Tries and suffixes trees

Alon Efrat
Computer Science Department
University of Arizona

Trie: A data-structure for a set of words

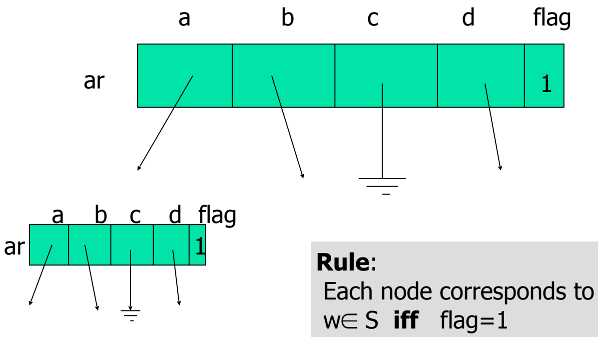
All words over the alphabet Σ={a,b,..z}.
In the slides, the alphabet is only {a,b,c,d}.
S – set of words = {a,aba, a, aca, addd}.
Need to support the operations
- insert(*w*) – add a new word *w* into *S*.
- delete(*w*) – delete the word *w* from *S*.
- find(*w*) is *w* in S ?
  - Future operation:
    - Given text (many words) where is *w* in the text.

- The time for each operation should be O(*k*), where *k* is the number of letters in *w*

- Usually each word is associated with addition info – not discussed here.

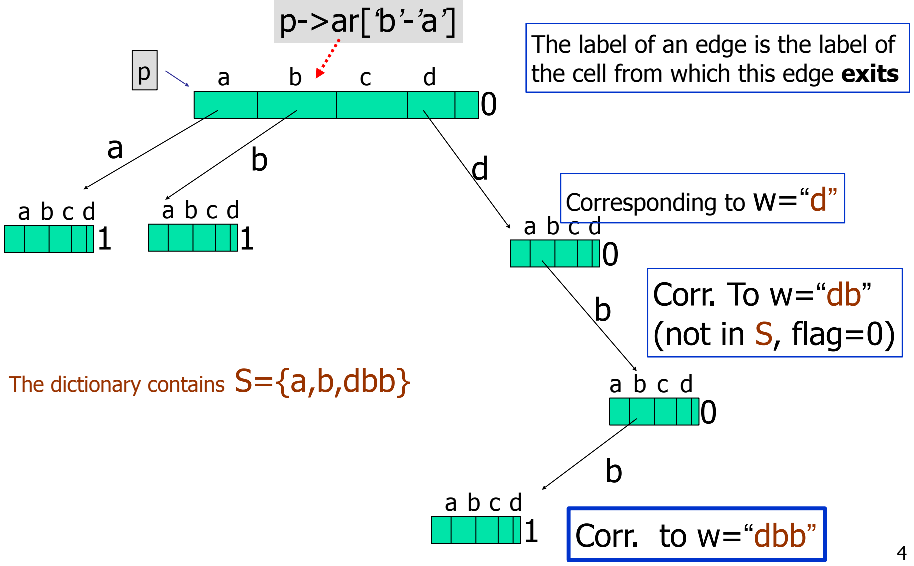
Trie (Tree+Retrive) for S

- A tree where each node is a struct consist
- Struct node {
  - char[4] *ar;
  - char flag ; /* 1 if a word ends at this node. Otherwise 0 */



Rule:
Each node corresponds to a word w.
w∈ S **iff** flag=1

A trie - example



p->ar['b'-'a']

The label of an edge is the label of the cell from which this edge **exits**

Corresponding to W="d"

Corr. To w="db"
(not in S, flag=0)

The dictionary contains S={a,b,dbb}

Corr. to w="dbb"

Finding if word *w* is in the tree

```
p=root; i =0 // remember - each string ends with `\0’
While(1){
    If w[i] == ‘\0’   //we have scanned all letters of w
        then return the flag of p ; else
    If $(p.a[w[i]-'a']) == NULL$  //the entry of p correspond to w[i] is NULL
        return false;
    $p = (p.a[w[i]-'a'])$ //Set p  to be the node pointed by this entry
    i++;
}
```

Inserting a word *w*

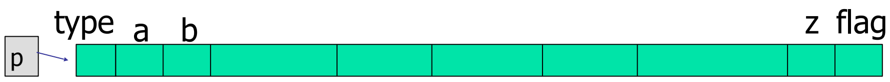
- Try to perform find(*w*).
  - If runs into a NULL pointers, create new nodes along the path.
  - The flag fields of all new nodes is 0.
- Set the flag of the last node to 1

Deleting a word *w*

- Find the node p corresponding to w (using `find’ operation).
- Set the flag field of p to 0.
- If p is dead (I.e. flag==0 and all pointers are NULL) then free(p), set p=parent(p) and repeat this check.

Heuristics for saving space

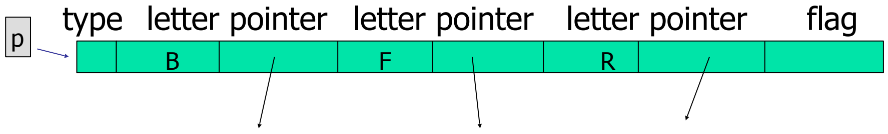
- The space required is Θ(|Σ| |S|).
- To save some space, if Σ is larger, there are a few heuristics we can use. Assume Σ={a,b..z} .
- We use two types of nodes
  - Type “A”, which is used when the number of children of a node is more than 3



Note – the letters are not stores explicitally

Heuristics for space saving

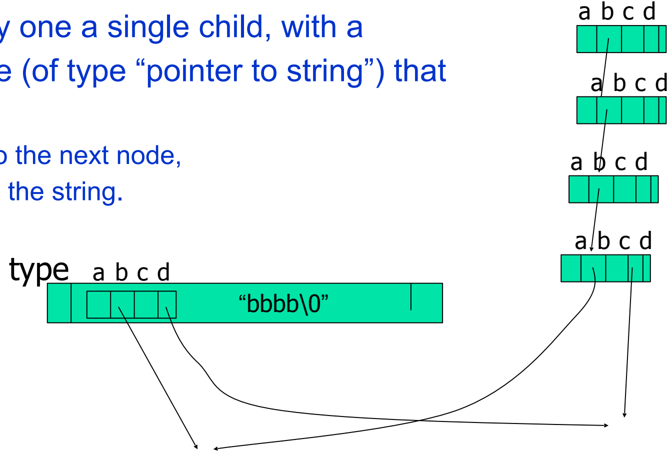
- Type “B” is used if there are 3 or less children:
- The “letter” of the child is also stored:



•The rule of the flag is the same as in type “A” nodes.
•We only store the 3 pointers, but we need to know to which letters they corresponds to.

Another Heuristics – path compression

- Replace a long sequence of nodes, all having only one a single child, with a single node (of type “pointer to string”) that maintains
  - a point to the next node,
  - a point to the string.



