(*Lec19*) Geometric Data Structures for Layouts

- What you know
  - Some basic ASIC placement (by annealing)
  - Some basic ASIC routing (global versus detailed, area routing by cost-based maze routing)
  - Some timing analysis: logical and electrical
- What you don't know
  - Basic representation techniques for handling a lot of geometry
  - Standard "procedures" supported on these geometric data structures
  - Pros, cons comparative features of common geometric structures

Copyright Notice

Where Are We?

- Geometric data structures for BIG layouts…

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

Nominal Deadlines…

Last 760 lecture (probably…

Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16
	17	18	19	20	21	Finals

Revisions:

Move HW5 and Paper3 to 17 Dec Monday, due 5pm Lyz Knight's office. Proj3 still Due 14 Dec Friday.

HW5

6 PPT slide paper review

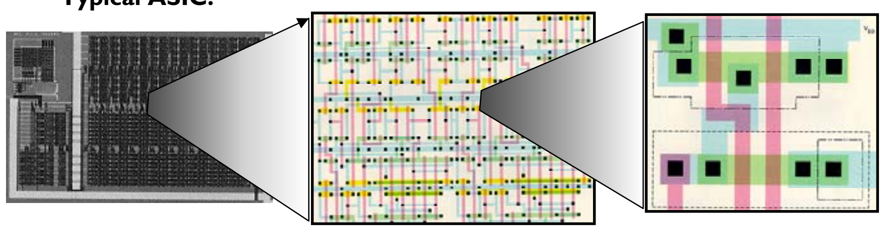
Proj 3 demos

- …and, this is clearly a bit extreme for the last week of class
  - Open to suggestions for moving *some* deadlines BACK *some*…
  - …but need to be careful not to mess up people with finals, early travel plans for break, etc

Data Structures

- A large IC layout represents a lot of data
  - 10M to 100M devices
  - ~10 to 20 pieces of geometry per device
  - 0.1 to 1 *billion* rectangles--easily
- How do we manage this large collection of data?
  - Depends on what you want to do with the layout.

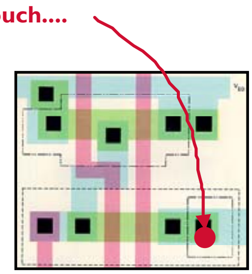
Typical ASIC: | Module or cell: | Logic gate:



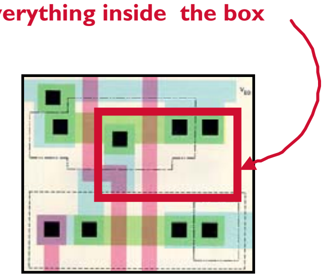
1000s of cells. | 10s of gates. | 100s of rectangles.

2 Basic Kinds of "Queries" on these Structures

A "pick" operation--
Given (x,y), tell me what
I touch....



A "region query" operation--
Given a box, tell me
everything inside the box



Data Structures: Queries

- **Data structure queries**
  - Pick: Given an x,y location, tell me what lives there.
  - Region query: Given a bounding box, tell me what's inside it.
- **Uses**
  - Checking DRC-type layout interactions
  - Printing masks.
  - Extracting electrical circuits from layout.
  - Searching the neighborhood of a given device or circuit.
- **Note:**
  - No inserting or deleting data is done -- just asking where things are.

Data Structures: Layout Modification

- **Adding & Deleting geometry**
  - Inserting or removing rectangles from the data collection.
- **Uses**
  - Interactive layout editing: Cadence Virtuoso or MAGIC.
  - Global and detailed routing.
  - Local rip-up and reroute.
  - Placement "legalization" = fine local adjustments.
- **Caveats**
  - With some data structures, it is easy to add new geometry but difficult to delete.
  - Need to be careful to match structure to the application

Data Structures: Useful References

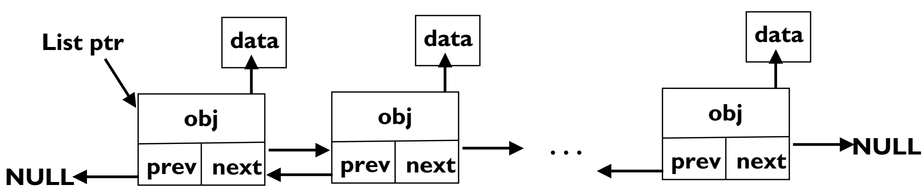
- Survey paper
  - J. Rosenberg, "Geographical data structures compared: a study of data structures supporting region queries," *IEEE Trans. CAD*, CAD-4, 1, Jan. 1985.
  - Old, but very useful. Actually has code (in a somewhat archaic looking C now) for a lot of these data structures.
- Tile plane paper
  - J. Ousterhout, "Corner stitching: a data structuring technique for VLSI layout tools," *IEEE Trans. CAD*, CAD-3, 1, Jan. 1984.
  - It's the data structure underneath the Berkeley MAGIC layout editor
  - Yeah, the *same* Ousterhout who developed tcl/tk, and MAGIC.

Data Structures: Survey of Types

- Linked List
  - Simple but slow
- Bins
  - Straightforward - commonly used
- Quad Trees
  - Very widely used
- k-d Trees
  - Good complexity but limited
- Corner Stitched Tile Planes
  - Good for layout editing, fine-grain local manipulation of shapes

Data Structures: Linked Lists

- OK for grouping small amounts of data
  - But as soon as you have lots of rectangles & complex queries, this is bad
- Complexity (assume you have N rectangles)
  - Time:
    - Find O(N)
    - Insert O(N) (O(1) if not sorted)
    - Delete O(N)
  - Memory:
    - O(N) - one link for each data item.



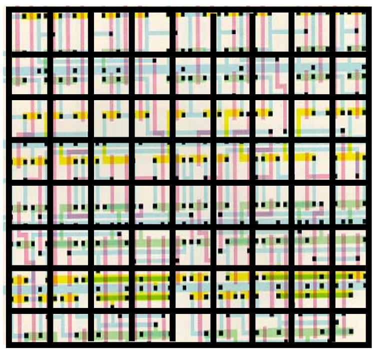
Practical Issues: Linked Lists

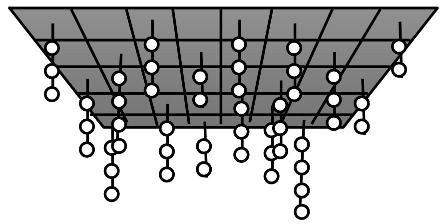
- Lousy complexity, but widely used…
  - …when you only have a "a few" objects to represent
  - More sophisticated structures will get presented after this…
  - …but, they all come with some overhead; nothing is free
  - If you have a small # of rectangles to handle, a linked list *is the right way*
  - Where the breakpoint is between doing it like this ("flat") and using a smart structure is something you have to determine empirically

Data Structures: Bins

Bin idea

- Divide up surface of the chip into rectangular **bins** (also called **buckets**)
- Inside each bin, you have a linked list of all the rectangles you touch.



