CS618: Indexing and Searching Techniques in Databases
Memory-Based Index Structures

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs618/

2nd semester, 2014-15

Mon 1200-1315, Tue 0900-1015 at CS101

Index structures

Index structures

- Data structures (with their associated algorithms) that organize database objects for efficient searching and retrieval
- Desirable properties

Index structures

- Data structures (with their associated algorithms) that organize database objects for efficient searching and retrieval
- Desirable properties
  1. *Efficient access*: must be faster than linear scan of the entire database

Index structures

- Data structures (with their associated algorithms) that organize database objects for efficient searching and retrieval
- Desirable properties
  1. *Efficient access*: must be faster than linear scan of the entire database
  2. *Small update overhead*: if used for dynamic databases

Index structures

- Data structures (with their associated algorithms) that organize database objects for efficient searching and retrieval
- Desirable properties
  1. *Efficient access*: must be faster than linear scan of the entire database
  2. *Small update overhead*: if used for dynamic databases
  3. *Small size*: preferably, should fit in memory

Index structures

- Data structures (with their associated algorithms) that organize database objects for efficient searching and retrieval
- Desirable properties
  1. *Efficient access*: must be faster than linear scan of the entire database
  2. *Small update overhead*: if used for dynamic databases
  3. *Small size*: preferably, should fit in memory
  4. *Correctness*: answers without indexing should be the same
- Five important questions when deciding on an index structure:
  1. Type of data
     - Points or shapes or intervals
     - Discrete or continuous
     - Dimensionality and/or distance
  2. Queries on data
  3. Index data or index space where data resides
  4. Static or dynamic
  5. Memory (small) or disk-resident (large)

Classification of index structures

- Storage medium of index structure
  - Memory-based
    - Entire data fits in memory
    - No I/O cost
    - CPU efficiency
    - Generally, for low dimensions
  - Disk-based
    - Tries to reduce random (and sequential) I/O cost
    - Ignores CPU costs
    - Structure can depend on disk page size

Classification of index structures (contd.)

- Partitioning scheme for hierarchical structures
  - Space-partitioning
    - Divides *entire* space among children
    - Problem of *dead space* indexing
    - Fanout generally independent of dimensionality
    - No guarantee on space usage
  - Data-partitioning
    - Amount of data handled by each child is typically balanced
    - Eliminates *dead space* indexing
    - Fanout decreases with dimensionality
    - Guarantee on space usage

Classification of index structures (contd.)

- Partitioning scheme for hierarchical structures
  - Space-partitioning
    - Divides *entire* space among children
    - Problem of *dead space* indexing
    - Fanout generally independent of dimensionality
    - No guarantee on space usage
  - Data-partitioning
    - Amount of data handled by each child is typically balanced
    - Eliminates *dead space* indexing
    - Fanout decreases with dimensionality
    - Guarantee on space usage
- Type of data handled
  - Point access methods (PAM)
    - Only points are indexed
  - Spatial access methods (SAM)
    - Spatial objects (such as polygons) can be indexed

Binary search tree (BST)

- Binary tree
- Unbalanced
- If a node has key v, then every key l in left subtree and every key r in right subtree follows the properties:
  1. $l \leq v$
  2. $r > v$

Binary search tree (BST)

- Binary tree
- Unbalanced
- If a node has key v, then every key l in left subtree and every key r in right subtree follows the properties:
  1. $l \leq v$
  2. $r > v$
- Searching, insertion, deletion takes $O(\lg n)$ time on *average*
- Worst-case times are $O(n)$ due to unbalanced property

Binary search tree (BST)

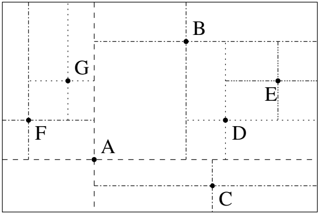
- Binary tree
- Unbalanced
- If a node has key v, then every key l in left subtree and every key r in right subtree follows the properties:
  1. $l \leq v$
  2. $r > v$
- Searching, insertion, deletion takes $O(\lg n)$ time on *average*
- Worst-case times are $O(n)$ due to unbalanced property
- Two important balanced structures are
  - AVL trees
  - Red-black trees

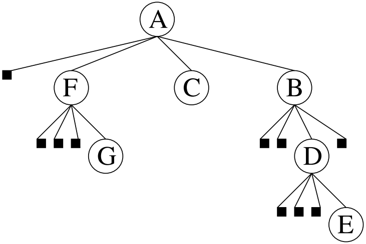
Binary search tree (BST)

- Binary tree
- Unbalanced
- If a node has key v, then every key l in left subtree and every key r in right subtree follows the properties:
    1. $l \leq v$
    2. $r > v$
- Searching, insertion, deletion takes $O(\lg n)$ time on *average*
- Worst-case times are $O(n)$ due to unbalanced property
- Two important balanced structures are
    - AVL trees
    - Red-black trees
- Not perfectly balanced but within $O(\lg n)$ bound
- Search is faster in AVL trees due to lesser height
- Insertion, etc. are slower due to more rotations