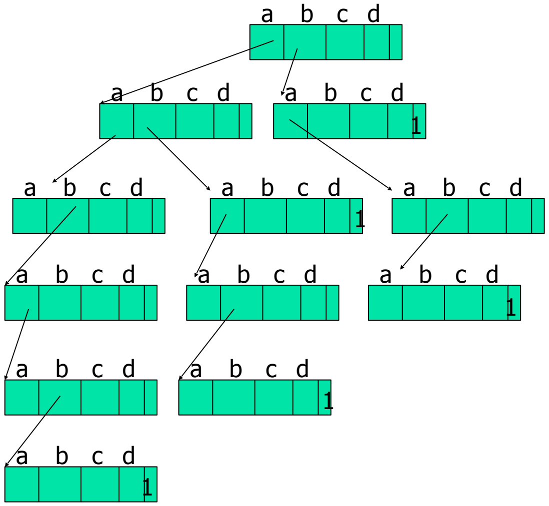
Suffix tree.

- Assume *B* (for book) is a very long text.
- Want to preprocess *B*, so when a word *w* is given, we can quickly find if it is in *B*.
- We can find it in O(|*w*|).
- Idea:
  - Consider *B* as a long string.
  - Create a trie *T* of all suffixes of *B*.
  - In addition to the flag (specifying if a word ends at node), we also stored the index in *B* where this word begins.
  - Example *B=“aabab”*
    S={“aabab”, “abab”, “bab”, “ab”, “b”}

Observation: w appears in B ⇔
w is the prefix of a suffix of B.
Example: B=“hello**nice**world”, w=“nice”.

Suffix tree.

Example *B=“aabab” S={“aabab”, “abab”, “bab”, “ab”, “b”}*



To know **where** a word appear in B, we store with each node the index of the beginning of the suffix in B.

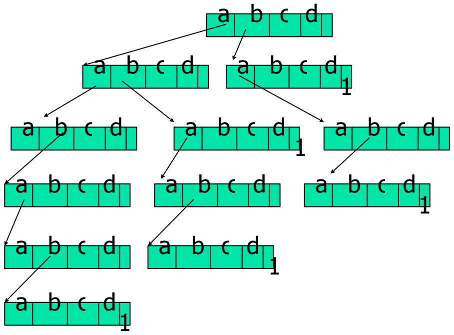
(we can store only the first appearance of the word in the text)

Size of suffix tree

Example *B=“aabab” S={“aabab”, “abab”, “bab”, “ab”, “b”}*

Assume n=|B|.
Total length of all string $\Theta(n^2)$
Size of a node is $|\Sigma|$
So size of the tree is $\Theta(n^2 |\Sigma|)$.

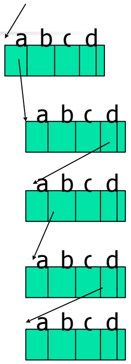
Time to construct the tree $\Theta(n^2)$



We can save some space.

Example B=“aabab”
S={“aabab”, “abab”, “bab”, “ab”, “b”}

Suffix tries on a diet

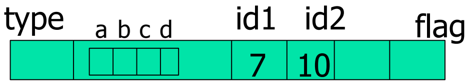


Def: a *thread is a* path from node *u to node v in the trie,* consisting of nodes of outdegree *1* (except maybe the last one) and *flag=0.*

Obs: There is a contagious part of *B*, identical to the string the shred represents. We call this part the shred-string

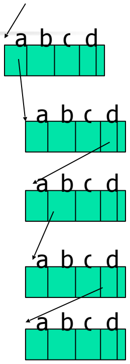
We stores the book *B* itself as an array.

We use a new type of nodes, called thread-nodes, maintain the first (*id1*) and last (*id2*) indexes of the shred-string in *B*.



1 7 10
B=“cadbdaadbd

Suffix tries on a diet - cont

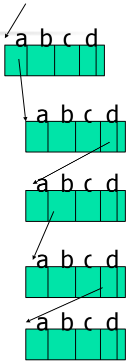


Algorithm for constructing a “thin” trie:

Given B – create an empty trie T, and insert all n suffixes of *B* into *T* --- generating a trie of size $\Theta(n^2)$.

Traverse the tries, and each time that a shred is seen, replace all nodes of the shred with a single shred-node.

Suffix tries on a diet - cont



- Clearly the use of thread-nodes saves some-but can we prove something ?
- Observations: Every leaf of T must be the end of some prefix of B. So the number of number of leaves of T is $\leq n$.
- n=|B|
- To bound the size of T, we will need to bound the number of internal nodes.
- Observations:
  - T might contain special nodes whose flag=1 (a suffix terminates at these nodes).
  - The number of special nodes is $\leq n$ (since this is the number of suffixes).
- What about other internal nodes of T ?

Suffix tries on a diet - cont

Lemma: Let T′ be a rooted tree with m leaves, where each internal node has ≥ 2 children. Then T′ has $\leq m$ internal nodes. (proof - easy induction. Homework)

Back to thin suffix tries *T*:

- *T has* $\leq n$ special nodes (with flag=1) and
- *T has* $\leq n$ leaves.
- Every other nodes has ≥ 2 children. (with flag=1). Applying the Lemma in this case, implies that the total number of internal nodes $\leq 2n$.

- Conclusion: The number of nodes in T is $\leq 3n$ (much better than the uncompressed version that could have $\Theta(n^2)$ nodes.
- So the size of the trie is only a constant more than the size of the book.

Quadtrees

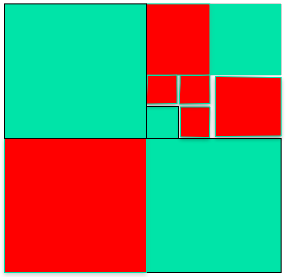
:

A simple data structure for geometric objects (e.g. points, houses, an image, 3D scene)

Support efficiently a very wide variety of queries.

Shares similarities with tries, hence taught together.

QuadTrees

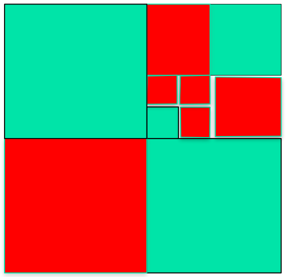


Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red.**

(more general and interesting examples – soon)

Need to represent the shape “compactly”

QuadTrees



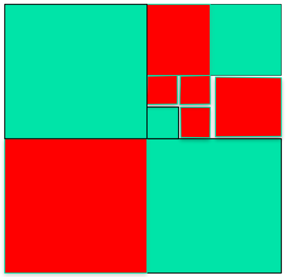
Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red.**

(more general and interesting examples – soon)

Need to represent the shape “compactly”

Need a data structure that could answers multiple types of queries. For example:

QuadTrees



Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red**.

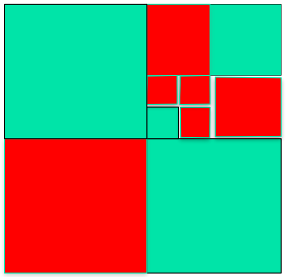
(more general and interesting examples – soon)

Need to represent the shape “compactly”

Need a data structure that could answers multiple types of queries. For example:

1.For a given point q, is q red or green ?

QuadTrees



Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red**.

