) masks is searched
 - The candidate blocks should have the same bits presented “1” as that in the composite query mask
 - Block boundaries should be taken care of
 - For phrases/proximities across two blocks



Signature files(V)

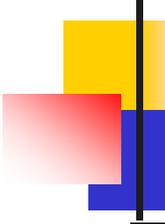
- **Construction**

- Text is cut in blocks, and for each block an entry of the signature file is generated
 - Bitwise OR of the signatures of all the words in it
- Adding text and deleting text are easy



Essential Reading

- A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching." SIGMOD Conference 1984: 47-57
- Nick Roussopoulos, Steve Kelley, and F. Vincent. "Nearest Neighbor Queries". Proc. of ACM-SIGMOD, pages 71--79, May 1995.
- [Baez+99] R. Baeza-Yates, et al, **Modern Information Retrieval** (Acm Press Series), 1999, Chapter 8.1-8.3.



Additional Reading

- F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2000.