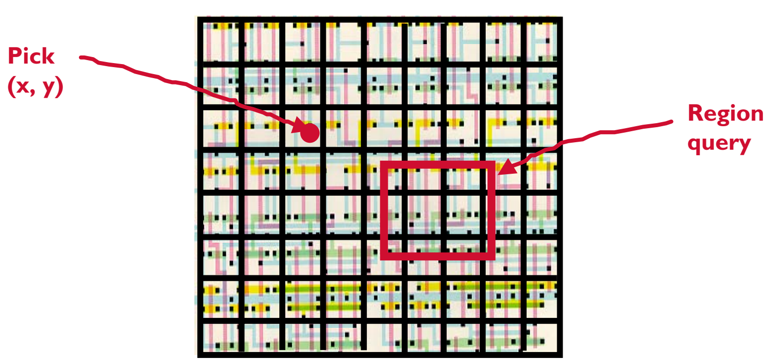


Data organized by a 2D array of geometric bins, each of which holds all local rectangles in a linked lists.

Data Structure: Bins

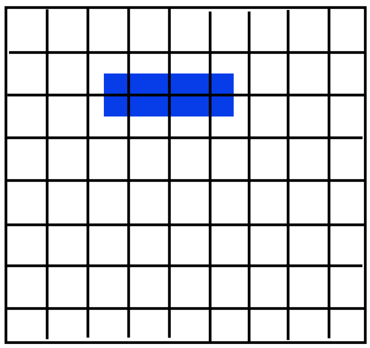
Queries

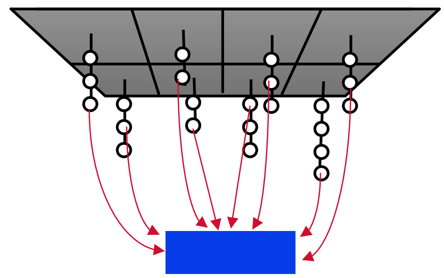
- Pick: go to the bin with the (x,y) you want, look at all the rectangles
- Region query: go to *all* the bins that touch the region, look at *all* the rectangles



Data Structures: Bins

- **How does it really work**
  - Need a pointer to a "rectangle object" from every bin it touches.
  - May have to walk thru lots of bins to insert/delete a *big* rectangle
  - Impacts the granularity of the grid you pick...

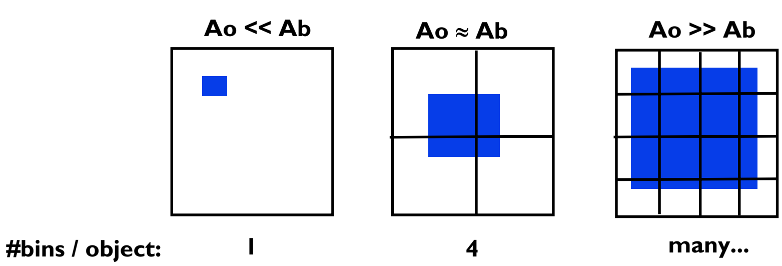




Eight bins point to this one object.

Data Structures: Bins

- **How big should the bins be?**
  - Let, Ao = average object size and Ab = bin size.



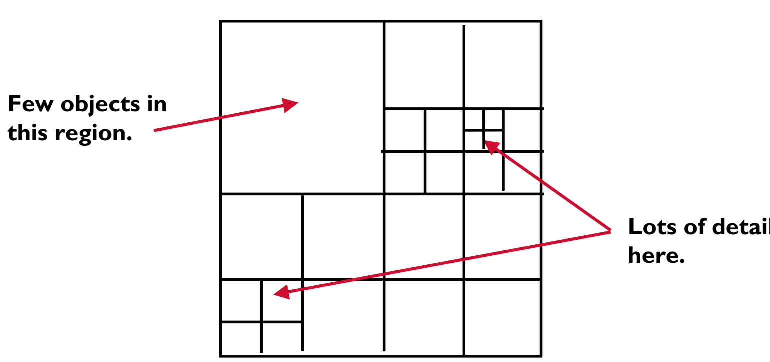
- **If you have many, small bins...**
  - Memory use is large, insert and delete times are long.
  - But "pick" operations are really fast (few objects per bin)
  - Need to be careful to tune bin granularity to problem

Data Structures: Bins

- Summary
  - Good for evenly distributed objects of similar size.
- Complexity
  - Time:
    - Find O(1)
    - Insert O(1)
    - Delete O(1)
  - Memory:
    - O(N) - if number of bins is < number of objects
      - small linked list per bin
  - But this assumes a lot of empirical tuning, and is pretty much a best-case estimate. BAD things can happen if your geometry isn't "evenly distributed"....

Data Structures: Quad Trees

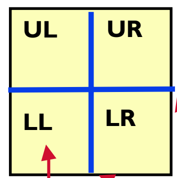
- Problem with bins:
  - What if data is not uniformly distributed?
  - If one small region of the chip is very dense, the search in that region is slow since the bin size is fixed and the linked lists for each bin are long.
  - We want to divide up space in a more dynamic fashion - only subdivide region where the "action" is.



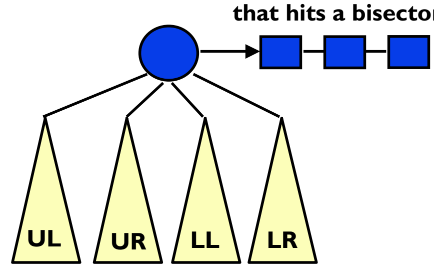
Data Structures: Quad Trees

- Tree data structure with four children:

Given a region (called a *quad*),
subdivide it into four equal parts:



List of geometry
that hits a bisector line

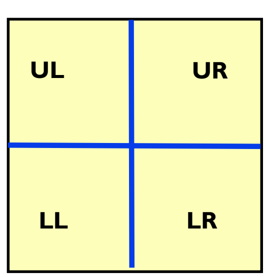


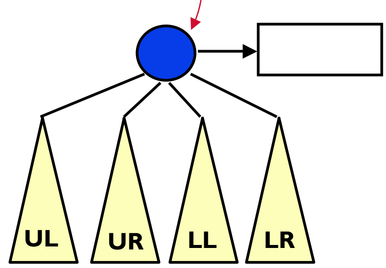
Each child is *another* quad tree.

Rules:
- If an object overlaps any of these bisector (center) lines in this quad, put it in the list for this quad.
- Else if it is totally within a single quadrant, pass the object to the appropriate child tree.

Quad Tree Construction

- Objects that hit either of the bisector lines…
  - These cannot be entirely inside the UL, UR, LL, LR regions
  - So, they go on the 'bisector list'' at the top.

