INFS4205/7205 Advanced Techniques for High Dimensional Data

Spatial Data Organization I

Semester 1, 2021

University of Queensland

+ Assignment 1

- Due 21 Apr 21, 4pm
- Blackboard submission
- Contents
  1. Database Design
  2. Spatial Query
  3. Query Execution
- No implementation, No actual code, No standard solution…
  - Design your own from scratch
  - Plain English description / Pseudo-code
  - Clear and concise

Advanced Techniques for High Dimensional Data

- Course Introduction
- Introduction to Spatial Databases
- Spatial Data Organization
- Spatial Query Processing
- Managing Spatiotemporal Data
- Managing High-dimensional Data
- Other High-dimensional Data Applications
- When Spatial Temporal Data Meets AI
- Route Planning
- Trends and Course Review

+ Learning Objectives

- What we will cover
  - Basic principles of managing multidimensional data
    - 2-D as example
    - How to construct index
  - Representative data access methods for point, line and polygon data
    - How to use index
  - Processing of some spatial operations using data access methods
    - Point query, Range / Window query, …
- Goals
  - Understand major types of multidimensional data access methods, and their strengths and limits
  - Understand how point query, window query and join query are processed using various data access methods

+ Readings

- R. Güting, An Introduction to Spatial Database Systems, *The VLDB Journal*, 3:4, 1994
- Oracle Business and Technical White Papers
- V. Gaede and O Günther, Multidimensional Access Methods, *ACM Computing Surveys*, 30:2, 1998
- J. Orenstein and F. Manola, PROBE Spatial Data Modeling and Query Processing in an Image Database Application, *IEEE Transactions on Software Engineering*, 14:5,1988
- T. Brinkhoff, H.-P. Kriegel and B. Seeger, Efficient Processing of Spatial Joins Using R-Trees, SIGMOD'93

+ Index

INDEX

+ Index

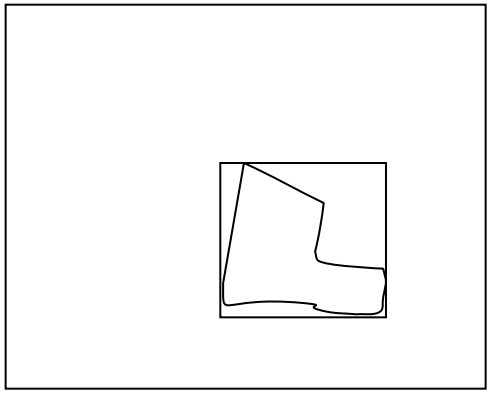
- A database index is similar to a book index
- Book index: lists important terms in alphabetical order with a list of page number(s) where the term appears
- Searching in a book:
  1. Search the book index for a term to find a list of addresses (i.e., page numbers)
  2. Use these addresses to retrieve the specified pages
  3. Search for the term in each page
- Otherwise, search through the entire book word by word (≈linear search)

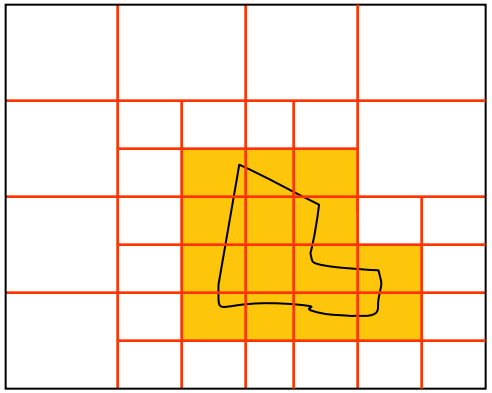
Spatial Indexing

- Purpose:
  - Efficiency in processing spatial selection, join and other spatial operations
- Two strategies to organize space and objects
  - Map spatial objects into 1D space and use a standard index structure (B-tree)
  - Dedicated external data structures
- Basic ideas
  - Approximation
    - Bounding box, Grids
  - Hierarchical Data Organization

Object Approximation

- A fundamental idea of spatial indexing is the use of approximation
- Continues Approximation
  - Object centric
  - Example:
    - Use of MBRs (Minimum Bounding Rectangles)
    - R-Tree
- Grid Approximation
  - Space centric
    - Faster mapping
    - Uniform / Non-uniform
    - High-D?
  - Example:
    - Quad-Tree





What We Need to Know

For each multidimensional access method:

- Motivation
  - Why it is proposed
  - Good for what type of data
    - Points? Line? Polygons?
- Operations for creating and maintaining an index
  - Insert, delete data items
- Query operations using an index
  - Point query and window query
  - Spatial join queries and other queries

+ Background Knowledge

- Disks and files
- Basic indexing methods in relational DBMS
  - B-Tree
  - Hashing
  - Bitmap
- Query processing using indexes
  - What to achieve, and how?

+ Secondary Storage Device

- Magnetic Tape
  - Still in use???
    - Cheap per GB: 2 cent
    - Better durability: 50 years or longer
    - Suit for backup of PB historical data
      - Telecom
    - Energy Efficient
  - Fast sequential read/write
    - LTO 8: 12TB 360MB/s
    - LTO 9: 25TB 708MB/s
    - LTO 10: 48TB 1100MB/s

IBM TS4500 Tape Library

Next-generation cloud storage library.

- Maximum capacity with LTO-8 / TS1160: 278 PB / 351 PB
- Drive Type(s): LTO and/or TS1100
- Maximum number of drives: 128

+ Secondary Storage Device

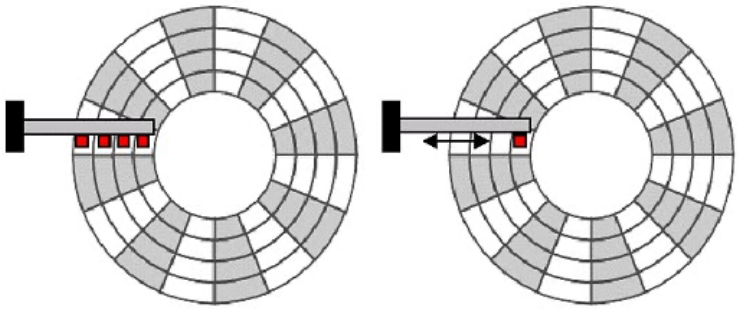
- (S)DBMS stores information on hard disks
- This has major implications for DBMS design:
  - READ: transfer data from disk to main memory
  - WRITE: transfer data from RAM to disk
    - In the unit of block
      - 512, 1K, 4K, 64M,…
  - Both are high-cost operations (I/O), so must be planned carefully!
    - External Memory Complexity: The number of I/Os that need to be performed
- Why not store everything in memory?
  1. Costs
  2. Main memory is volatile
     - PCM: Phase-Change Memory

Magnetic Disks

- Main advantage over tapes
  - Random access
- Data is stored and retrieved in units called disk blocks or pages or buckets
- Unlike RAM, time to retrieve a page varies depending upon location on disk

+ Accessing a Disk Block

- Time to access (read/write) a disk block:
  1. **Seek Time:** Moving arms to position disk head on track
  2. **Rotational Delay:** Waiting for block to rotate under head
     - 5400 / 7200 rpm
  3. **Transfer Time:** Actually moving data to/from disk surface
- Seek Time and Rotational Delay dominate
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?