(more general and interesting examples – soon)

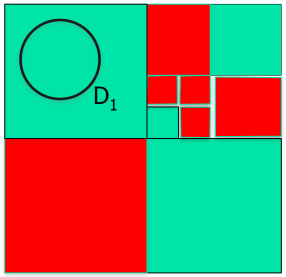
Need to represent the shape “compactly”

Need a data structure that could answers multiple types of queries. For example:

1.For a given point q, is q red or green ?

2.For a given query disk D, are there any green points in D ?

QuadTrees



Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red**.

(more general and interesting examples – soon)

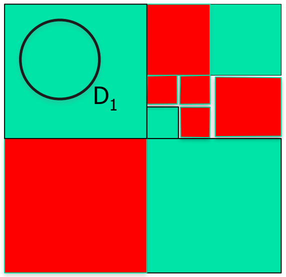
Need to represent the shape “compactly”

Need a data structure that could answers multiple types of queries. For example:

1.For a given point q, is q red or green ?

2.For a given query disk D, are there any green points in D ?

QuadTrees



Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red**.

(more general and interesting examples – soon)

Need to represent the shape “compactly”

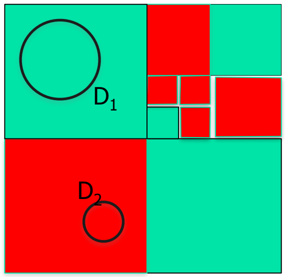
Need a data structure that could answers multiple types of queries. For example:

1.For a given point q, is q red or green ?

2.For a given query disk D, are there any green points in D ?

3.How many green points are there in D ?
4.Etc etc

QuadTrees



Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red.**

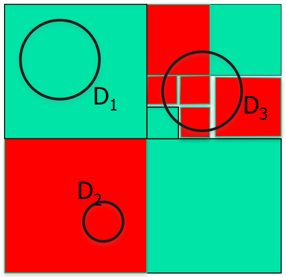
(more general and interesting examples – soon)

Need to represent the shape “compactly”

Need a data structure that could answers multiple types of queries. For example:

1. For a given point q, is q red or green ?
2. For a given query disk D, are there any green points in D ?
3. How many green points are there in D ?
4. Etc etc

QuadTrees



Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red.**

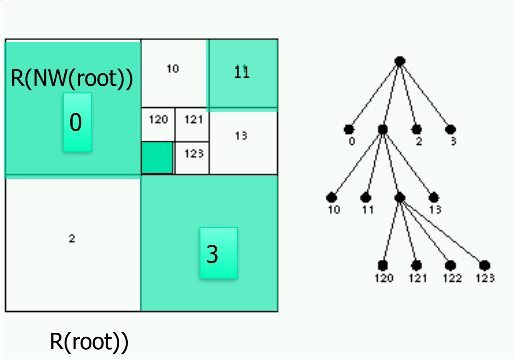
(more general and interesting examples – soon)

Need to represent the shape “compactly”

Need a data structure that could answers multiple types of queries. For example:

1. For a given point q, is q red or green ?
2. For a given query disk D, are there any green points in D ?
3. How many green points are there in D ?
4. Etc etc

Regions of nodes



R(root))

A tree where each internal node has 4 children.

In general, every node v is associated with a region of the plane. Lets denote this region by R(v).

R(root) is the whole region of interest (e.g. input image or USA)

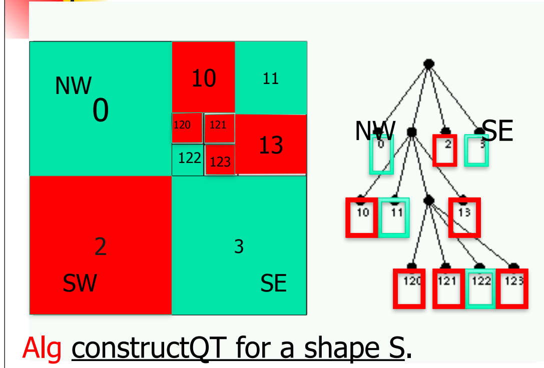
The smallest possible area of R(v) is a single **pixel**.

For every non-root node v, we have $R(v) \subset R(parent(v))$

Let NW(v) denote the North West child of v. (similarly NE, SW, SE)

R(v) = is the union of R(NW(v)), R(NE(v)) R(SW(v)), R(SE(v))

QuadTrees



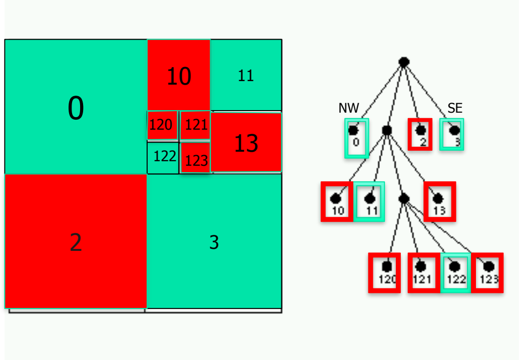
- Assume we are given a red/green picture defined on a $2^h \times 2^h$ grid of **pixels**.
- Each pixel has as a unique color (Green or Red)
- Every node $v \in T$ **is associated with a geometric region** ***R(v)***

Alg <u>constructQT for a shape S</u>.

- **input** – a node $v \in T$, and a shape *S.*
- **Output** – a Quadtree T_v representing the shape of *S within R(v)*).

- If *S* is fully green in *R(v),* or S is fully red in *R(v)* – then
-     *v* is a leaf, labeled Green or Red. Return ;
- Otherwise, divide *R(v)* into 4 equal-sized quadrants, corresponding to nodes v.*NW,* v.*NE,* v.*SW,* v.*SE*.
- Call <u>constructQT</u> recursively for each quadrant.

QuadTrees



Consider a picture stored on an $2^h \times 2^h$ grid. Each pixel is either red or green.

We can represent the shape "compactly" using a QT.

Height – at most h.

Point location operation – given a point q, is it black or white

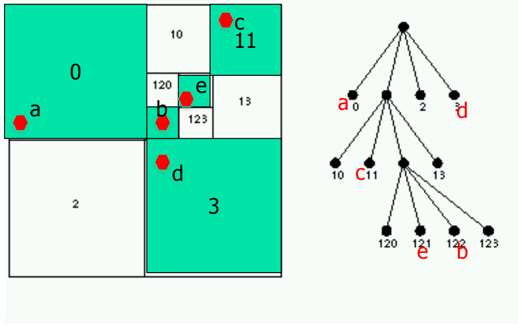
– takes time O(h)

- could it be much smaller ?

Many other operations are very simple to implement.

QuadTree for a set of points

given: a set of points $S=\{a,b,c,d,e\}$, each with its (x,y) coordinates



Now consider a set of points (red) but on a $2^h \times 2^h$ grid.

Splitting policy: Split until each quadrant contains ≤1 point.