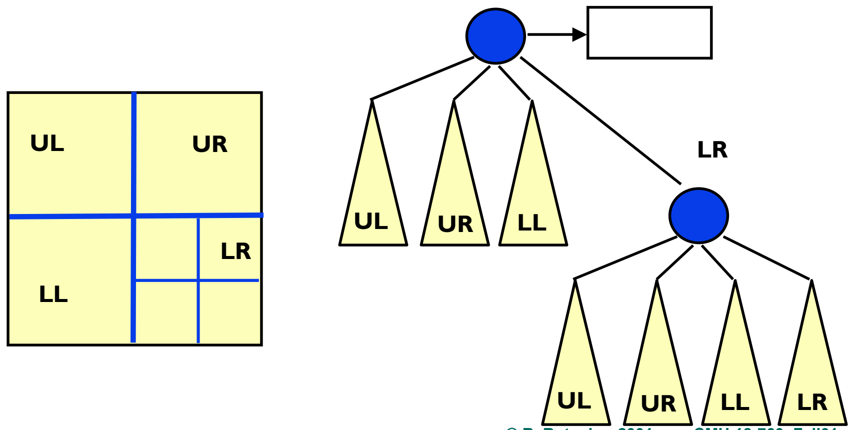




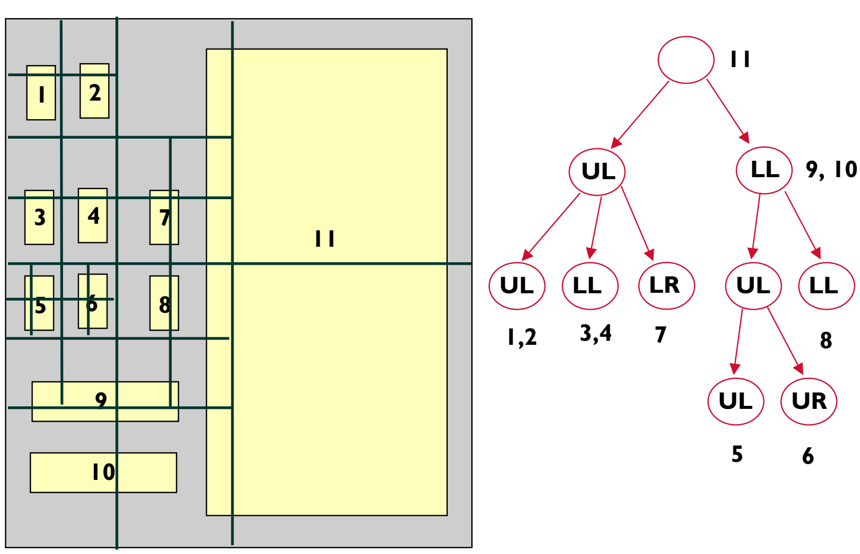
Each child is another quad tree.

Quad Tree Construction

- Objects that don't hit either of bisector lines
  - These live entirely inside one of the UL, UR, LL, LR regions
  - So, they get passed down to me quad tree for that region
  - Just repeat this recursion

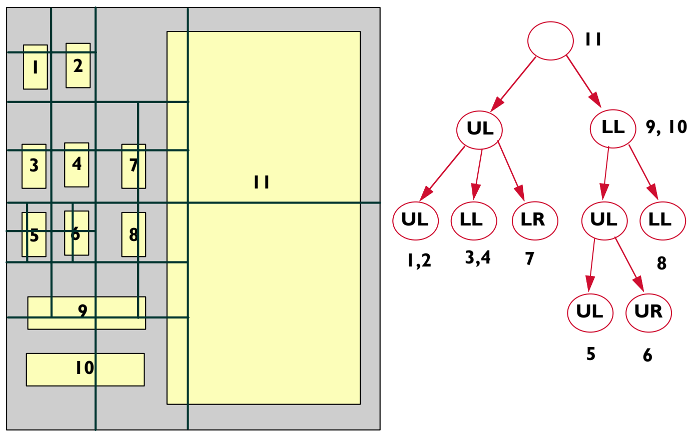


Data Structures: Quad Tree Example



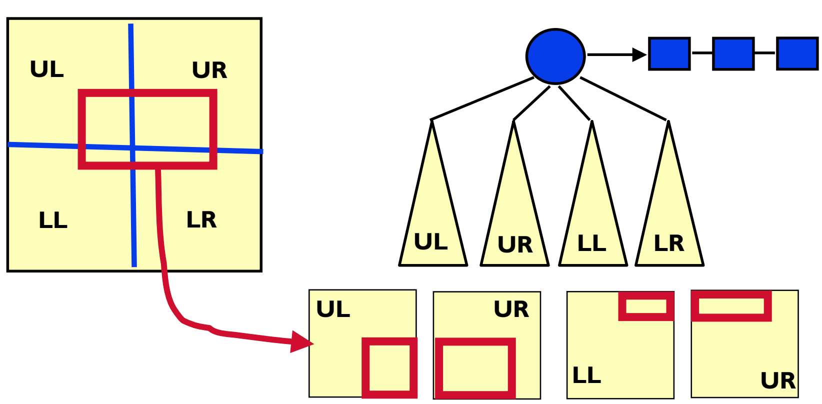
Quad Tree Query: Pick

- Just walk down the tree…
  - Going into the region that holds your x,y, till the tree ends
  - Look at the rectangles you find



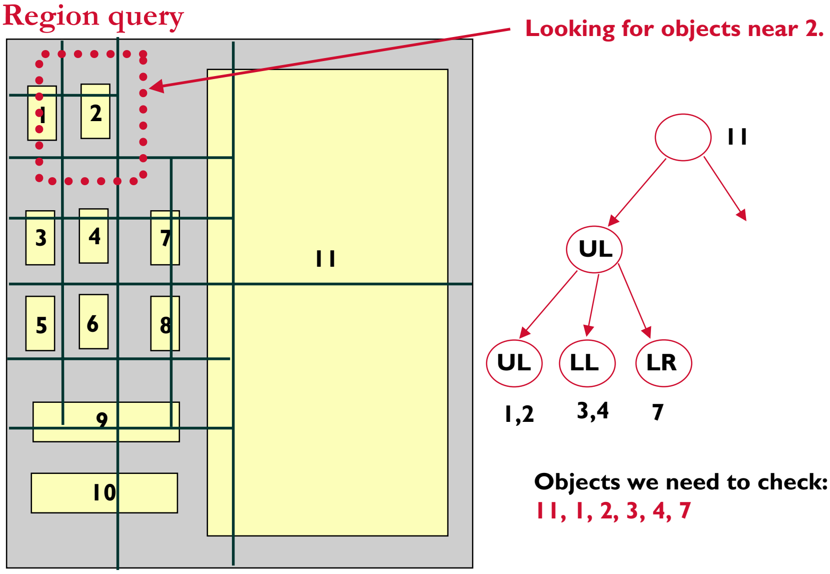
Quad Trees: Region Query

- Assume your region box hits a bisector
  - Look on bisector list first for all rectangles there
  - Then, chop up region box into (at most 4 pieces) and pass 4 new regions down tree, ie, recursively call region query 4 times on child trees



Quad Trees Queries

- Region query



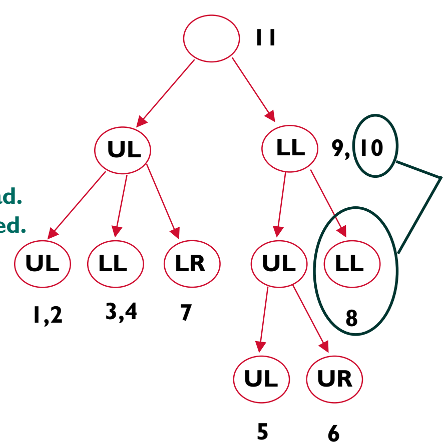
Objects we need to check:
11, 1, 2, 3, 4, 7

Data Structures: Quad Trees

- Insert and delete

Insert

Walk down tree to find appropriate quad. Create child if needed.