+ Index Structure

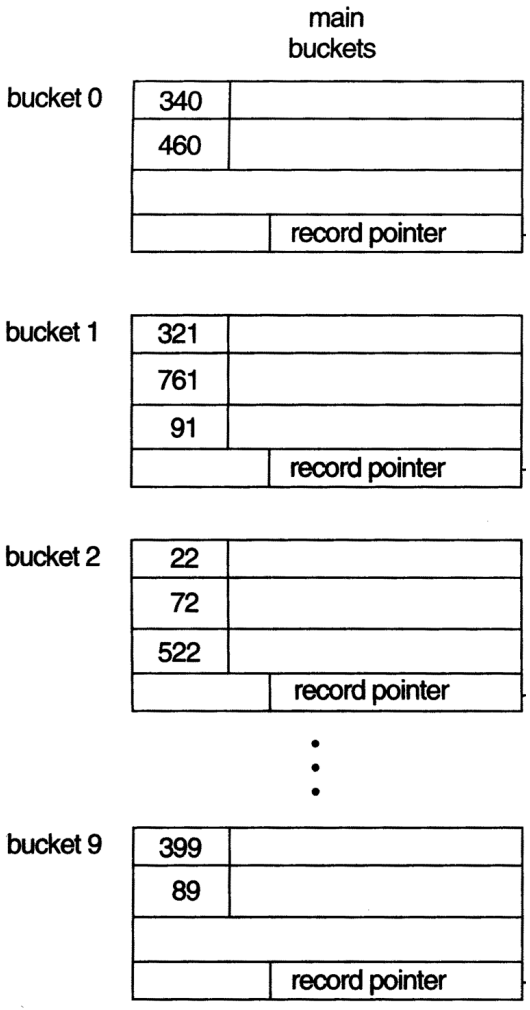
- An index is an **auxiliary** file that makes it more efficient to search for a record in the data file
- An index is usually specified on one field of the file (although it could be specified on several fields)
  - For another field? Build a new index for it
- One form of an index is a file of entries:

  <field value, pointer to record>
  - ordered by field value
- The index file occupies **less disk blocks** than the data file
  - Because its entries are much smaller
- A binary search on the index yields a **pointer** to the file record
  - Extra I/O access is needed to fetch data

+ Data Access Methods

- One dimensional
  - Hashing and B-Trees
- Line Data
  - Segment Tree, Interval Tree
- Point data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: point and region quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon data
  - Transformation: End point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R^+-tree

Static Hashing



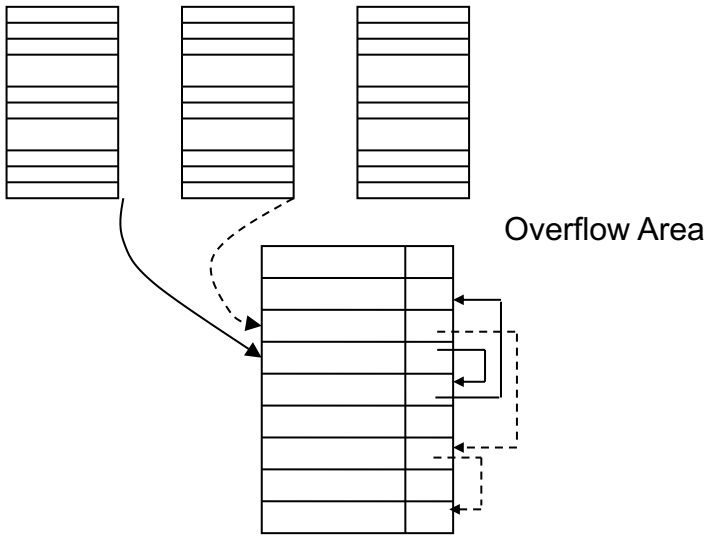
- **Hashing** converts the key of a record into an address in which the record is stored
  - Like a dictionary
  - Why not an array with direct addressing?
    - $O(1)$ worst case
    - Key number $k \ll N$ possible keys, waste space

Static Hashing

- Hashing converts the key of a record into an address in which the record is stored
- A hash function is used to map the key to the *relative* address of a bucket in which the record is stored.
  - If a file is allocated m buckets, the hash function must convert a key k into the relative address of the block:
    - $h(k) \in \{0, \ldots, m-1\}$
- A bucket is either one disk block or a cluster of contiguous blocks
- A table stored in the header of the file maps relative bucket numbers to disk block addresses

+ Collision

- Insertion of a new record may lead to collision
  - No space in $b = h(k)$
  - Data Skewness
- Chaining: Use overflow buckets
  - Example: Common overflow area for all blocks in a file
    - Each block has a pointer to its first record in the overflow area
    - Records belonging to the same block are linked by pointers
    - Hash could decay into linked list
  - Distribution does not meet expectation



Hashing Functions

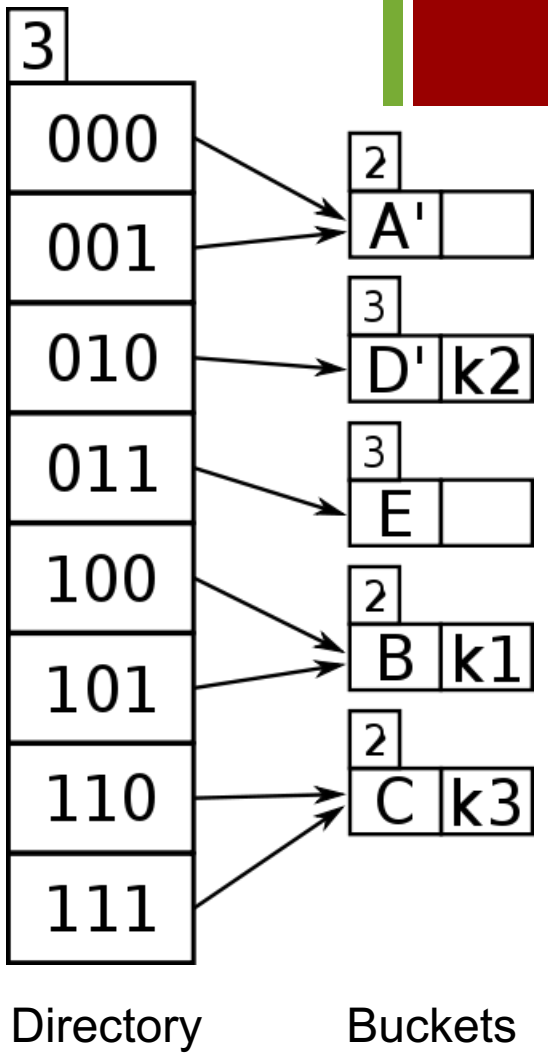
- A good hash functions must
  - Be computed efficiently
  - Minimize the number of collisions by spreading keys around the file as evenly and uniform as possible
- Example of hash functions
  - Truncation
    - Ignore part of the key and use the remaining part directly as the index
    - e.g. key is an 8-digit number, hash table has 1000 entries
      - Use the first, fourth, and eighth digit to make the hash function
      - 12345678 maps to 148
    - Not uniformly
  - Folding: Break up the key in parts and combine them in some way
    - 12345678 maps to 123+456+78=657
  - Division: $h(k) = k \ mod \ m$

+ Pros and Cons of Hashing

- Excellent performance for searching on equality on the key used for hashing
- Records are not ordered
  - Any search other than on equality is very expensive
  - e.g. Range Query
- Prediction of total number of buckets is difficult
  - Allocate a large space
  - Estimate a “reasonable” size and periodically reorganize