Build a similar QT, but we stop splitting a quadrant when it contain ≤1 point (or some other small constant)

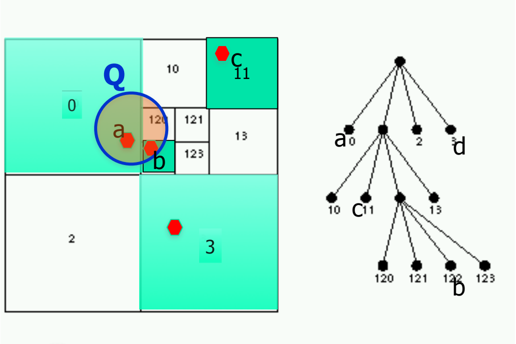
Point location operation – given a point q, is it black or white

– takes time O(h) (in practice, usually much less)

Many other **splitting polices** are very simple to implement.

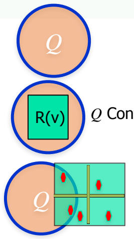
(eg. A leaf could contain contains ≤17 points)

QuadTrees for a set of points



```
Report(Q,v)
// Q – a query disk
/* report all the points in stored at the subtree rooted at v,  which are contained inside Q. */

1.If v is NULL – return.
2.If R(v) is disjoint from Q –return NULL.
3.If R(v) is fully contained in Q – report all points in the subtree rooted at v.
4.If v is a leaf – check each point in R(v) if inside Q
5.Else  //$R(v)$ Partially overlaps $Q$
    Report(Q, NW(v)) and
    Report(Q, NE(v))  and
    Report(Q, SW(v)) and
    Report(Q, SE(v))
```



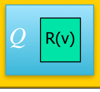
Q disjoint from $R(v)$; $Q \bigcap R(v) = \varnothing$

Q Contains $R(v)$;

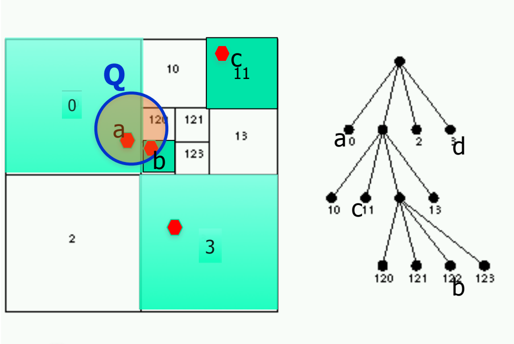
$R(v)$ Partially overlaps Q

Comment: In practice, it is much easier to work with query region which is an axis-parallel rectangle (why?). We use disks in the slides for visualization.

For example, to check if $R(v) \subseteq Q$, it is enough to check MinX, MinY, MaxX,MaxY

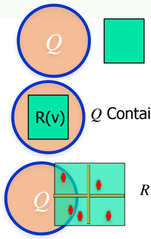


QuadTrees for a set of points



```
Report(Q,v)
// Q – a query disk
/* report all the points in stored at the subtree rooted at v,  which are contained inside Q. */

1.If v is NULL – return.
2.If R(v) is disjoint from Q –return NULL.
3.If R(v) is fully contained in Q – report all points in the subtree rooted at v.
4.If v is a leaf – check each point in R(v) if inside Q
5.Else  //$R(v)$ Partially overlaps $Q$
    Report(Q, NW(v)) and
    Report(Q, NE(v))  and
    Report(Q, SW(v)) and
    Report(Q, SE(v))
```



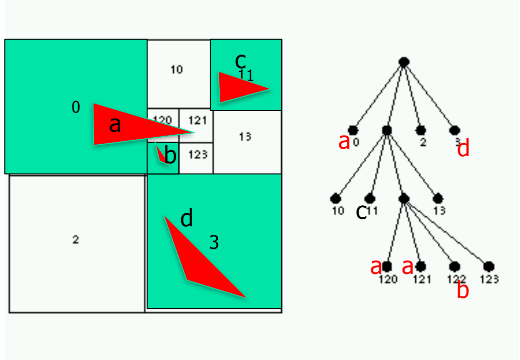
Q disjoint from $R(v)$; $Q \bigcap R(v) = \varnothing$

Q Contains $R(v)$;

$R(v)$ Partially overlaps Q

Comment: In practice, it is much easier to work with query region which is an axis-parallel rectangle (why?). We use disks in the slides for visualization.

QuadTrees for shape



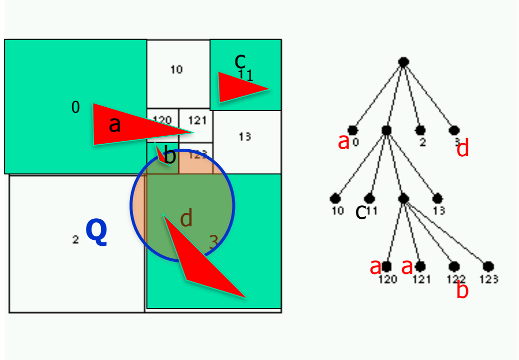
Input: Set S of triangles $S=\{t_{1 \dots} t_n\}$

Splitting policy: Split quadrant if it intersects more than 1 triangle of S.

Note – a triangle might be stored in multiple leaves. Some leaves might store no triangles.

Finding all triangles inside a query region Q – essentially same Report Report(Q,v) as before
(minor modifications)

QuadTrees for shape



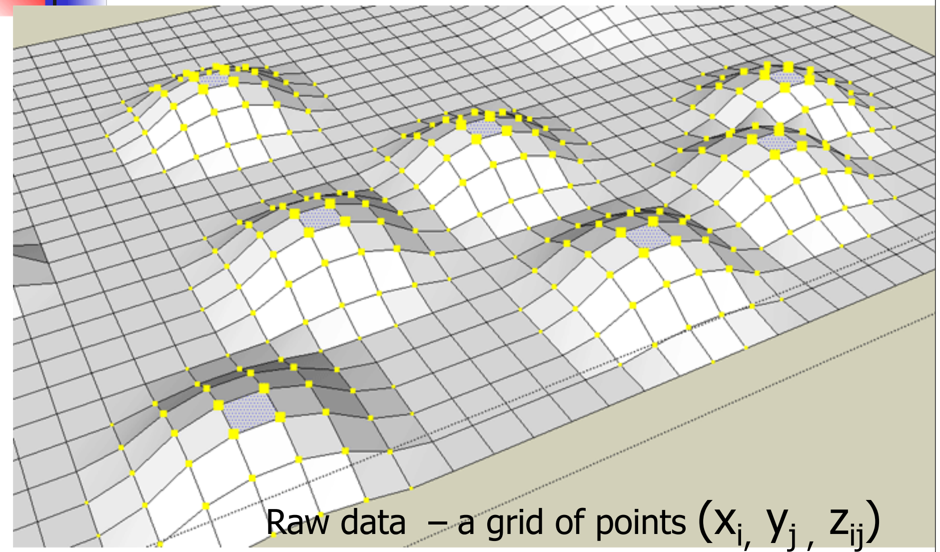
Input: Set S of triangles $S=\{t_{1 \dots} t_n\}$

Splitting policy: Split quadrant if it intersects more than 1 triangle of S.

Note – a triangle might be stored in multiple leaves. Some leaves might store no triangles.