Quadtree

- Two-dimensional *unbalanced* binary search tree
- Four children corresponding to 4 quadrants
- If key in node is (x, y), then 4 partitions
  1. $(\leq x, \leq y)$
  2. $(\leq x, > y)$
  3. $(> x, \leq y)$
  4. $(> x, > y)$
- Insertion is simple but order-dependent
- These are point quadtrees





Searching in quadtrees

- Point search is simple
- Range search requires
  1. Minimum bounding rectangle computation
  2. Overlapped rectangles computation
- Average time is $O(\log n)$
- Time can be $O(\sqrt{n})$ in worst case

Searching in quadtrees

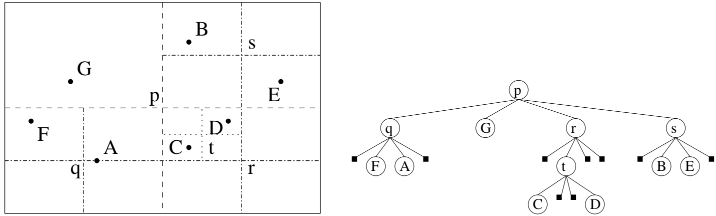
- Point search is simple
- Range search requires
  1. Minimum bounding rectangle computation
  2. Overlapped rectangles computation
- Average time is $O(\log n)$
- Time can be $O(\sqrt{n})$ in worst case
- Can be extended to d dimensions
- Fanout becomes

Searching in quadtrees

- Point search is simple
- Range search requires
  1. Minimum bounding rectangle computation
  2. Overlapped rectangles computation
- Average time is $O(\log n)$
- Time can be $O(\sqrt{n})$ in worst case
- Can be extended to d dimensions
- Fanout becomes 2^d
- Worst case searching time is $O(d.n^{(1-1/d)})$
- Called octrees in three dimensions

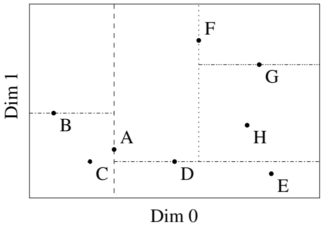
Region quadtrees

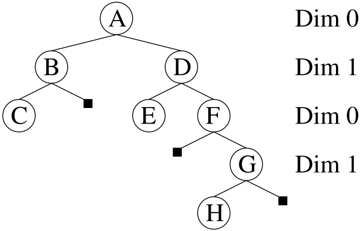
- Always divide at the geographical centre



K-d-tree

- Multi-dimensional unbalanced binary search tree
- Splits only one dimension at a time
- Fanout is constrained to be 2
- Dimensions are ordered and splits are cycled through dimensions
- If node at level i stores object v
  - Left child: $l[i \bmod d] \leq v[i \bmod d]$
  - Right child: $r[i \bmod d] > v[i \bmod d]$
- These are point K-d-trees





Discussion

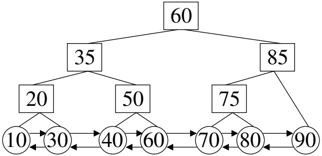
- Average searching time is $O(\lg n)$
- Worst case may become $O(d.n^{(1-1/d)})$

Discussion

- Average searching time is $O(\lg n)$
- Worst case may become $O(d.n^{(1-1/d)})$
- Region K-d-trees: splits are always done at middle of range
- For uniformly distributed data, region K-d-trees are more balanced than point K-d-trees

Range tree

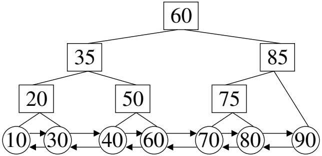
- Designed to support *window queries* more efficiently
- In 1-dimension, it is similar to a balanced binary search tree
- Leaves are *sorted* and *doubly linked*



- Suppose, search for range $[p : q]$

Range tree

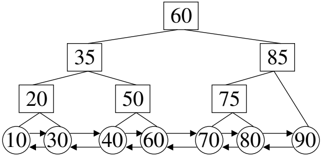
- Designed to support *window queries* more efficiently
- In 1-dimension, it is similar to a balanced binary search tree
- Leaves are *sorted* and *doubly linked*



- Suppose, search for range $[p : q]$
- First, proceeds to leaf just less than or equal to p
- Then, traverse forward pointers till leaf just greater than q
- Time taken is $O(\lg n + T)$ where T is size of the answer set

Range tree

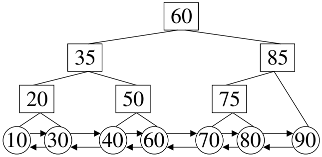
- Designed to support *window queries* more efficiently
- In 1-dimension, it is similar to a balanced binary search tree
- Leaves are *sorted* and *doubly linked*