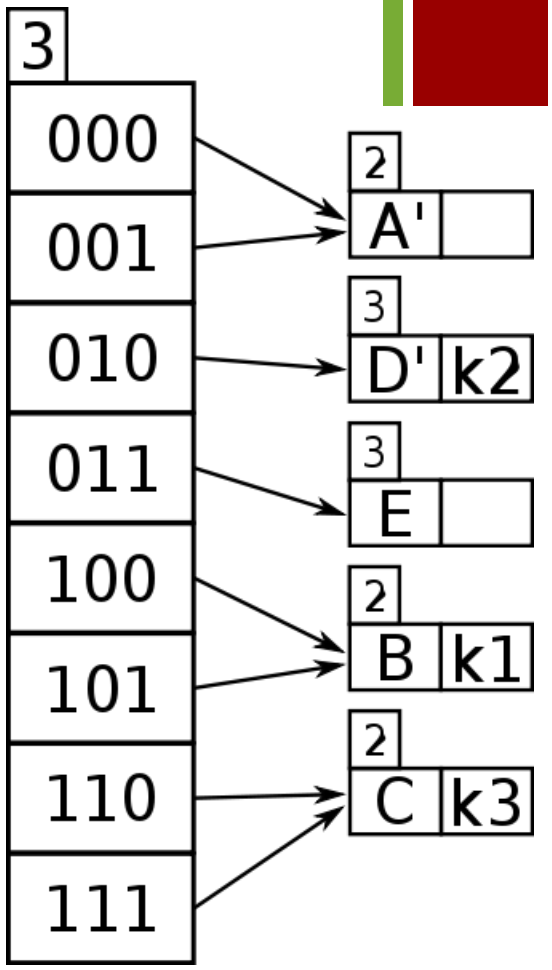
+ Extendible Hashing

- The file is structured into two levels:
  - Directory and Buckets
  - The directory has 2^d entries (d: global depth)
    - Each entry points to a bucket
    - Use some hashing function that generates a string of bits
    - The first/last d digits used as index into the directory
      - Prefix: Only the first d' bits are used
      - Suffix: mod operation, the last d' bits are used
    - Several entries can point to the same bucket
  - Local depth d' ($d' \leq d$)



+ Extendible Hashing Adjustment

- The directory size can be doubled or halved
  - Double: $d = d + 1$
    - When a bucket with $d' = d$ overflows
  - Half: $d = d - 1$
    - When $d > d'$ for all buckets
  - No directory size change
    - $d' = d - 1$
    - Split the bucket when overflow
- At most two-level search, highly efficient
  - No *just-in-case* space waste, no overflow page, no chain search
  - No space is allocated in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed



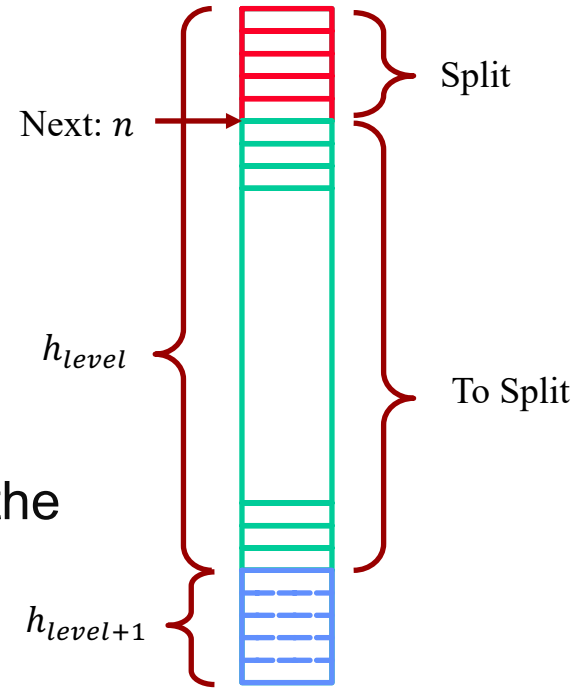
Example in Tutorial

Linear Hashing

- Dynamic hashing scheme
  - The file size grows linearly, bucket by bucket
  - No directory
    - The file starts with m main buckets
    - Buckets are numbered from 0 to $m-1$
  - A family of hash functions $h_0, h_1, h_2, \ldots$
    - $h_i(k) = k \bmod (2^i \times m)$
    - Initial hashing function $h_0(k) = k \bmod m$
    - h_{i+1} doubles the range of h_i
  - Still has overflow
    - Split only when the $load\ factor = \frac{|item|}{|bucket\ capacity|}$ is met
      - Otherwise, use overflow page
    - Split the buckets with round-robin
      - Round ends when all initial buckets are split
      - Only two neighboring hash functions exist at the time

Linear Hashing

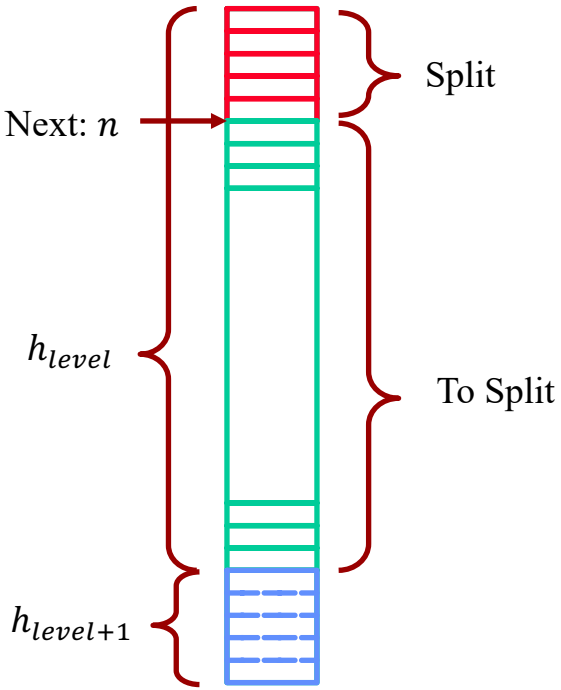
- We keep the following info:
  - n: a pointer to the bucket that should be split next
  - Initially, $n = 0$
    - 0 to n have been split
    - n to $2^i \times m$ to be split
- Search
  - $h_{level}(k)$ is larger than n
    - Search inside the bucket
  - Otherwise, also apply $h_{level+1}(k)$ to find out the exact location



+ Insertion of a Record

If there is a collision, do:

- Push record to an overflow bucket
- (Grow) A new bucket is appended at the end of the hash table
- (Split) Records in bucket number n (including those in the overflow space) are hashed again using
  - $h_1(k) = k \bmod (2m)$ (will return either n or $m + n$)
  - These records will either remain in bucket n or in bucket $m + n$
- If $n = m$ (all original buckets have been split)
  - Set $n = 0$
  - $h_{1(k)} = k \bmod (2m)$
  - $h_2(k) = k \bmod (4m)$