Finding all triangles inside a query region Q – essentially same Report Report(Q,v) as before
(minor modifications)

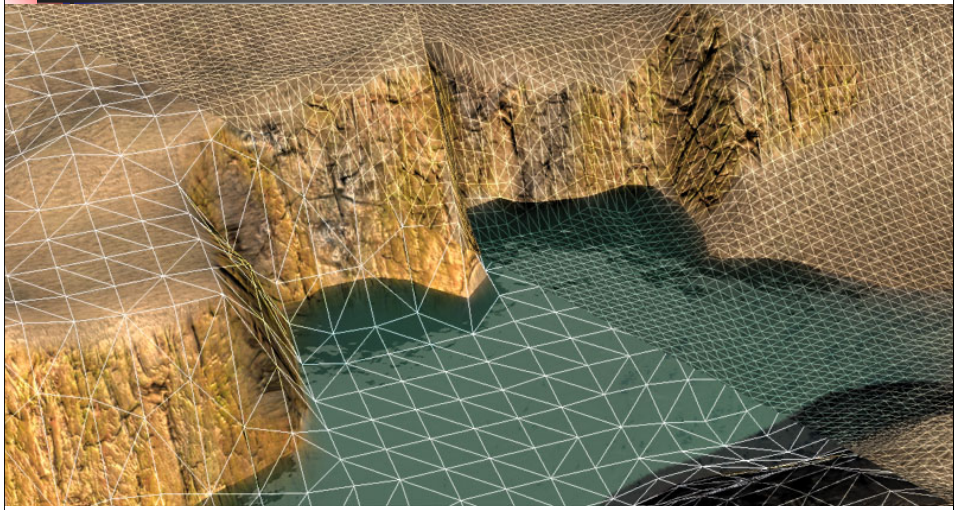
Terrain representations



Raw data – a grid of points (x_i, y_j, z_{ij})

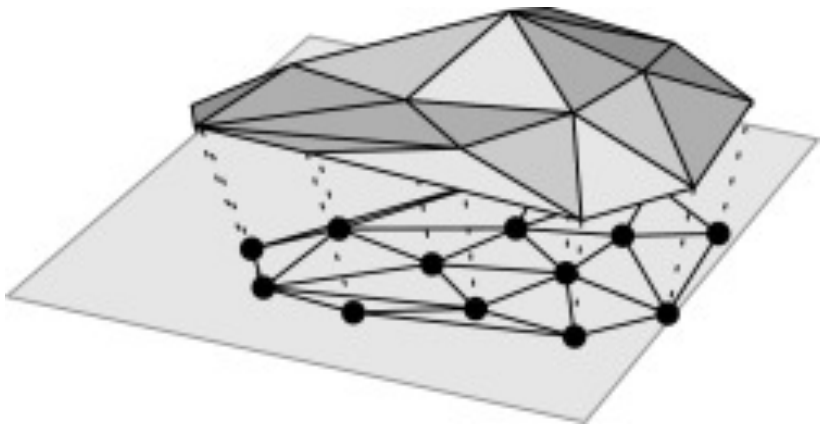
For every grid point i,j

Triangulated terrain
(TIN – Triangulated irregular network



Each triangle approximately fits the surface below it

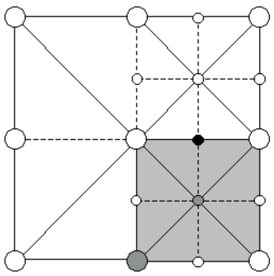
How to find good triangulation ?



Each triangle approximately fits the surface below it

(credit SCALGO)

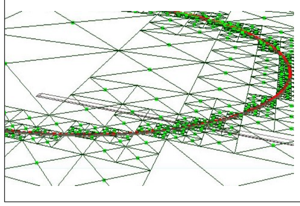
How to find good triangulation ?



- Input – a very large set of points $S=\{ (x_i , y_j , z_{ij}) \}$.
- z_{ij} is the elevation at point (x_i , y_i)
- Want to create a surface, consists of triangles, where each triangle interpolates the data points underneath it.
- Idea: Build a QT T for the 2D points.
- (if want triangles: Each quadrant is split into 2 triangles)
- Assign to each vertex the height of the terrain above it.
- The approximated elevation of the terrain at any point is the linear interpolation of its elevated vertices.

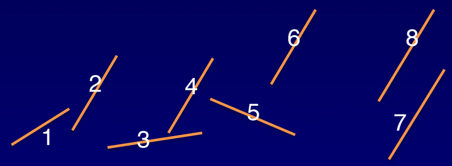
QT Split Policy: Splitting a quadrant into 4 sub-quadrants:

- split a node ***v*** if for some date point $(x_i , y_i)\in R(v)$, the elevation of z_{ij} is too far from the the corresponding triangle. If not, leave ***v*** as a leaf.
- That is, (x_i ,y_j , z_{ij}) it is too far from the interpolated elevation.
- Note: A quadrant might contain a huge number of points, but they behave smoothly. E.g. all a the sloop of a mountain, but this slope is more or less linear.

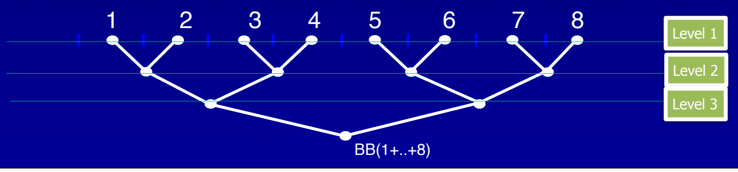


R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

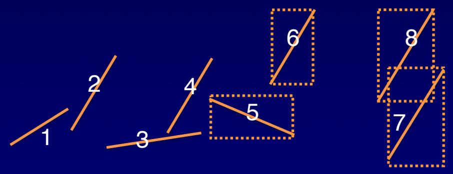


- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1”.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.