- Suppose, search for range $[p : q]$
- First, proceeds to leaf just less than or equal to p
- Then, traverse forward pointers till leaf just greater than q
- Time taken is $O(\lg n + T)$ where T is size of the answer set
- Search for $[p : \infty]$:

Range tree

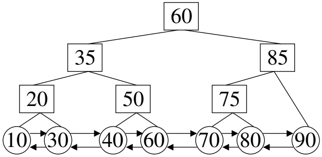
- Designed to support *window queries* more efficiently
- In 1-dimension, it is similar to a balanced binary search tree
- Leaves are *sorted* and *doubly linked*



- Suppose, search for range $[p : q]$
- First, proceeds to leaf just less than or equal to p
- Then, traverse forward pointers till leaf just greater than q
- Time taken is $O(\lg n + T)$ where T is size of the answer set
- Search for $[p : \infty]$: all forward leaves from p
- Search for $[-\infty : q]$:

Range tree

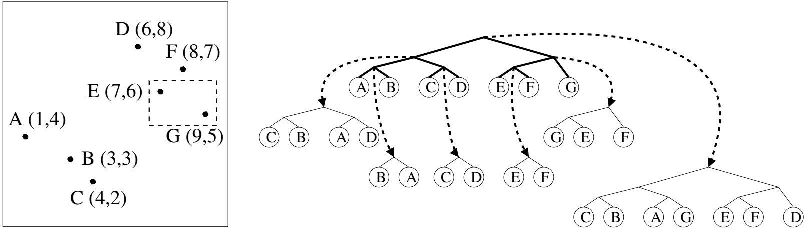
- Designed to support *window queries* more efficiently
- In 1-dimension, it is similar to a balanced binary search tree
- Leaves are *sorted* and *doubly linked*



- Suppose, search for range $[p : q]$
- First, proceeds to leaf just less than or equal to p
- Then, traverse forward pointers till leaf just greater than q
- Time taken is $O(\lg n + T)$ where T is size of the answer set
- Search for $[p : \infty]$: all forward leaves from p
- Search for $[-\infty : q]$: all backward leaves from q

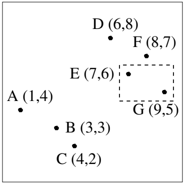
Multi-dimensional range trees

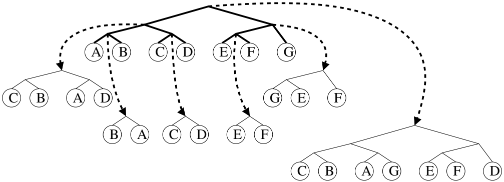
- 1-dimensional range tree of $(d-1)$-dimensional range trees
- Each node is another range tree catering to a set of points in dimensionality one less
- Order dimensions as $x_0, x_1, \ldots, x_{d-1}$
- Main range tree or "base" tree is built on dimension x_0
- For each node T_{x_0} of this base tree, a range tree R_{x_1} of dimensionality $(d-1)$ is associated
- Range tree R_{x_1} contains all points in subtree T_{x_0}, but the points are now organized according to dimension x_1



Multi-dimensional range tree searching

- Search is first through x_0
- Leaves L and R covering range in x_0 are identified
- Least common ancestor Q of L and R is determined
- Range search for dimensions x_1 to x_{d-1} is issued on the range tree linked from Q

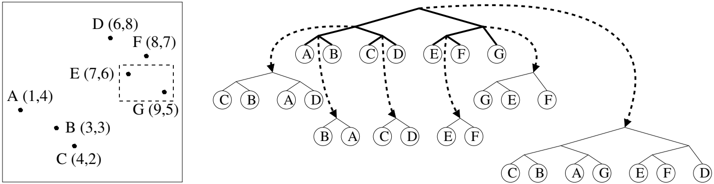




- Example: $[6.5, 9.5], [4.5, 6.5]$

Multi-dimensional range tree searching

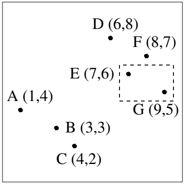
- Search is first through x_0
- Leaves L and R covering range in x_0 are identified
- Least common ancestor Q of L and R is determined
- Range search for dimensions x_1 to x_{d-1} is issued on the range tree linked from Q

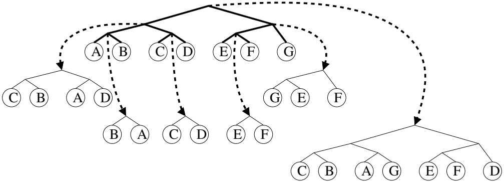


- Example: $[6.5, 9.5], [4.5, 6.5]$
- Leaves E, F and G in x dimension
- Tree corresponding to this in y is searched
- In y dimension, only G and E are returned