Delete

Remove object from the list and child from tree if necessary.

Data Structures: Quad Trees

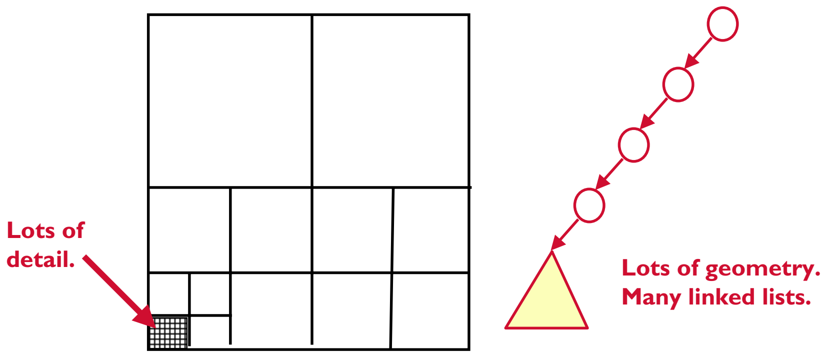
- Lots of variants and parameters
  - In particular, you can be more clever about bisector list structure
- How many rectangles per leaf node of tree?
  - One. — Called *perfect quad tree*. Easy search but huge tree. (Not realistic…)
  - Not less than K. — Called an *adaptive quad tree*. Smaller trees but with longer lists/node. Everybody does something like this now…
- How small can a quad in tree be?
  - Not less than area A. — We do no quad division if region is too small; use linked list of objects at leaves. Another adaptive sort of a tree. Smaller trees but lists may be long.
- Use these ideas to tune the tree to the problem

Data Structures: Quad Trees

- Summary
  - Good for non-uniformly distributed data.
  - Not a balanced tree.
- Complexity:
  - Time:
    - Find O(log N)
    - Insert O(log N)
    - Delete O(log N)
  - Memory:
    - O(N) - worst case, 4N tree nodes (perfect quad tree)

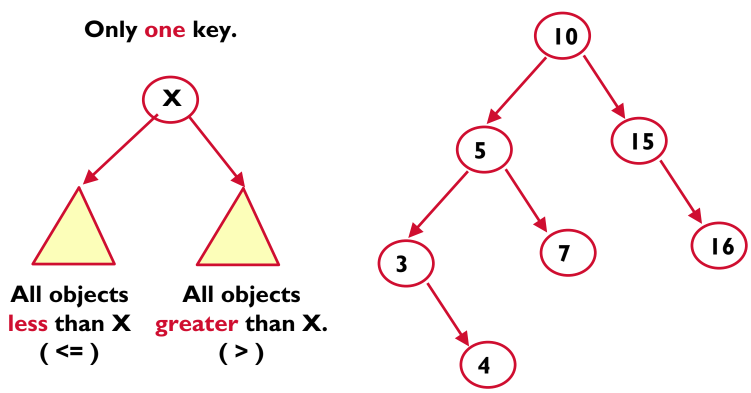
Data Structures: k-d Trees

- **Problems with basic Quad Trees**
  - **Quad trees do the right thing in that they cut up the layout area into fine bins only where needed.**
  - **However, if there are a few spots of fine detail those areas suffer from the same slow search problems as with bins.**



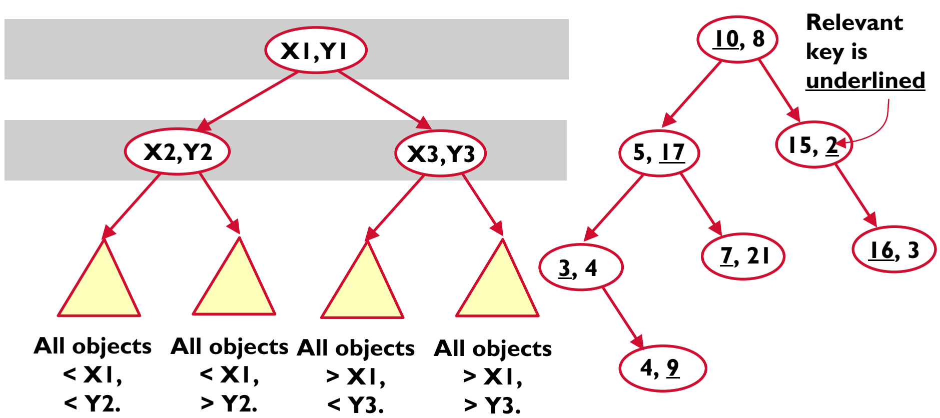
Data Structures: k-d Trees

- ***k-d tree* is multidimensional binary tree**
  - **k = number of keys used in comparison, hence "k dimensional"**
  - **Probably most familiar with 1-d tree:**



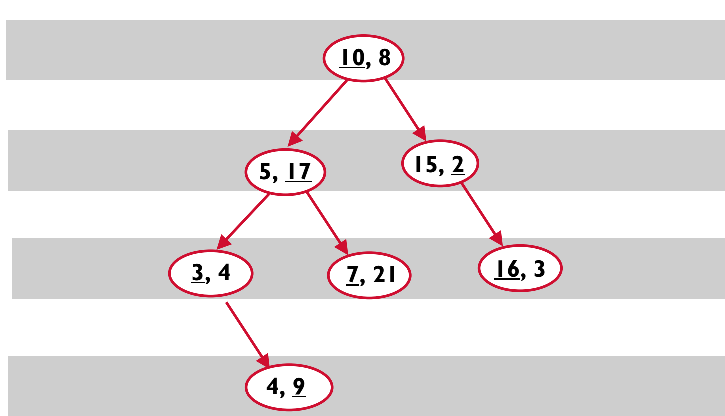
Data Structures: k-d Trees

- **Consider a 2-d tree.**
  - 2 keys in each data item (k1, k2)
  - This is a decision tree in which you look at a different key at each level of the tree.