Example in Tutorial

Linear Hashing

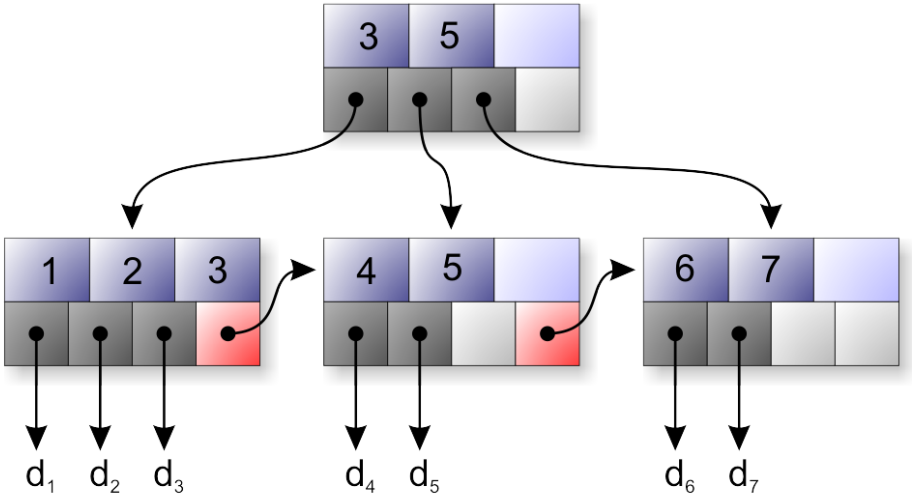
- Advantage
  - The file size grows linearly, bucket by bucket
- Disadvantage
  - No guarantee that a split relieves the overloaded bucket
  - Still requires overflow buckets and chaining

Hashing Summary

- Best for equality search, bad for range search
- Static Hashing can lead to long overflow chain
- Extendible Hashing avoids overflow by splitting a full bucket when a new data entry is added
  - Directory to keep track of buckets, doubles periodically
  - Can be large with skewed data
- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages
  - Overflow pages not likely to be long
  - Space utilization could be lower than Extendible Hashing

+ About B-Tree

- What does “B” stands for?
- How to perform a search for a given key value?
- How to insert a value?
- Any overhead?
- When to create a B-tree index, and on what attributes?



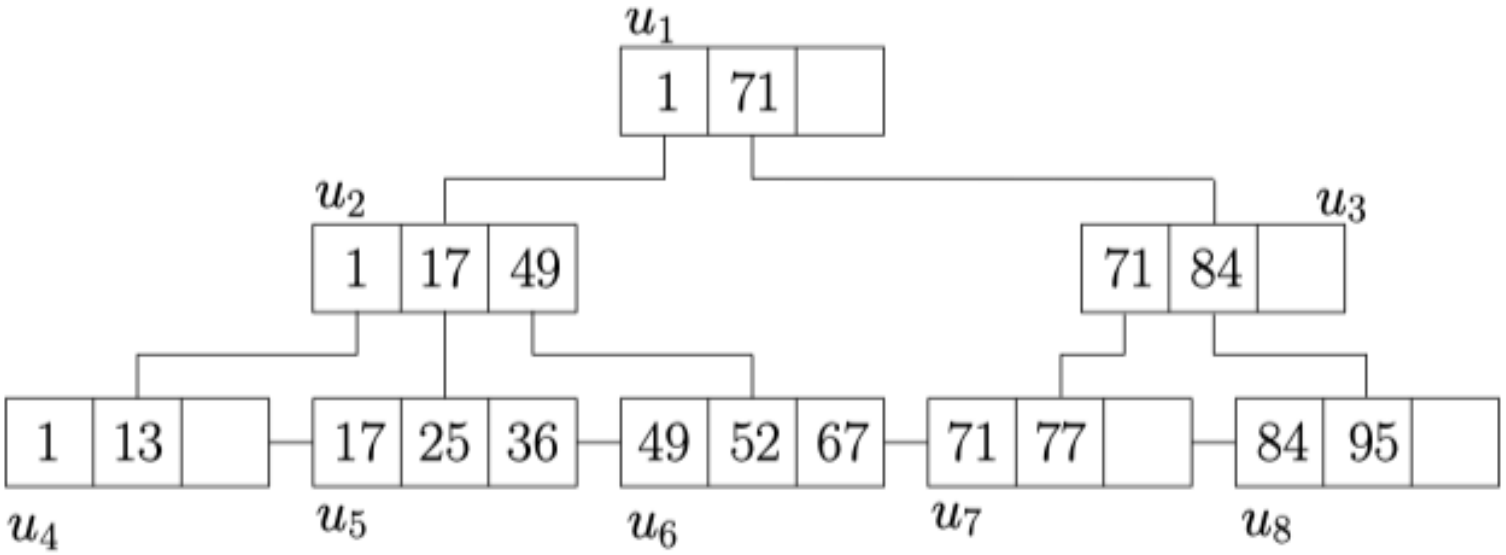
...B-tree and B^+-tree: what's the difference?

+ B-Tree

- Each leaf node has between $B/2$ and B data elements, where $B \geq 3$
- The only exception takes place when the leaf is the root, in which case it can have any number of elements
- All the leaf nodes are at the same level
  - Each internal node has between $B/2$ and B child nodes
  - Except that the root can have as few as 2 child nodes

+ B-Tree Structure

- For any node u, denote by S_u the set of elements in the subtree of u. Now let u be an internal node with child nodes $v_1, \ldots, v_f$ $(f \leq B)$
  - All the data elements in S_{v_i} must be strictly smaller than any data element in S_{v_j} for any $1 \leq i < j \leq f$
  - For each v_i $(i \leq f)$, u stores the smallest element $r(v_i)$ in S_{v_i}, which is referred to as a routing element

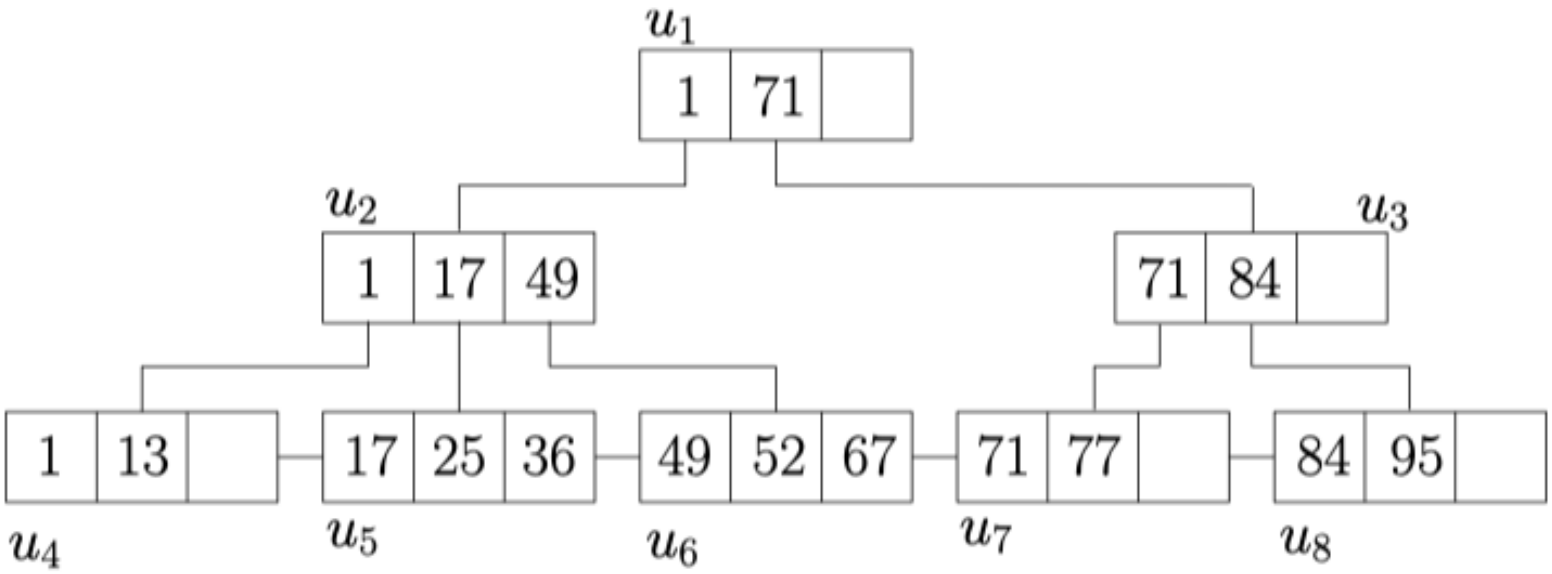


B-Tree Successor and Predecessor Search

- Successor and Predecessor Search
  - Let S be a set of real values
  - The successor / predecessor of a value x in S is the smallest/largest value in S that is at least / most x
  - Example: $S = \{1, 13, 17, 25\}$
    - Successors of 10 and 13 are both 13
    - Predecessor of 12 is 1
  - Algorithm
    1. Set u to the root of T
    2. If u is a leaf, return the successor of x among the elements in u
    3. Otherwise, find the predecessor of x among the elements in u.