Multi-dimensional range tree searching

- Search is first through x_0
- Leaves L and R covering range in x_0 are identified
- Least common ancestor Q of L and R is determined
- Range search for dimensions x_1 to x_{d-1} is issued on the range tree linked from Q

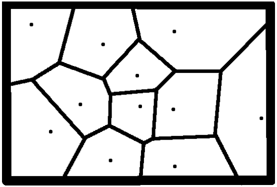




- Example: $[6.5, 9.5], [4.5, 6.5]$
- Leaves E, F and G in x dimension
- Tree corresponding to this in y is searched
- In y dimension, only G and E are returned
- Running time is $O(\log^d n + T)$

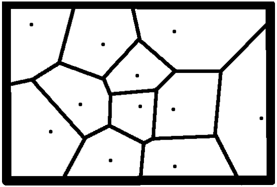
Voronoi diagrams

- Partitioning of a plane with n points (called sites) into *convex polygons* such that
  - Each polygon contains one and only one site
  - Every point in the polygon is closest to the site that corresponds to the polygon than any other site
- Also known as Voronoi tessellation or Thiessen tessellation
- Polygons are Voronoi regions or Voronoi cells or Thiessen polygons
- In metric spaces, structure is called Dirichlet tessellation and regions are called Dirichlet domains



Voronoi diagrams

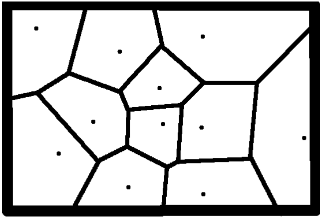
- Partitioning of a plane with *n* points (called sites) into *convex polygons* such that
  - Each polygon contains one and only one site
  - Every point in the polygon is closest to the site that corresponds to the polygon than any other site
- Also known as Voronoi tessellation or Thiessen tessellation
- Polygons are Voronoi regions or Voronoi cells or Thiessen polygons
- In metric spaces, structure is called Dirichlet tessellation and regions are called Dirichlet domains

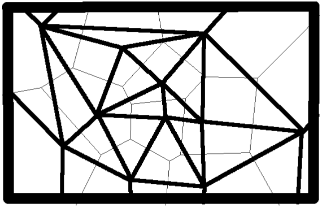


- Immediate answer to 1NN queries

Delaunay triangulation

- Dual of Voronoi diagram is Delaunay triangulation
- Triangulate such that circumcircles of every triangle is empty, i.e., it does not contain any other site
- Resulting structure resembles a graph called Delaunay graph
- Vertices of the resulting graph are sites called Voronoi vertices
- Edges are Voronoi edges





Construction

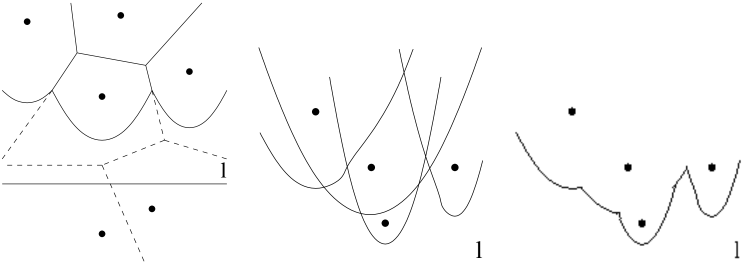
- Naïve construction requires $O(n^2)$ time
  - Draw perpendicular bisectors between every pair of sites
  - Trim the lines appropriately to form polygons

Construction

- Naïve construction requires $O(n^2)$ time
  - Draw perpendicular bisectors between every pair of sites
  - Trim the lines appropriately to form polygons
- Fortune's algorithm
  - *Plane sweep* paradigm
  - Time taken is $O(n \lg n)$
  - Optimal running time
  - Requires $O(n)$ space to store

Fortune's algorithm

- Sweep line swept from top to bottom of space
- All sites above the sweep line compute the beach line
- Beach line indicates locus of points closer to some site above the sweep line than the sweep line itself
- Beach line is actually a set of parabolic arcs
- Voronoi diagram above the beach line never changes
- So, beach line needs to be maintained correctly

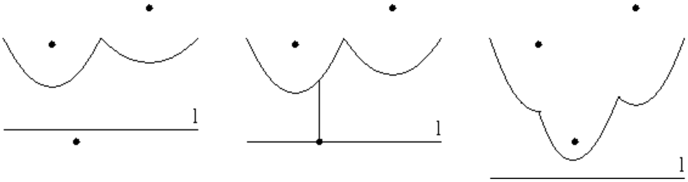


Adding a parabolic arc

- A parabolic arc gets added to the beach line

Adding a parabolic arc

- A parabolic arc gets added to the beach line only when the sweep line hits a site
- This is called a site event

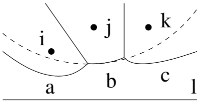


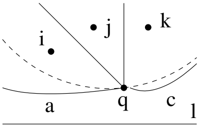
Deleting a parabolic arc

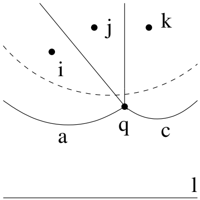
- A parabolic arc gets deleted from the beach line