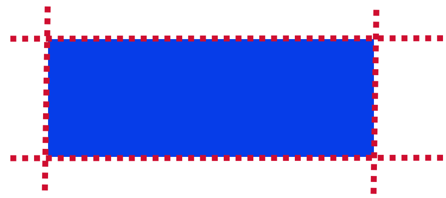
Data Structure: k-d Trees

- **Key-comparison alternates, repeats down tree**
  - Level 1: look at k1. Level 2: look at k2. Level 3: k1 again. Level 4: k2
  - At level n you look at key n mod 2



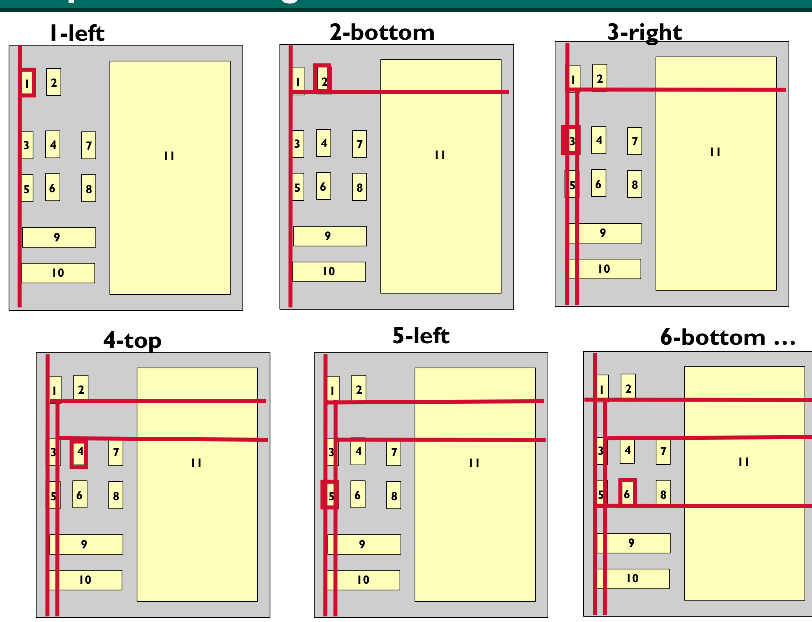
Data Structures: k-d Trees

- We'll use 4-d trees to organize rectangles:
- Keys are:
  - k1 = left edge (x min)
  - k2 = bottom edge (y min)
  - k3 = right edge (x max)
  - k4 = top edge (y max)



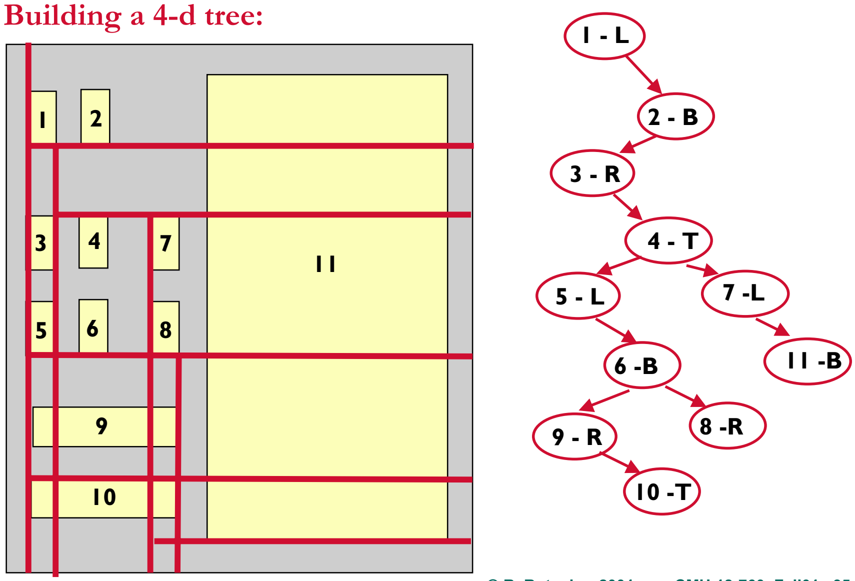
- Tree structure
  - Builds a binary tree, but at level i mod 4, we look at key $k_{(i \bmod 4)}$ to decide which child to pass the rectangle to.
- New idea
  - *Partitions space in a way that is dynamically dependent on the data.*
  - Each new rectangle inserted in tree defines a new bisector-type cut thru the chip surface.

Example: Building k-d Tree



Data Structures: k-d Trees

- Building a 4-d tree:



k-d Tree Ideas

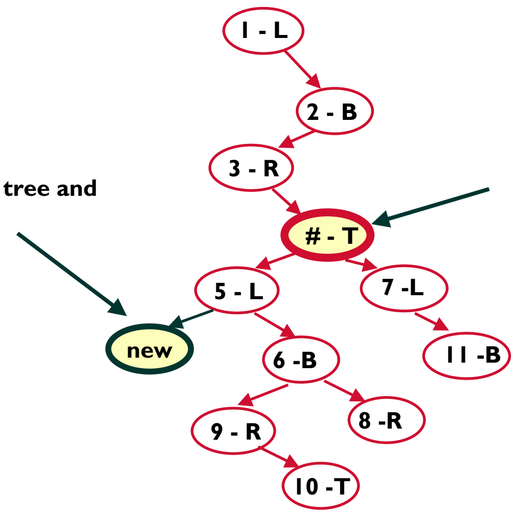
- Space still chopped up like quad tree
  - But the partition *is not static like* a quad tree
  - *It depends on the data*
  - Each new rectangle defines another cut line
- Pro
  - You get finer partition where you need it
  - Cuts adapt to data
- Con
  - Cuts depend *entirely* on the insertion order of the data
  - If you insert things in a bad order, you can still get lousy partition
  - Can insert easily, cannot delete easily...

Data Structures: k-d Trees

Insert and delete

Insert

Walk down tree and add a child.



Delete

Cannot. Rest of tree depends on this key! **Keep the keys and just mark object pointer as void. Basically, you have to garbage collect and rebuild tree later**

Data Structures: k-d Trees

Summary

- **Useful if data is pretty static and you want to mostly do queries**
- **Bad for highly dynamic data (lots of insertions and deletions).**

Historically

- **Theoretically hot in the 80s, today not used as much anymore**
- **Recent variants of quadtrees that allow each rectangle to be *redundantly* stored in several places in the tree do very well here; they've basically displaced k-d trees today**

Complexity

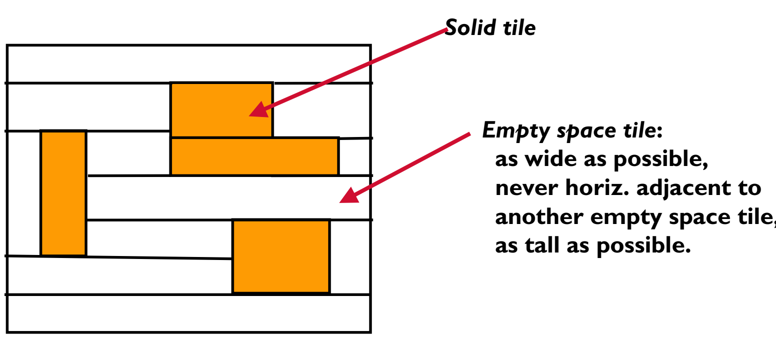
- **Time:**
  - **Find** **O(log N)**
  - **Insert** **O(log N)**
  - **Delete** **O(log N)**
- **Memory:**
  - **O(N)** **- exactly one node per object.**