B-Tree Successor and Predecessor Search

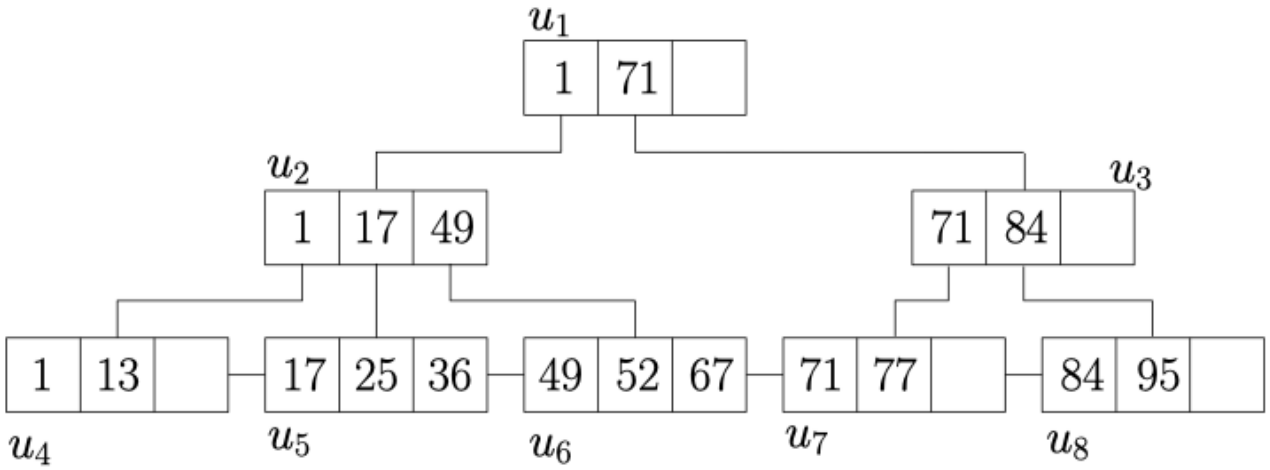
- u_1, u_2, u_5 are accessed to retrieve the successor of 20
  - Cost = 3 I/Os



B-Tree Range Query

- $q = [x, y]$
  1. Locate the leaf u containing the successor of x
  2. Report all elements in u that fall in q
  3. If no element in u is greater than y
     - Set u to the succeeding leaf node, and go Step 2

- $q = [20,75]$
  - $u_1, u_2, u_5, u_6, and\ u_7$

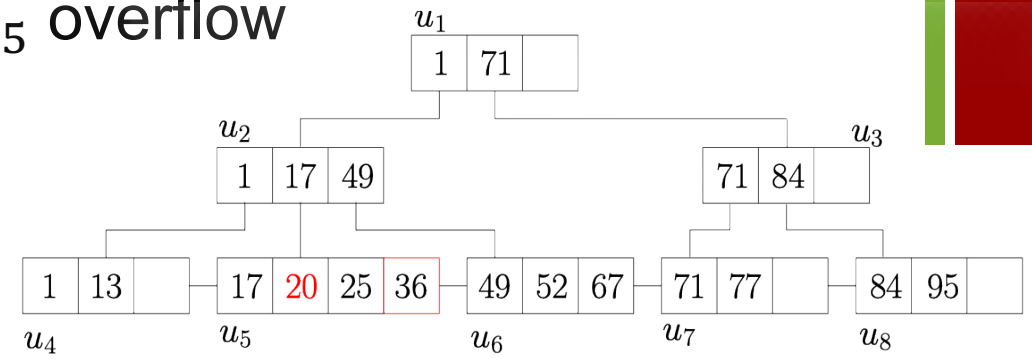


B-Tree Insertion

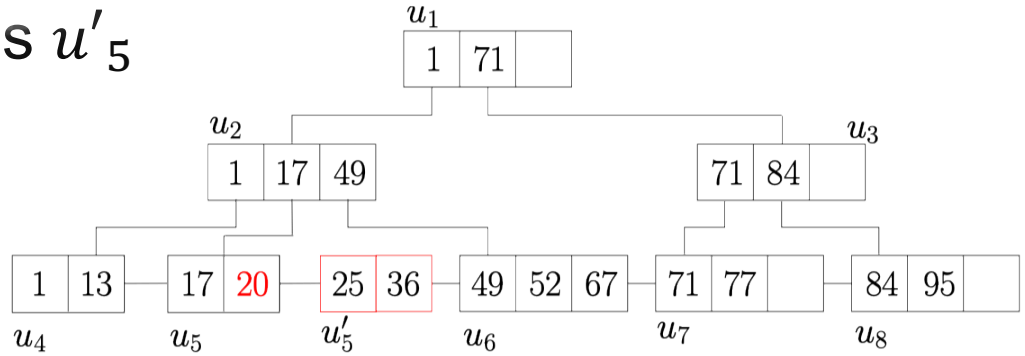
- To insert a value x into a B-Tree
  1. Find the leaf node u that should accommodate x without violating the B-Tree definition, add x to u
  2. If u has no more than B elements, the insertion is complete
     - Otherwise, u overflows
- Overflow Handling
  1. Create a new node u'
  2. Split the element of u in two halves (u and u')
  3. Insert $r(u')$ into the parent p of u
  4. If p has no more than B elements, done
     - Otherwise, hand the overflow at p in the same way