Deleting a parabolic arc

- A parabolic arc gets deleted from the beach line only when its contribution shrinks to a point and it disappears
- This is called a circle event
- When arc b reduces to a point, the point q is equidistant from all the three sites i, j, and k
- A circumcircle can be defined that passes through all i, j, and k
- Circumcircle also touches the sweep line, i.e., it is tangent to it
- As soon as sweep line moves downward a little, the arc b disappears







Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance
  - Possible since it only stretches dimensions

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance
    - Possible since it only stretches dimensions
- Same for Mahalanobis distance since it amounts to stretching and rotating

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance
  - Possible since it only stretches dimensions
- Same for Mahalanobis distance since it amounts to stretching and rotating
- Manhattan distance or L_1 distance

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance
  - Possible since it only stretches dimensions
- Same for Mahalanobis distance since it amounts to stretching and rotating
- Manhattan distance or L_1 distance
  - No guarantee that such a structure exists

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance
  - Possible since it only stretches dimensions
- Same for Mahalanobis distance since it amounts to stretching and rotating
- Manhattan distance or L_1 distance
  - No guarantee that such a structure exists
- In d dimensions,

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance
  - Possible since it only stretches dimensions
- Same for Mahalanobis distance since it amounts to stretching and rotating
- Manhattan distance or L_1 distance
  - No guarantee that such a structure exists
- In d dimensions, requires $O(n^{\lceil d/2 \rceil})$ time

Discussion

- At most $O(n)$ circle events and site events
- Processing each event requires $O(\lg n)$ time (since the sites are sorted)
- Therefore, total time is $O(n \lg n)$
- Weighted Euclidean distance
  - Possible since it only stretches dimensions
- Same for Mahalanobis distance since it amounts to stretching and rotating
- Manhattan distance or L_1 distance
  - No guarantee that such a structure exists
- In d dimensions, requires $O(n^{\lceil d/2 \rceil})$ time
- Voronoi diagram of order-k
  - Partition according to k closest sites
  - Useful for kNN searches
  - Order-$(n-1)$ is called farthest point Voronoi diagram
  - In two dimensions, requires $O(k(n-k))$ space and $O(n \lg n + k^2(n-k))$ time

Tries

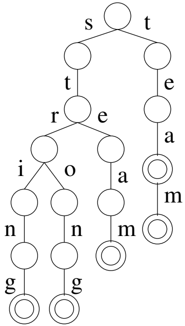
- Comes from the word retrieval
- Mostly used for strings
- Structure of a *basic* trie
  - Root represents null string
  - Each edge defines the next character
  - Each node stores a string or a prefix of a string
  - Strings with same prefix share the path
- Advantages over binary search trees
  - Search time is $O(m)$ where m is the length of the query
  - Generally, $m \ll \lg n$
- Also called a prefix tree

Example

- Trie for strings "steam", "string", "strong", "tea", and "team"

Example

- Trie for strings "steam", "string", "strong", "tea", and "team"

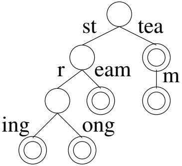


Compressed trie

- Compresses all non-terminal unary nodes

Compressed trie

- Compresses all non-terminal unary nodes

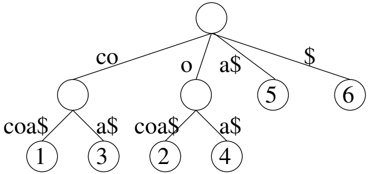


Suffix trees

- For efficient *substring* searching
- It encodes all the suffixes of a string
- Every edge represents the next character(s) from a suffix
- Every leaf represents a suffix of the string
- A compressed trie of all the suffixes
- Suffix tree for the string “cocoa” is

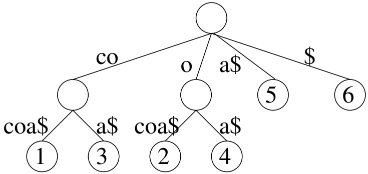
Suffix trees

- For efficient *substring* searching
- It encodes all the suffixes of a string
- Every edge represents the next character(s) from a suffix
- Every leaf represents a suffix of the string
- A compressed trie of all the suffixes
- Suffix tree for the string "cocoa" is



Suffix trees

- For efficient *substring* searching
- It encodes all the suffixes of a string
- Every edge represents the next character(s) from a suffix
- Every leaf represents a suffix of the string
- A compressed trie of all the suffixes
- Suffix tree for the string "cocoa" is



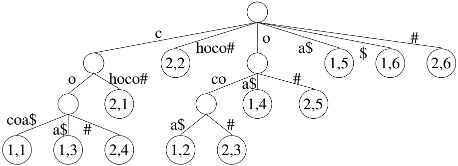
- A special end-of-string symbol ($) is assumed
- This ensures that no suffix is a prefix of another suffix