Data Structures: Corner Stitching

- **Very different alternative**
  - Not a tree or a bin
- **Big ideas**
  - All space, both occupied & empty space, is explicitly represented.
  - All layout area is tiled with nonoverlapping rectangles:
    - object space = object tiles
    - empty space = empty tiles
  - Tiles are stitched together at the corners.
- **New: *Canonical representation***
  - Given a layout of objects, there is only one space tile representation
  - Makes searching & editing easier: you know how things *must* be organized

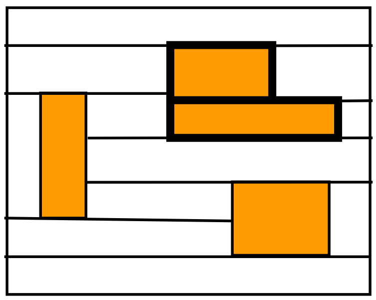
Data Structures: Corner Stitching

- **The rules:**
  - 2 tile types: *space*=empty, *solid*="something there"
  - *Maximal horizontal strips:* every *space* tile must be as *wide* as possible
  - No space tile can have another space tile to the left or right of it
  - Each space tile is as tall as it can be without violating above rules.

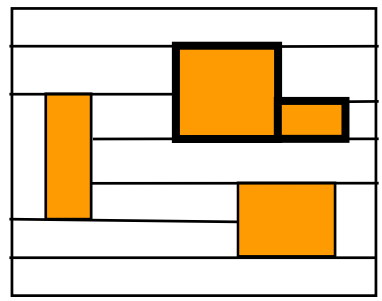


Aside: About Canonical Form

- Tile planes represent *empty space* in a canonical way
  - **But, as originally presented, they don't represent "solid" canonically**



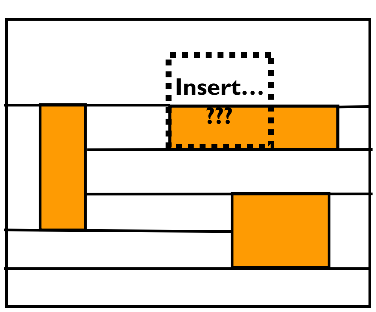
Legal tile plane



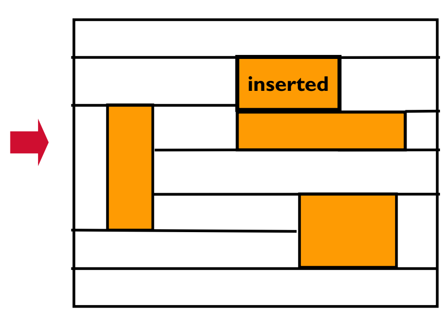
Also a legal tile plane; Note only space tiles are same here…

Aside: About Canonical Form

- Actual tiles may depend on order of insertion of solid tiles
  - **In the original version, you "chop up" the new solid tile into pieces that only overlap space tiles in the layout, then insert each of these pieces**
  - **This is how Ousterhout does it in the paper…**



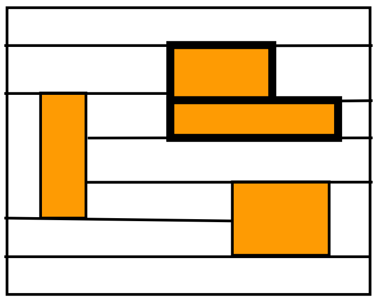
Starting tile plane



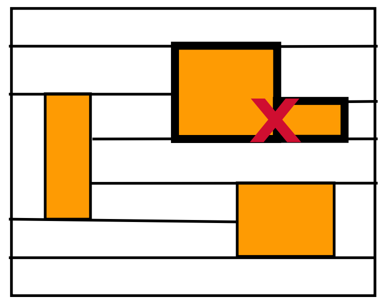
After insertion

Aside: About Canonical Form

- Can *also* insist on canonical form for the solid tiles
  - Just like space tiles: they are maximally wide, never have same-color tiles at left or right, and after satisfying this, they are maximally tall



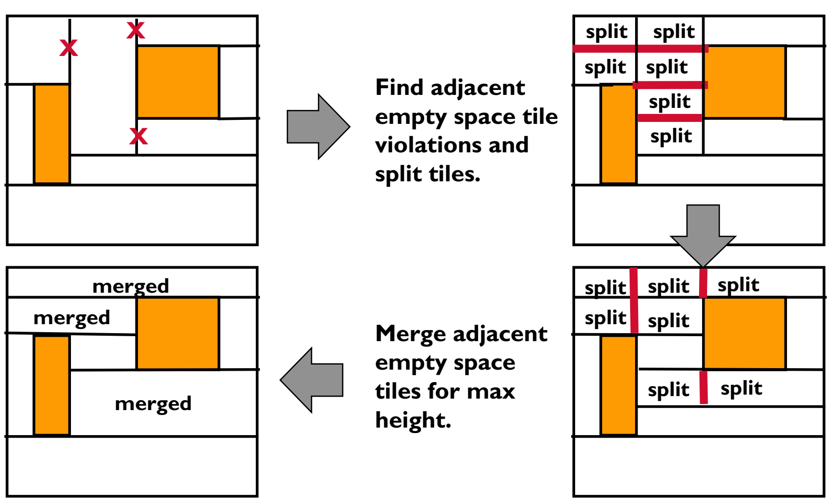
If solids use same rules as space tiles, only this is legal



Illegal tile plane: solids not max wide

Repairing Violations of Canonical Form

- Always same pattern: split the bad tiles, re-merge them correctly



Find adjacent empty space tile violations and split tiles.

Merge adjacent empty space tiles for max height.