+ B-Tree Insertion

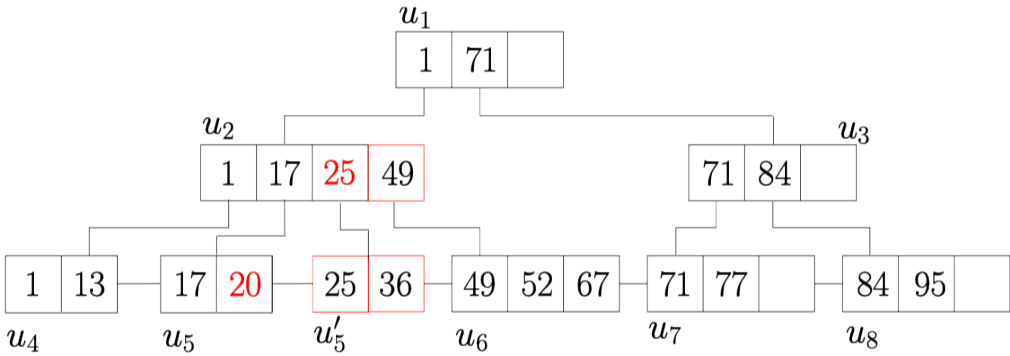
- Insertion of 20 make u_5 overflow



- u_5 splits, and generates u'_5

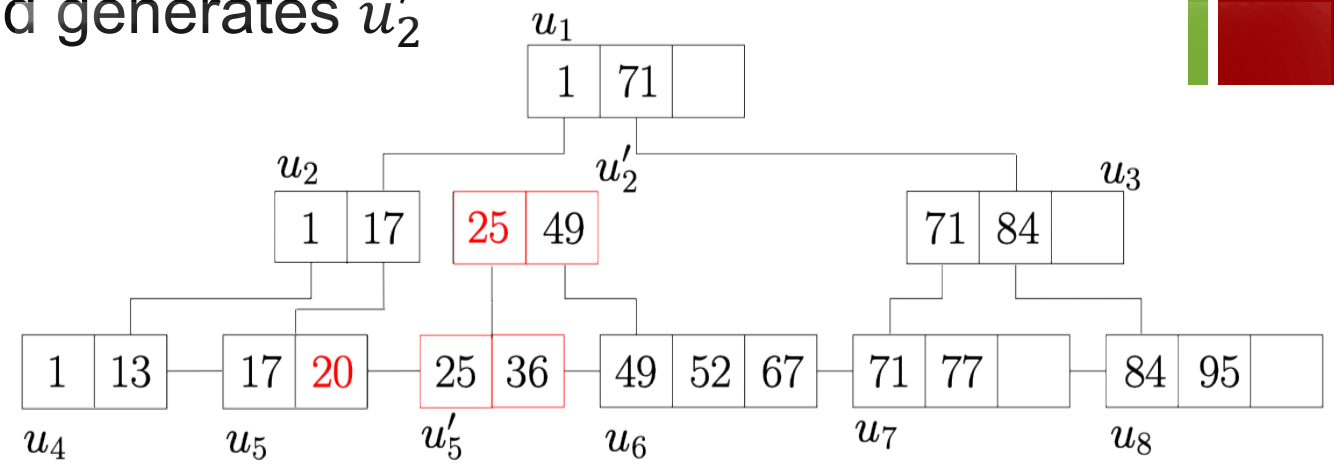


- $r(u'_5) = 25$ is added to u_2, which overflows

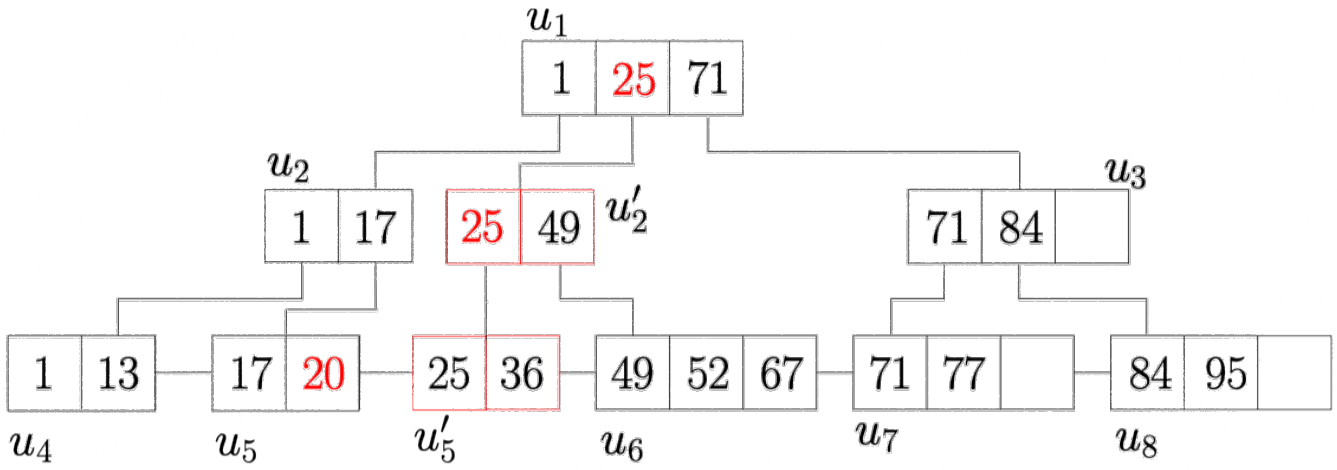


+ B-Tree Insertion

- u_2 splits, and generates u_2'



- $p(u_2')$ is added to the root, done!



+ B-Tree Deletion

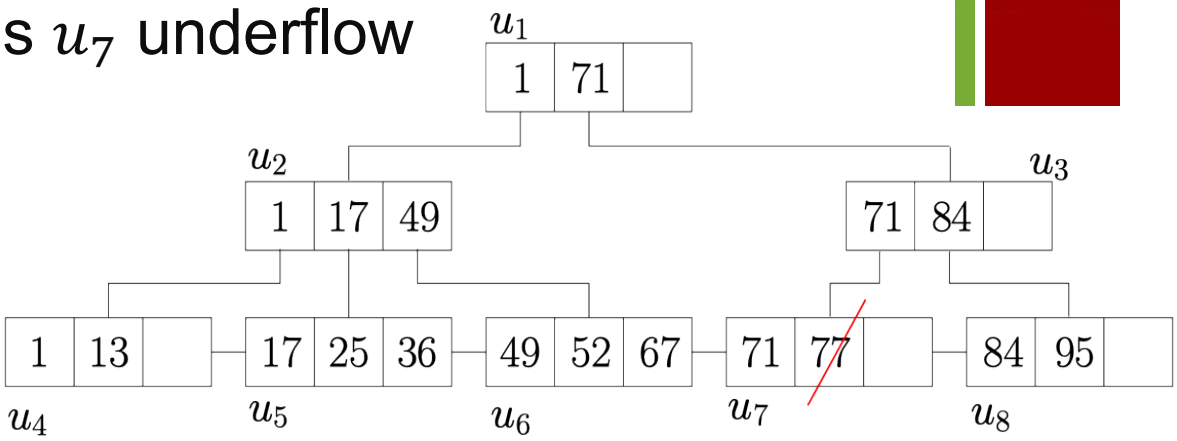
- To delete a value x from a B-Tree
  1. Find the leaf node u that contains x, remove x from u
  2. If x is the smallest element in u, adjust the routing elements in the ancestors of u appropriately
  3. If u is the root or has at least $\frac{B}{2}$ elements, the deletion is complete
     - Otherwise, u underflows