Generalized suffix trees

- A generalized suffix tree is built from multiple strings

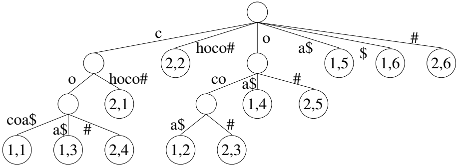
Generalized suffix trees

- A generalized suffix tree is built from multiple strings



Generalized suffix trees

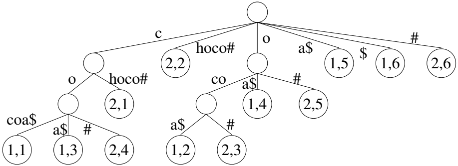
- A generalized suffix tree is built from multiple strings



- Different end-of-string symbols are used
- Suppose, “co” is searched

Generalized suffix trees

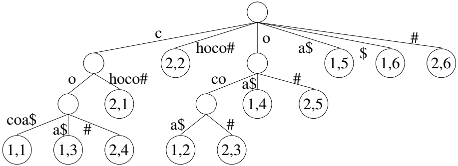
- A generalized suffix tree is built from multiple strings



- Different end-of-string symbols are used
- Suppose, "co" is searched
- First two leftmost pointers are traversed
- Every leaf under this subtree ("cocoa$", "coa$", "co#") returns an answer

Generalized suffix trees

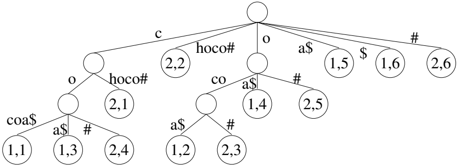
- A generalized suffix tree is built from multiple strings



- Different end-of-string symbols are used
- Suppose, “co” is searched
- First two leftmost pointers are traversed
- Every leaf under this subtree (“cocoa$”, “coa$”, “co#”) returns an answer
- Searching for “cha” will end without reaching any node

Generalized suffix trees

- A generalized suffix tree is built from multiple strings



- Different end-of-string symbols are used
- Suppose, “co” is searched
- First two leftmost pointers are traversed
- Every leaf under this subtree (“cocoa$”, “coa$”, “co#”) returns an answer
- Searching for “cha” will end without reaching any node
- Naïve construction requires $O(m^2)$ time for a m-length string
- Can be brought down to $O(m)$
- For multiple strings of length $m_1, m_2, \ldots$, can be done in $O(m_1 + m_2 + \ldots)$ time

Ukkonen's algorithm

- Considers substrings from the beginning
- *Online* in nature
- For length up to k, builds *implicit suffix trees* correct up to the k-length prefix

Ukkonen's algorithm

- Considers substrings from the beginning
- *Online* in nature
- For length up to k, builds *implicit suffix trees* correct up to the k-length prefix
- Runs in *phases* 1 to $m - 1$
- Each phase i has *extensions* 1 to $i + 1$

Ukkonen's algorithm

- Considers substrings from the beginning
- *Online* in nature
- For length up to k, builds *implicit suffix trees* correct up to the k-length prefix
- Runs in *phases* 1 to $m-1$
- Each phase i has *extensions* 1 to $i+1$
- Before phase i and extension j starts, it is assumed that the substring $S[j \dots i]$ is in the implicit suffix tree
- It is then extended by the symbol $S[i+1]$ such that the substring $S[j \dots (i+1)]$ is now in the tree

Ukkonen's algorithm

- Considers substrings from the beginning
- *Online* in nature
- For length up to k, builds *implicit suffix trees* correct up to the k-length prefix
- Runs in *phases* 1 to $m - 1$
- Each phase i has *extensions* 1 to $i + 1$
- Before phase i and extension j starts, it is assumed that the substring $S[j \dots i]$ is in the implicit suffix tree
- It is then extended by the symbol $S[i + 1]$ such that the substring $S[j \dots (i + 1)]$ is now in the tree
- Extension in three ways
  1. Path for $S[j \dots i]$ ends in a leaf:

Ukkonen's algorithm

- Considers substrings from the beginning
- *Online* in nature
- For length up to k, builds *implicit suffix trees* correct up to the k-length prefix
- Runs in *phases* 1 to $m-1$
- Each phase i has *extensions* 1 to $i+1$
- Before phase i and extension j starts, it is assumed that the substring $S[j \dots i]$ is in the implicit suffix tree
- It is then extended by the symbol $S[i+1]$ such that the substring $S[j \dots (i+1)]$ is now in the tree
- Extension in three ways
  1. Path for $S[j \dots i]$ ends in a leaf: $S[i+1]$ is added to the path
  2. Path for $S[j \dots i]$ ends in an internal node and there is no path from that node starting with $S[i+1]$:

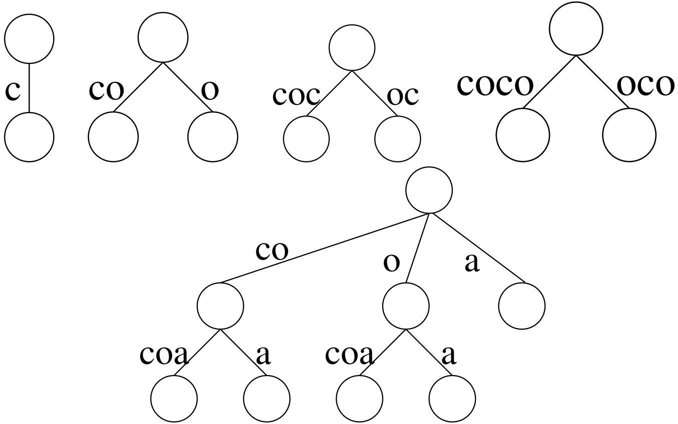
Ukkonen's algorithm

- Considers substrings from the beginning
- *Online* in nature
- For length up to k, builds *implicit suffix trees* correct up to the k-length prefix
- Runs in *phases* 1 to $m - 1$
- Each phase i has *extensions* 1 to $i + 1$
- Before phase i and extension j starts, it is assumed that the substring $S[j \dots i]$ is in the implicit suffix tree
- It is then extended by the symbol $S[i + 1]$ such that the substring $S[j \dots (i + 1)]$ is now in the tree
- Extension in three ways
  1. Path for $S[j \dots i]$ ends in a leaf: $S[i + 1]$ is added to the path
  2. Path for $S[j \dots i]$ ends in an internal node and there is no path from that node starting with $S[i + 1]$: $S[i + 1]$ is added as a new path from the node
  3. Path for $S[j \dots i]$ ends in an internal node and there is a path from that node starting with $S[i + 1]$:

Ukkonen's algorithm

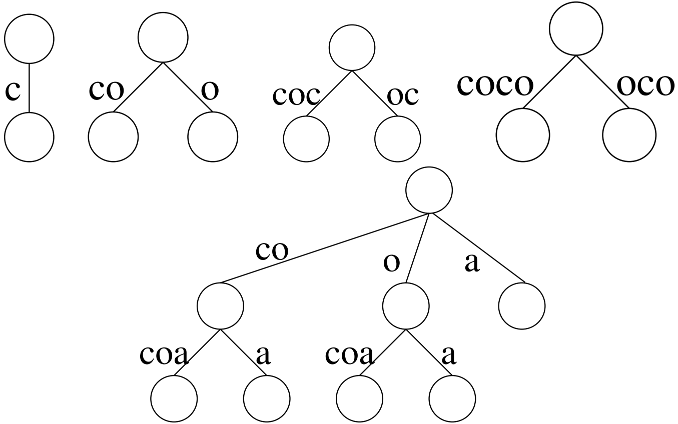
- Considers substrings from the beginning
- *Online* in nature
- For length up to k, builds *implicit suffix trees* correct up to the k-length prefix
- Runs in *phases* 1 to $m - 1$
- Each phase i has *extensions* 1 to $i + 1$
- Before phase i and extension j starts, it is assumed that the substring $S[j \ldots i]$ is in the implicit suffix tree
- It is then extended by the symbol $S[i + 1]$ such that the substring $S[j \ldots (i + 1)]$ is now in the tree
- Extension in three ways
  1. Path for $S[j \ldots i]$ ends in a leaf: $S[i + 1]$ is added to the path
  2. Path for $S[j \ldots i]$ ends in an internal node and there is no path from that node starting with $S[i + 1]$: $S[i + 1]$ is added as a new path from the node
  3. Path for $S[j \ldots i]$ ends in an internal node and there is a path from that node starting with $S[i + 1]$: Nothing is done

Example



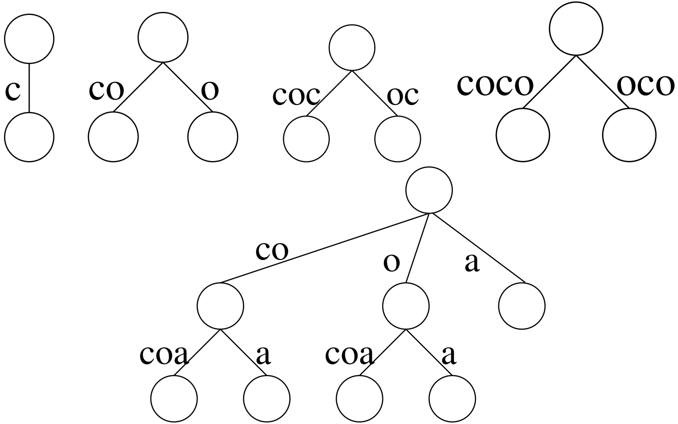
- Adding 'o' to "c":