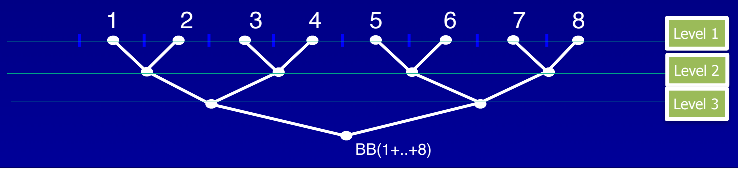


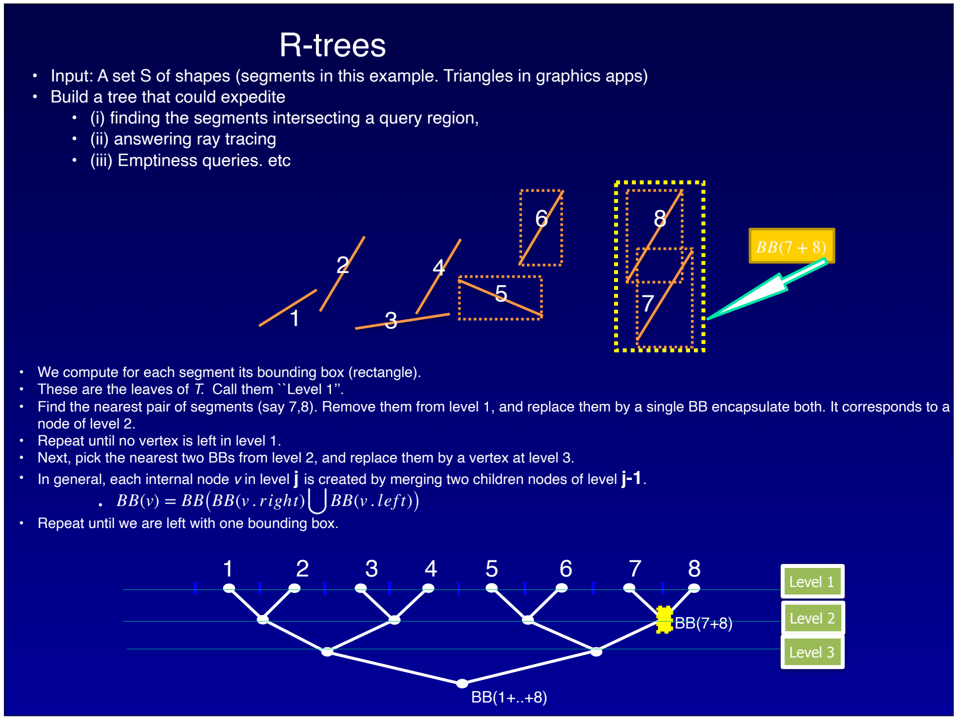
R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc



- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1”.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.

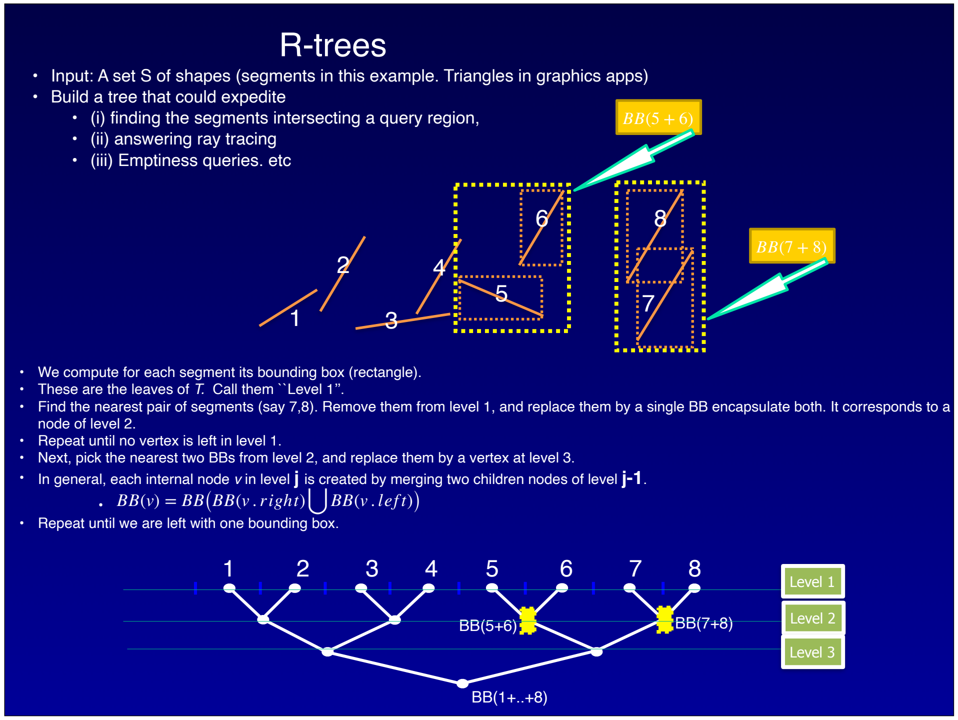




R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

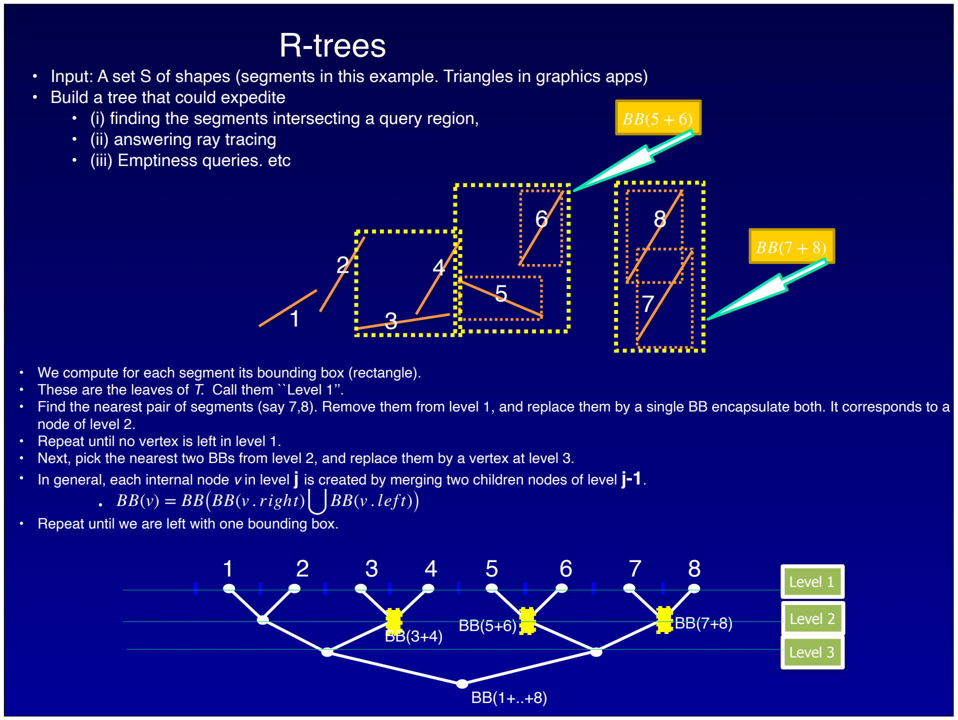
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1”.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.



R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

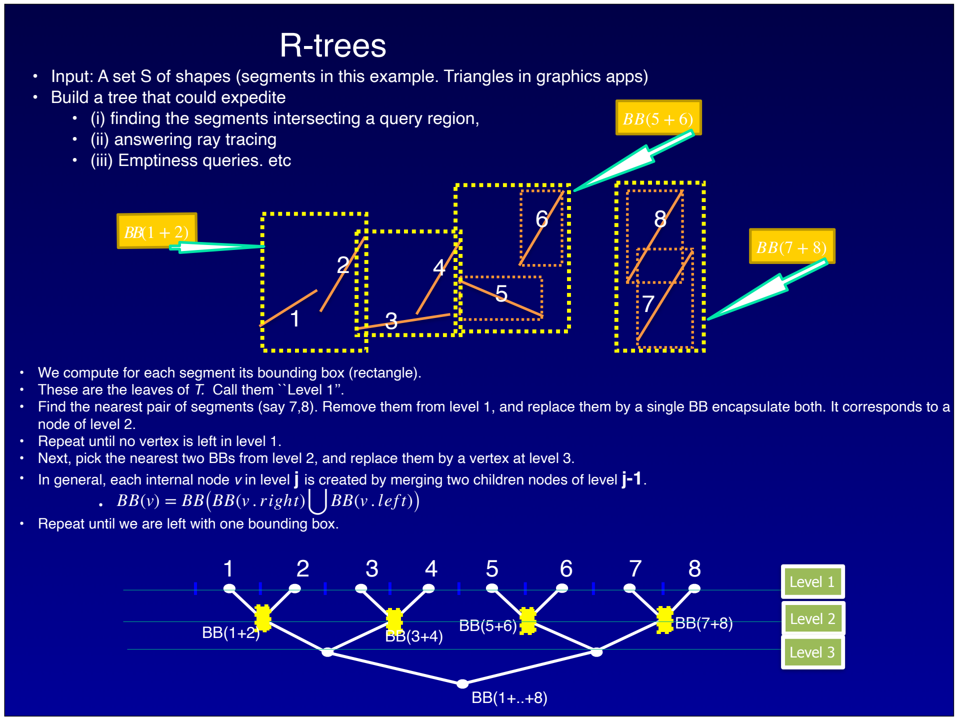
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1”.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.