Corner Stitched Tile Planes

Ideas

- Canonical representation for "one layer" of stuff on an IC
- Example: metal2, or polysilicon
- Just like BDDs: all algorithms that manipulate a tile plane are required to keep it -- *at least space tiles* -- in *canonical* form after the processing

Manipulations

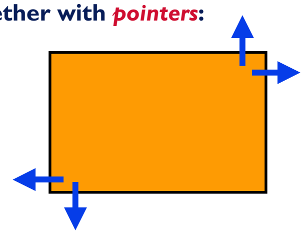
- Insert a new tile
- Delete a tile
- Point search
- Region search

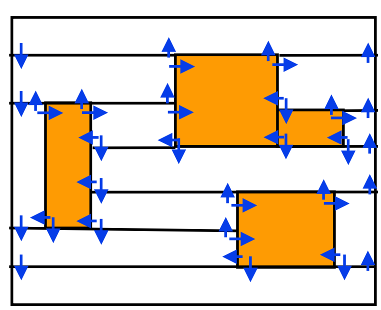
Corner Stitching Tile Plane: Implementation

Stitches at 2 corners of each tile

- *Can* put stitches at all 4 corners; but only *need* them at 2 corners

Corners are stitched together with *pointers*:





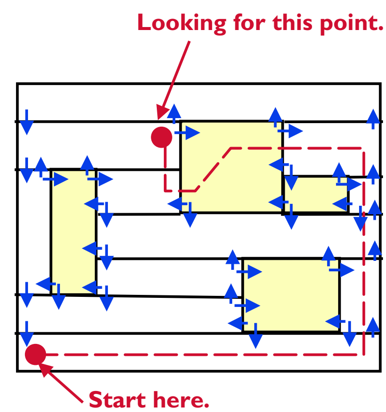
Only know about other tiles that are near the upper right and lower left corners.

Only know some of your adjacent neighbors directly

Tile Planes: Point Search

- **Point Search**
  - Given x,y
  - Search horizontally to find a tile the bounds the x coord.

    (*Bounds* means "overlaps")
  - Search vertically to find a tile that bounds the y coord.
  - Repeat till you hit a tile the contains that point



Tile Planes: Region Query

- **Region search**
  - Ousterhout gives a neat recursive enumeration algorithm
- **Basic idea**
  - Visit tiles in region top to bottom, left to right (sort of...)
  - Each tile is "owned" by just other tile, which calls the recursive "enumerate" algorithm on that tile.
  - If your neighbor tile's lower left corner touches "you", you "own" that tile and recursively call the enumerate algorithm on that tile
  - Or, if you and your neighbor tile are both at the bottom of the region, you "own" him and call enumerate on him
  - Each tile visited many times, but enumerated just once.

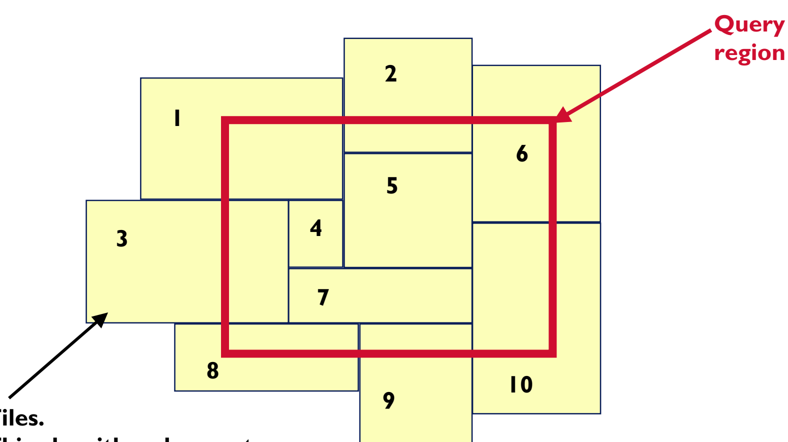
Outerhout's Algorithm

```
R = region
Find all tiles on left edge of the region R
for (each of these tiles T on left edge of R)
    enumerate( tile T)


enumerate( tile T )
{
   Put tile T on list.
   If (right edge of T out of region R)
              return.

   Find all tiles touching right side of tile T
   For (each of these neighbor tiles S)
              call enumerate( neighbor tile S )
   return
}
```

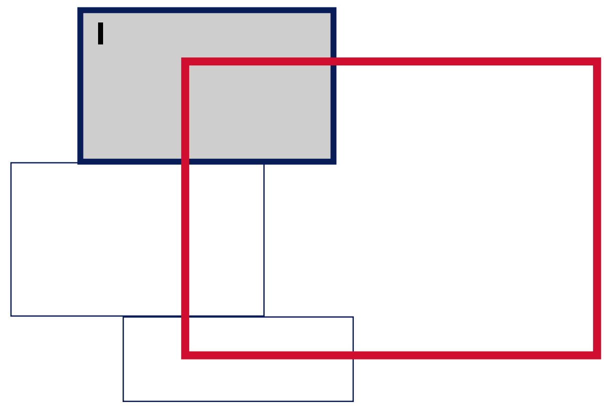

Ousterhout's Example



Tiles.
This algorithm does not actually care whether a tile is "space" or "solid"

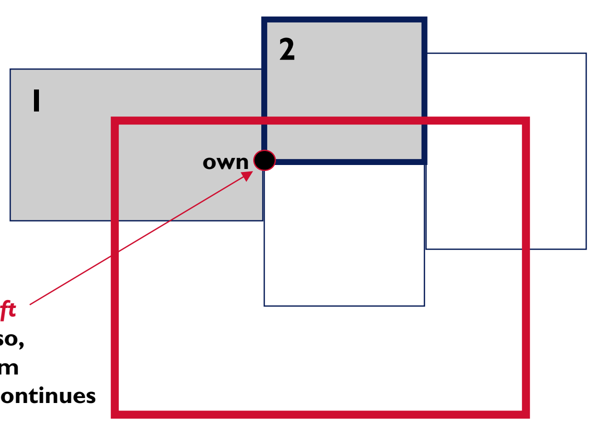
Ousterhout's Algorithm

- Find tiles on left side of region R
  - Find top tile, call it "1"
  - Call enumerate (tile 1)



Ousterhout's Algorithm

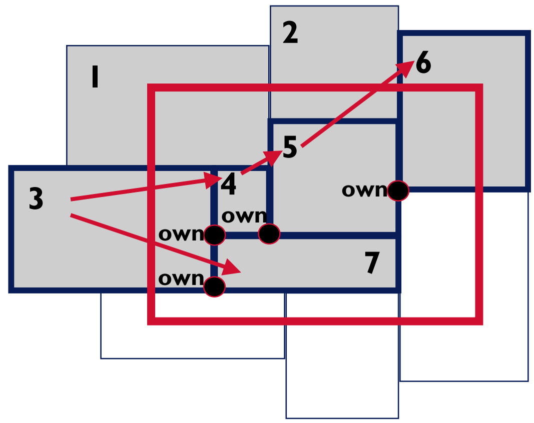
- enumerate(tile 1)
  - you (tile 1) *own* tile 2.
  - tile 2 does *not* own its right neighbor, so quit



In this recursion, a parent tile "owns" an adjacent neighbor tile if the *lower-left* corner touches parent. If so, parent recursively call enum on this neighbor; process continues