+ B-Tree Deletion

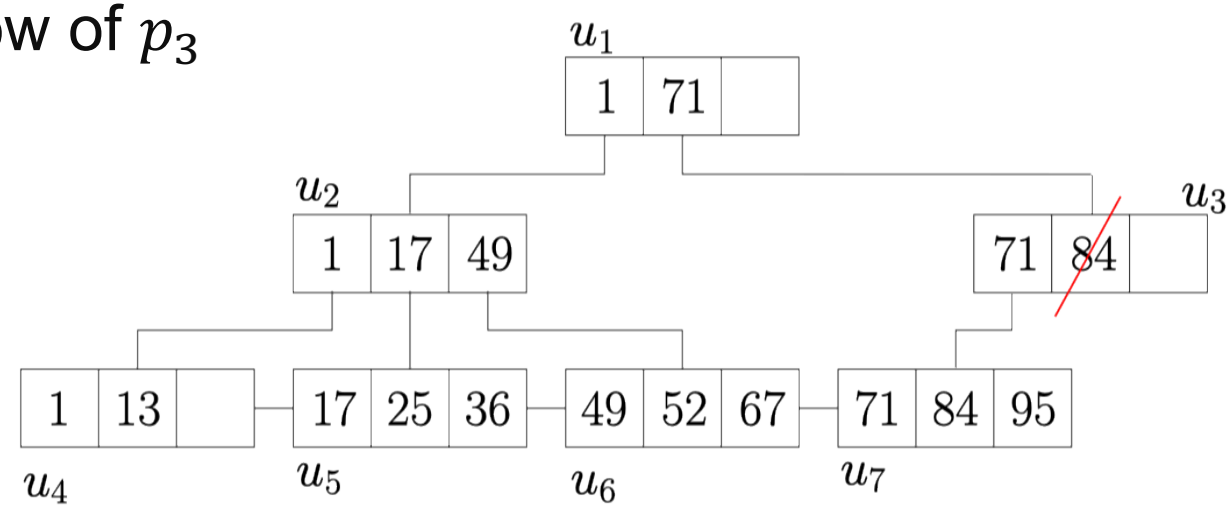
- Underflow Handling
  - Let u be the node that underflows, and u' be the right sibling of u (if u does not have a right sibling, set u' to its left sibling, and swap u and u' in the below):
  1. If u and u' contain no more than B elements in total, perform a merge
     - Put all the elements in u
     - Remove $r(u')$ from the parent p of u
     - If p underflows, handle it in the same way
  2. Otherwise, perform a share
     - Distribute the elements in u and u' equally between them
     - Modify $r(u')$ in p
  - If the root ends up having only one child, remove the root

+ B-Tree Deletion

- Deletion of 77 makes u_7 underflow

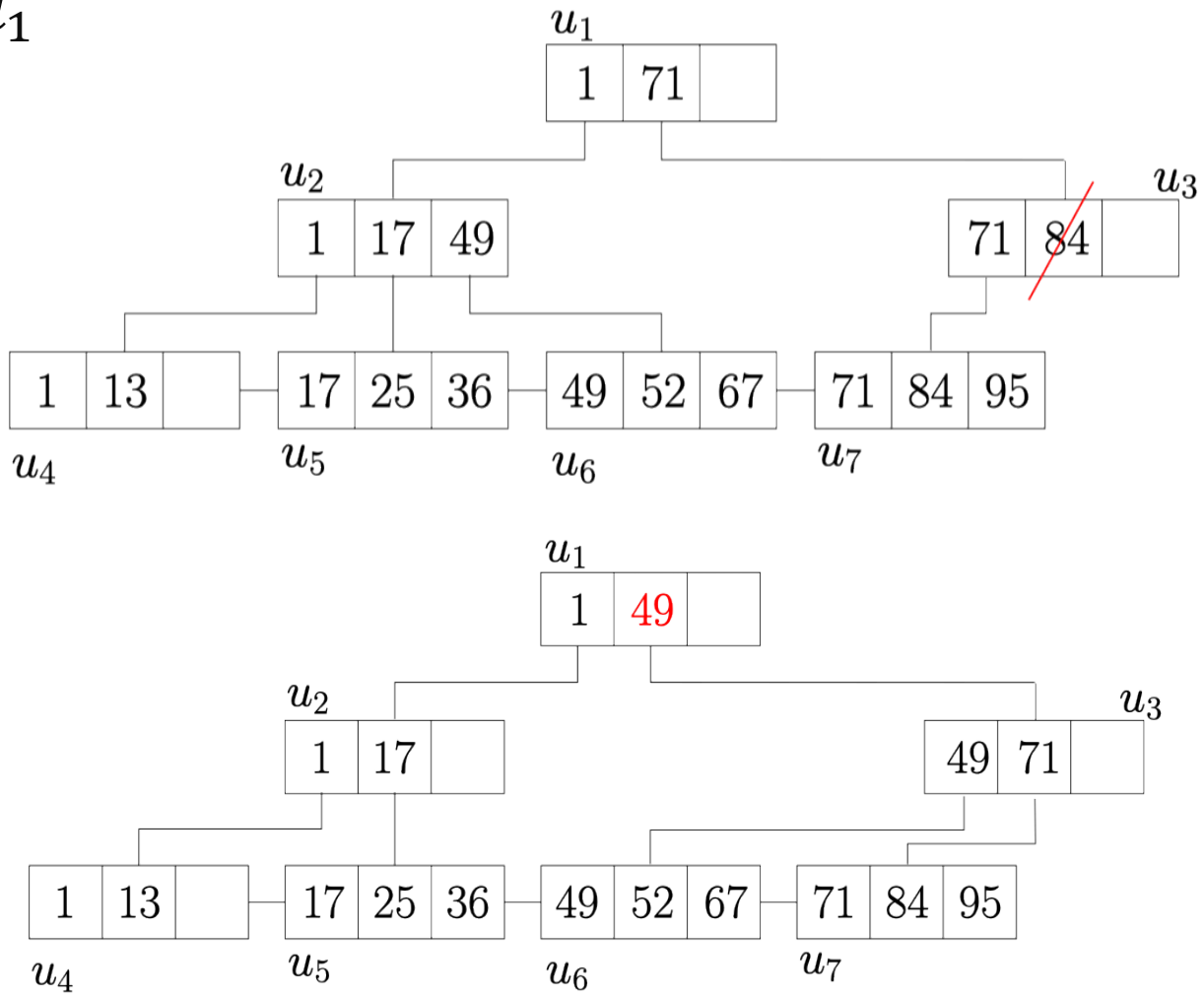


- Merge u_7 and u_7', and remove $r(u_7') = 84$ from p_3, causing underflow of p_3