R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

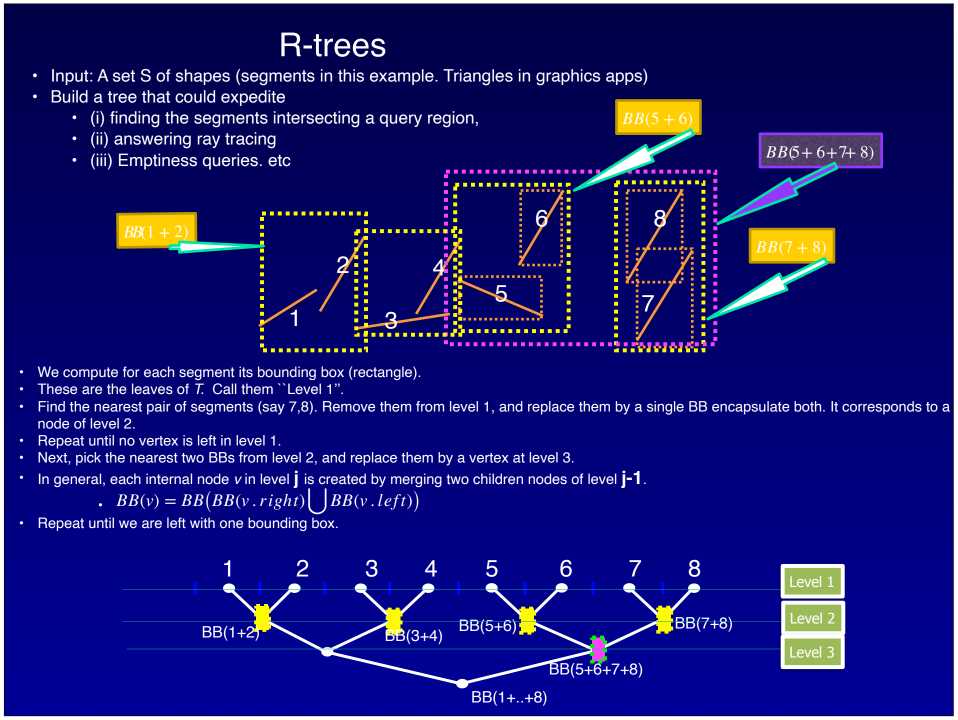
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1”.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.



R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

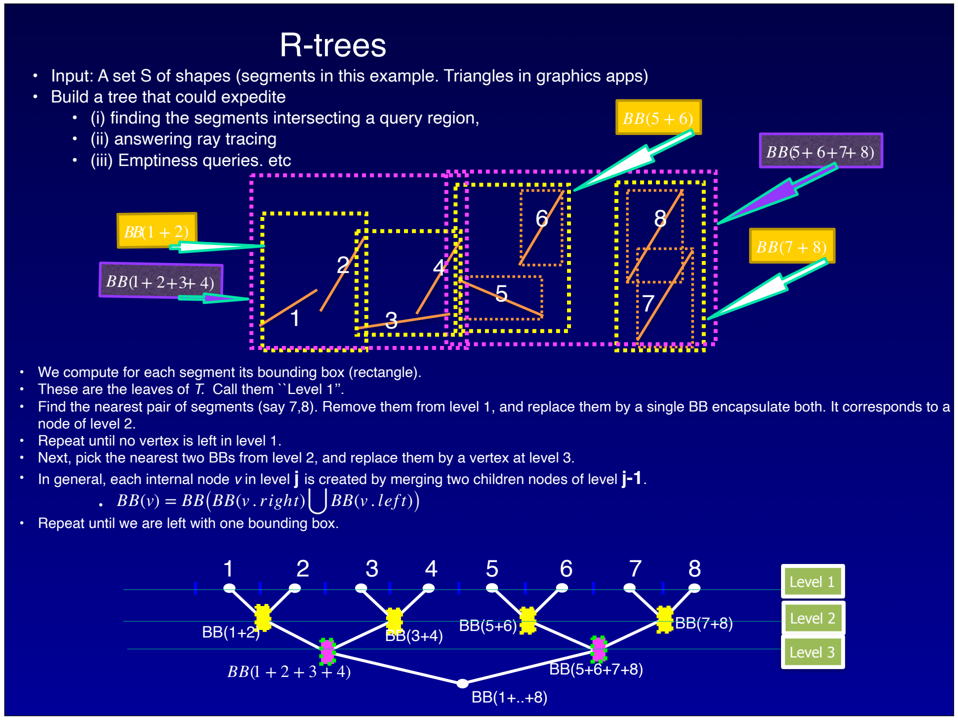
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1”.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.



R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

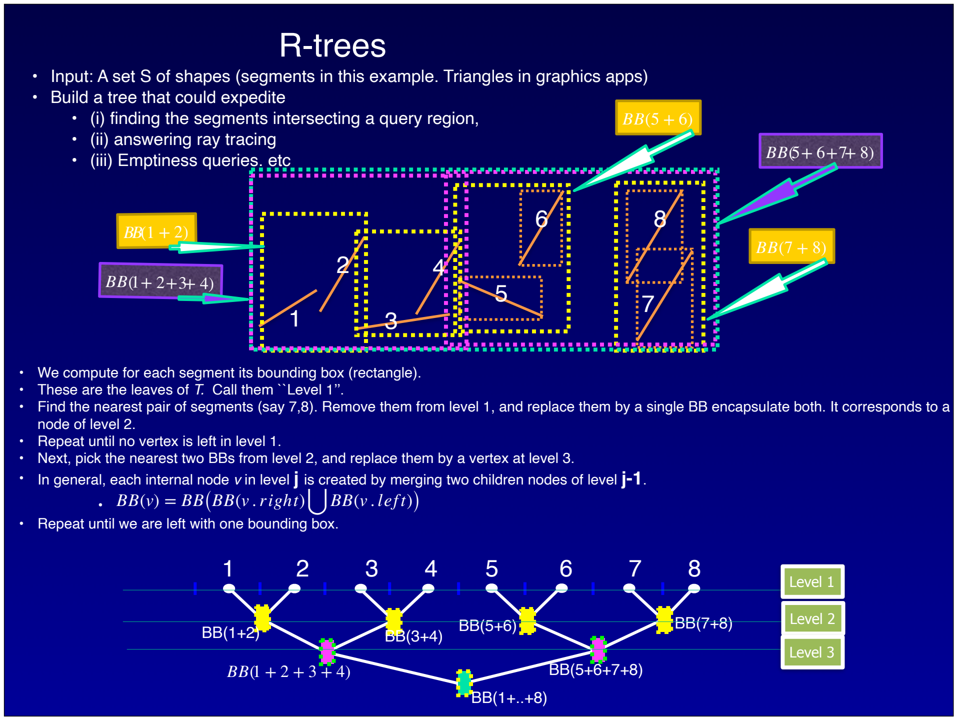
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1''.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.