Example



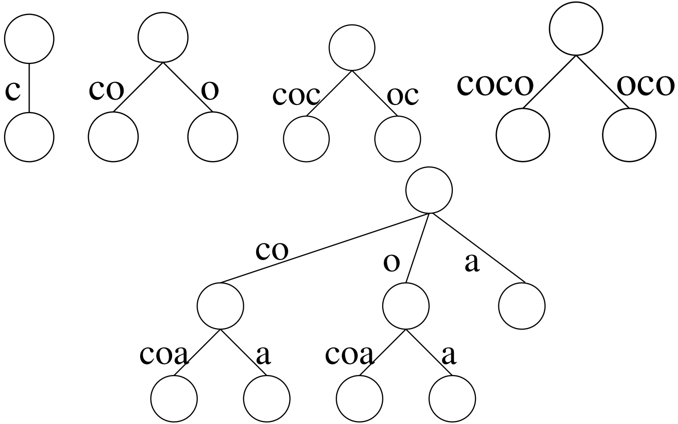
- Adding ‘o’ to “c”: first kind
- Adding ‘o’ to “”:

Example



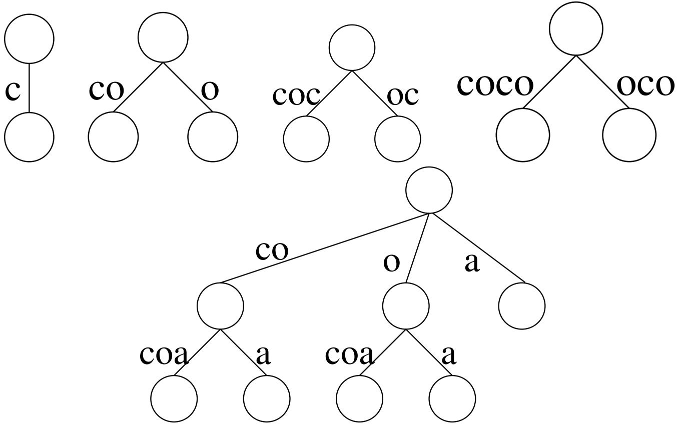
- Adding ‘o’ to “c”: first kind
- Adding ‘o’ to “”: second kind
- Adding second ‘c’:

Example



- Adding 'o' to "c": first kind
- Adding 'o' to "": second kind
- Adding second 'c': third kind

Example



- Adding ‘o’ to “c”: first kind
- Adding ‘o’ to “”: second kind
- Adding second ‘c’: third kind
- Naïve implementation is *not* $O(m)$
- Improved by implementation tricks

Bitmap index

- Attribute domain consists of a small number of distinct values
- Each distinct value has a bitmap or a bit vector
- Length of bit vector is size of database
- Suppose number of objects is n
- Number of distinct values is m $(v_1, v_2, \ldots, v_m)$
- If value of attribute for i^{th} object is v_j,
  - i^{th} bit of j^{th} bit vector is 1
  - i^{th} bit of all other bit vectors is 0

Bitmap index

- Attribute domain consists of a small number of distinct values
- Each distinct value has a bitmap or a bit vector
- Length of bit vector is size of database
- Suppose number of objects is n
- Number of distinct values is m $(v_1, v_2, \ldots, v_m)$
- If value of attribute for i^{th} object is v_j,
    - i^{th} bit of j^{th} bit vector is 1
    - i^{th} bit of all other bit vectors is 0
- Can be stored as a regular two-dimensional bit array

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
    - Gender:

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male $= (10011)$,

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male = (10011), female = (01100)
  - Grade:

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male = (10011), female = (01100)
  - Grade: A = (01001),

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male = (10011), female = (01100)
  - Grade: A = (01001), B = (00000),

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male $=$ (10011), female $=$ (01100)
  - Grade: A $=$ (01001), B $=$ (00000), C $=$ (10100),

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male = (10011), female = (01100)
  - Grade: A = (01001), B = (00000), C = (10100), D = (00010)

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male = (10011), female = (01100)
  - Grade: A = (01001), B = (00000), C = (10100), D = (00010)
- Queries are answered using bitmap operations
- O/S allows efficient bitmap operations when packed in word sizes

Example

Object	Gender	Grade
O_1	male	C
O_2	female	A
O_3	female	C
O_4	male	D
O_5	male	A

- Two sets of bit vectors, corresponding to each type of attribute
  - Gender: male = (10011), female = (01100)
  - Grade: A = (01001), B = (00000), C = (10100), D = (00010)
- Queries are answered using bitmap operations
- O/S allows efficient bitmap operations when packed in word sizes
- Example: Find male students who got C
  - bitmap(male) (bitwise)-AND bitmap(C)
  - 10011 AND 10100 = 10000
  - Object O_1
- Null values require a special bitmap for null in relational databases