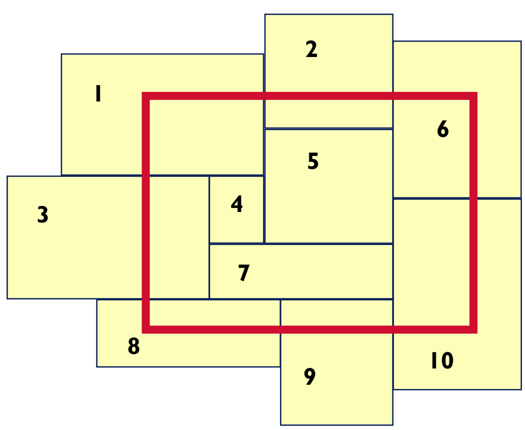
Ousterhout's Algorithm

- Back up to tile 1, down to new tile 3
  - 3 owns 4. call enumerate(4)
    - 4 owns 5. call enumerate(5)
      - 5 owns 6. call enumerate(6)
  - 3 owns 7. enumerate(7)
  - Continue…



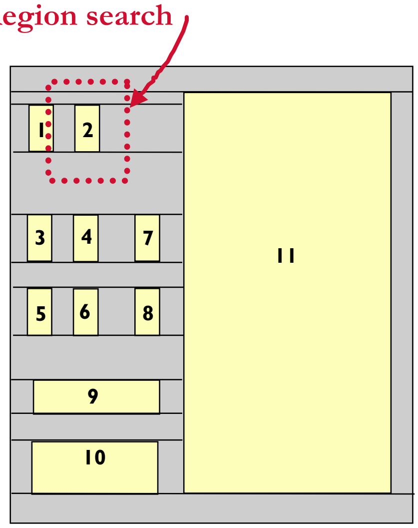
Ousterhout's Algorithm

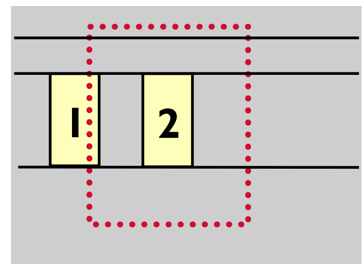
- Recursively touches tiles in this order
  - Can return these tiles to the region query as the relevant ones



Tile Plane Queries

- Region search

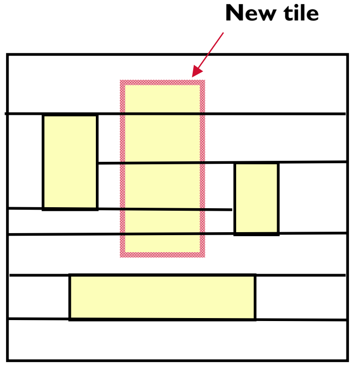




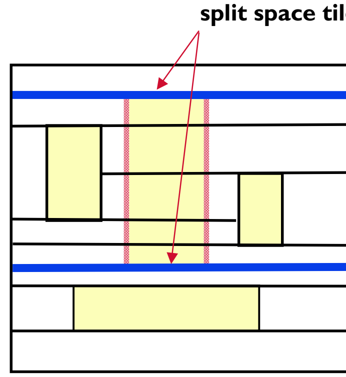
Objects Found

Tile Planes: Tile Insertion

- Insertion is complex; simplest case is solid tile touching only space tiles



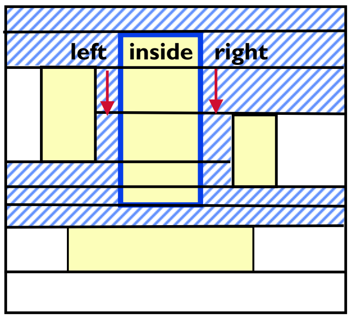
Use point search to find the existing tiles in the location of the new tile.



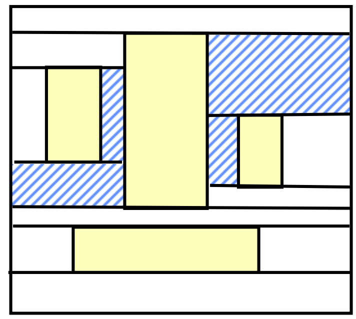
Split space tiles containing the top and bottom edges of the new tile.

Tile Insertion

- **Insertion**



Walk the left and right edges, splitting tiles into left, inside, and right tiles as needed. (Stripes tiles are all the ones you will touch as you split)



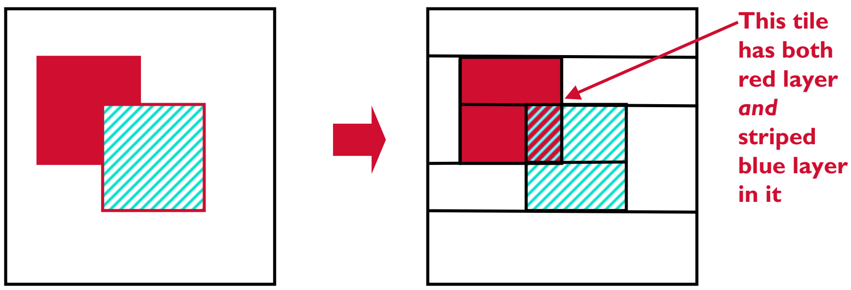
Merge like-space tiles vertically wherever possible. (Stripes tiles show space tiles that result from this vertical merging)

Tile Deletion

- **It is even messier…**
  - Have to find all the appropriate neighbors
  - Have to merge all the appropriate neighbors to maintain canonical structure of tile plane
  - See the Ousterhout paper
- **Roughly speaking**
  - You find all the tiles in the region you want to delete
  - You split ALL the tiles you find on ALL the possible edges of ALL tiles
  - You then “glue” back together all these little chopped up pieces to reassemble the tile plane

Practical Usage: Multi-Layer Tile Plane

- You usually want more than just "solid" and "empty" tiles
  - Imagine you want in *one* tile plane *all* the layers in, say, a FET: poly, diffusion, well, metal1, contact-cut
  - The way to do this is to "multiply-fracture" the tile plane
  - Idea is each tile gets a unique "stack" of material types
  - Can label each tile with a bit vector, bit Bi =1 is layer i is present



Data Structures: Corner Stitched Tile Planes

- Summary
  - Easy to do local modifications.
  - Adapts to sparse and dense layout sections.
  - Manhattan geometry mainly (you *can* do 45-degrees using trapezoids).
  - Can either have parallel tile planes to handle multiple layers, or do "multi-layer tile plane" directly (a more complex implementation)
- Complexity
  - Time:
    - Find O(sqrt(N))
    - Insert O(1)
    - Delete O(1)
    - worst case for all O(N)
  - Memory:
    - O(N) - upper bound on empty space tiles is 3N + 1

Summary

- Many different structures to handle geometry
  - **Bin:** simple, static, nonhierarchical decomposition of space
  - **Quad tree:** static hierarchical decomposition of space
  - **k-d tree:** dynamic hierarchical decomposition of space
  - **Tile plane:** dynamic, local, canonical nontree decomp. of space

- All support basic operations, with different tradeoffs
  - **Pick:** find me objects that touch x,y
  - **Region query:** find me all objects that touch this box
  - **Insert:** add a new rectangle or polygon to structure
  - **Delete:** remove geometry (or a whole region) from structure