R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

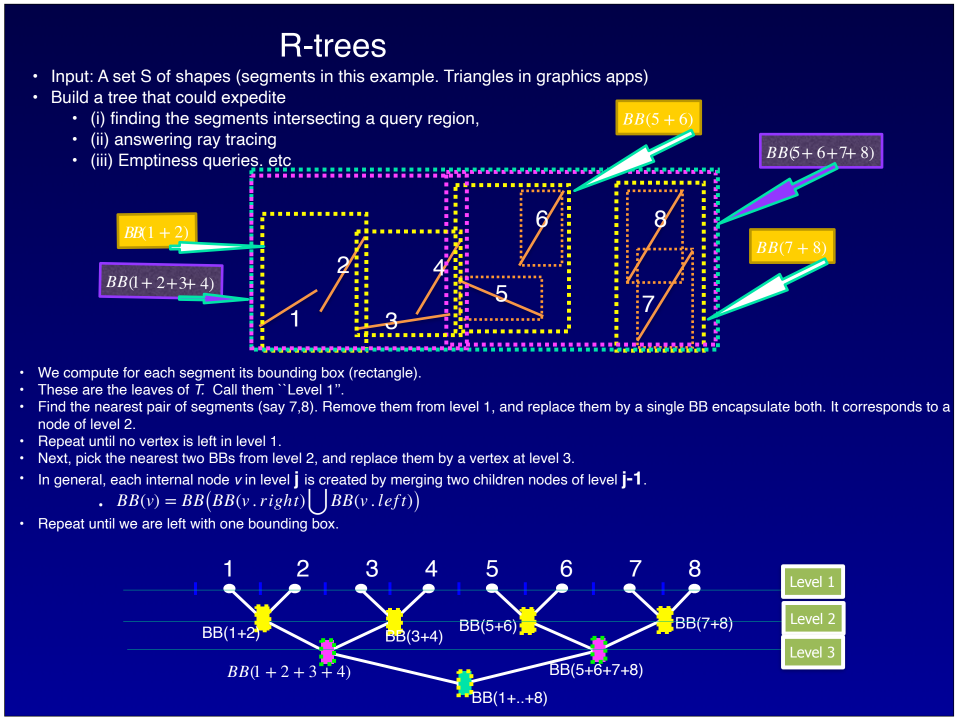
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1''.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.



R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1''.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.



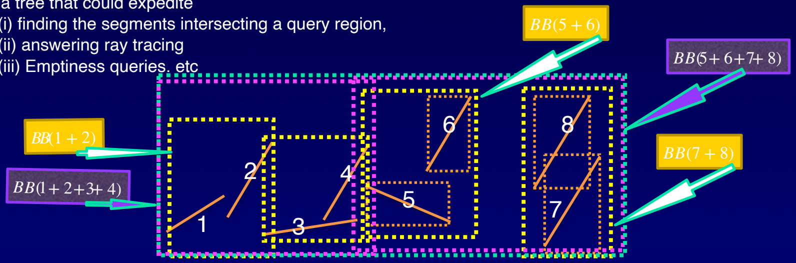
R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

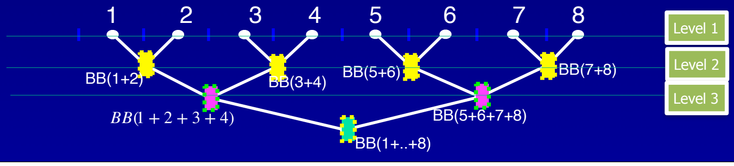
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1''.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.

R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

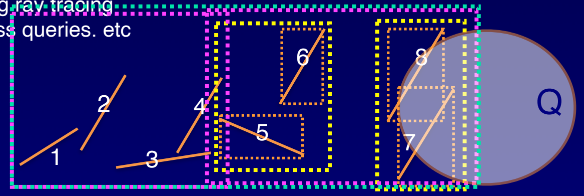


- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1''.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v.right) \bigcup BB(v.left)\right)$
- Repeat until we are left with one bounding box.

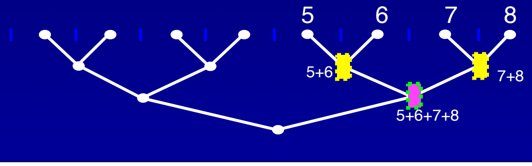


R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc

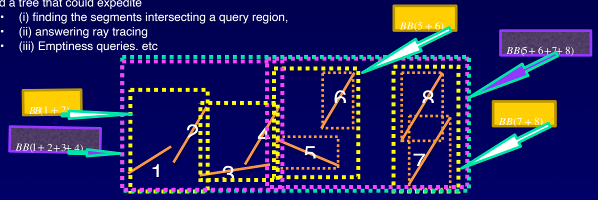


Once a query region Q is given, we need to report the segments intersecting Q.
Check if Q intersects BB(root)
If not, we are done. If yes, check recursively if Q intersects BB(v.left) and BB(v.right)



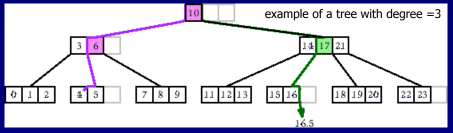
R-trees, B-trees and hard drives . Large degree helps

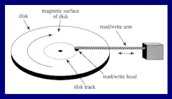
- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc



- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1''.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **i** is created by merging two children nodes of level **i-1**.

- In practice, it is sometimes preferable to create trees with a very large degrees. For example, each internal node, will have between 100 to 500 children





- Lets think about the process of a search. We visit the root, then one of its children, one of its grand-children … until we reach a leaf.
- The seek-time in disks, and even in SSD, is much slower than the seek-time for main memory. Therefor, once the head of the disks is located in the correct place, we usually read a bucket - about 4KByte of memory.
- The bottleneck of the *search/insert/delete* operation is the number of seek operations (number of I/Os).
- The number of seek-operation is proportional to height of the tree.
- Say $n = 10^9$. The height of a tree of degree 2 with n leaves is $\log_2(10^6) \approx 30$
- If the each node contains about 1000 segments, or keys, then the height (and number of I/Os) is only $\log_{1000}(10^9) = 3$
- B-trees and R-trees are the most popular and important data structures for big data.