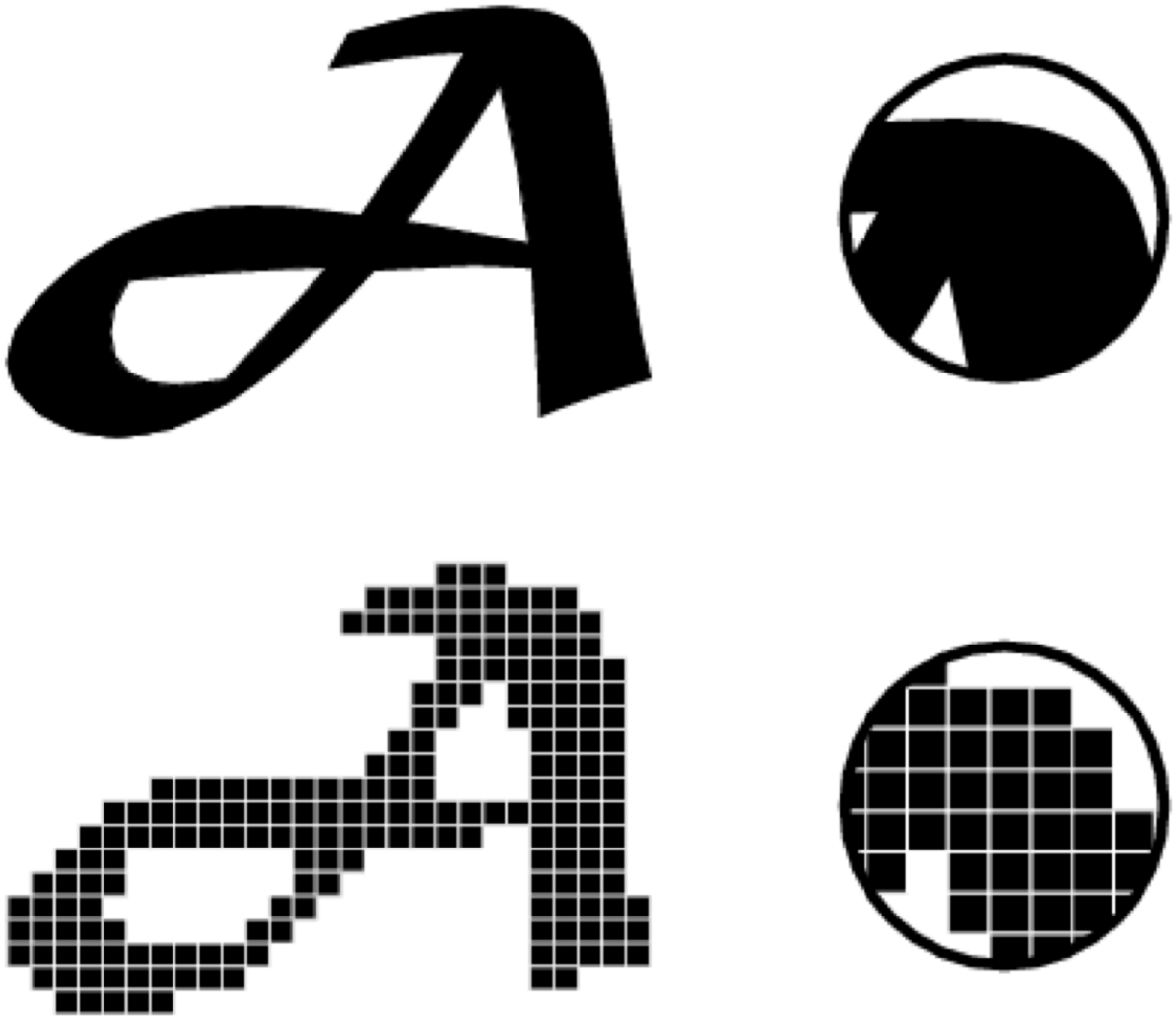


+ B-Tree Deletion

- Perform a share between u_2 and u_3. Update $r(u_3) = 49$ in u_1



+ Bitmap



+ Bitmap

SID	Name	Gender
6007	Peter	M
6100	Ann	F
6107	Bob	M
6207	Jane	F
6240	Suzy	F
6350	Ben	M
6420	Peter	M
6500	Jenn	F

M	F
1	0
0	1
1	0
0	1
0	1
1	0
1	0
0	1

+ Bitmap

SID	Name	Gender	State	M	F	VIC	QLD	NSW
6007	Peter	M	VIC	1	0	1	0	0
6100	Ann	F	QLD	0	1	0	1	0
6107	Bob	M	NSW	1	0	0	0	1
6207	Jane	F	QLD	0	1	0	1	0
6240	Suzy	F	NSW	0	1	0	0	1
6350	Ben	M	NSW	1	0	0	0	1
6420	Peter	M	QLD	1	0	0	1	0
6500	Jenn	F	VIC	0	1	1	0	0

+ Bitmap

SID	Name	Gender	State
6007	Peter	M	VIC
6100	Ann	F	QLD
6107	Bob	M	NSW
6207	Jane	F	QLD
6240	Suzy	F	NSW
6350	Ben	M	NSW
6420	Peter	M	QLD
6500	Jenn	F	VIC

QLD
0
1
0
1
0
0
1
0

- SELECT * FROM Students S WHERE S.state = “QLD”
- Return the Row_ids of: all “1”s in the “QLD” bitmap
  - Rows: 2, 4, 7

+ Bitmap

SID	Name	Gender	State	QLD	F
6007	Peter	M	VIC	0	0
6100	Ann	F	QLD	1	1
6107	Bob	M	NSW	0	0
6207	Jane	F	QLD	1	1
6240	Suzy	F	NSW	0	1
6350	Ben	M	NSW	0	0
6420	Peter	M	QLD	1	0
6500	Jenn	F	VIC	0	1

- SELECT * FROM Students S WHERE S.state = “QLD” and S.Gender = “F”
- Return the Row_ids of: all “1”s in the intersection of “F” and “QLD” bitmaps
  - Rows: 2, 4

+ Bitmap Size

- Size of each bitmap (in bits) is equal to the number of rows in the relation
- Number of bitmaps for a field is equal to the number of distinct values of that field
- Total space needed to index one field (in bits)
  - = number of distinct values × number of rows
- Whereas, file size (in bits)
  - = record size in bits × number of rows
- In general, bitmap indexes are space-efficient

+ Bitmap Limitations

- Not good for data that is modified regularly
  - Updates will require modifying all the associated bitmap indexes
- Used in warehouse data sets which are large and are not updated frequently
  - Mainly for Online Analytical Processing (OLAP)
- How to use it to index the spatial data?

Data Access Methods

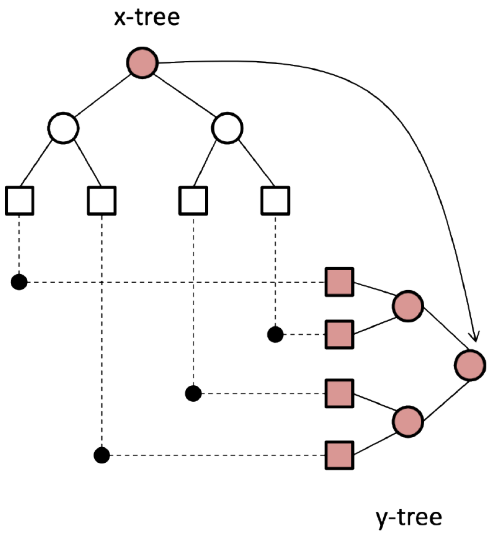
- One Dimensional
  - Hashing and B-Trees
- Line Data
  - Interval Tree, Segment Tree
- Point Data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: Point and Region Quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon Data
  - Transformation: End point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R^+-tree

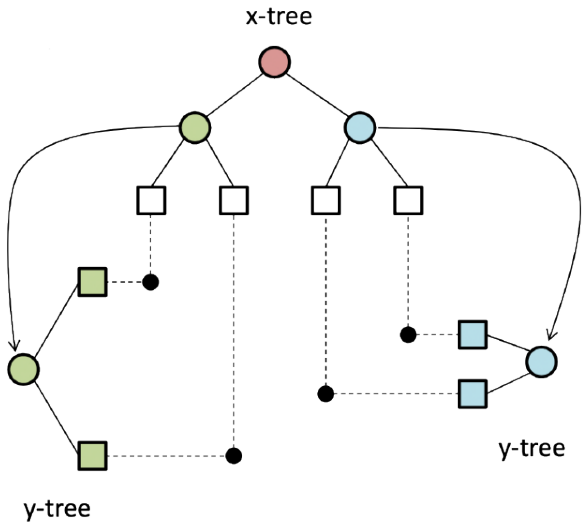
+ Line Data

- Arbitrary direction and shape
  - Use MBR and treat like polygons
  - Treated as trajectory data
- Straight line segments with perpendicular directions
  - Align parallel the axis
  - Interval Tree
- Straight line segments with different directions
  - Use Point Index techniques to index the two end points separately
  - Segment Tree

+ 2D Range Tree

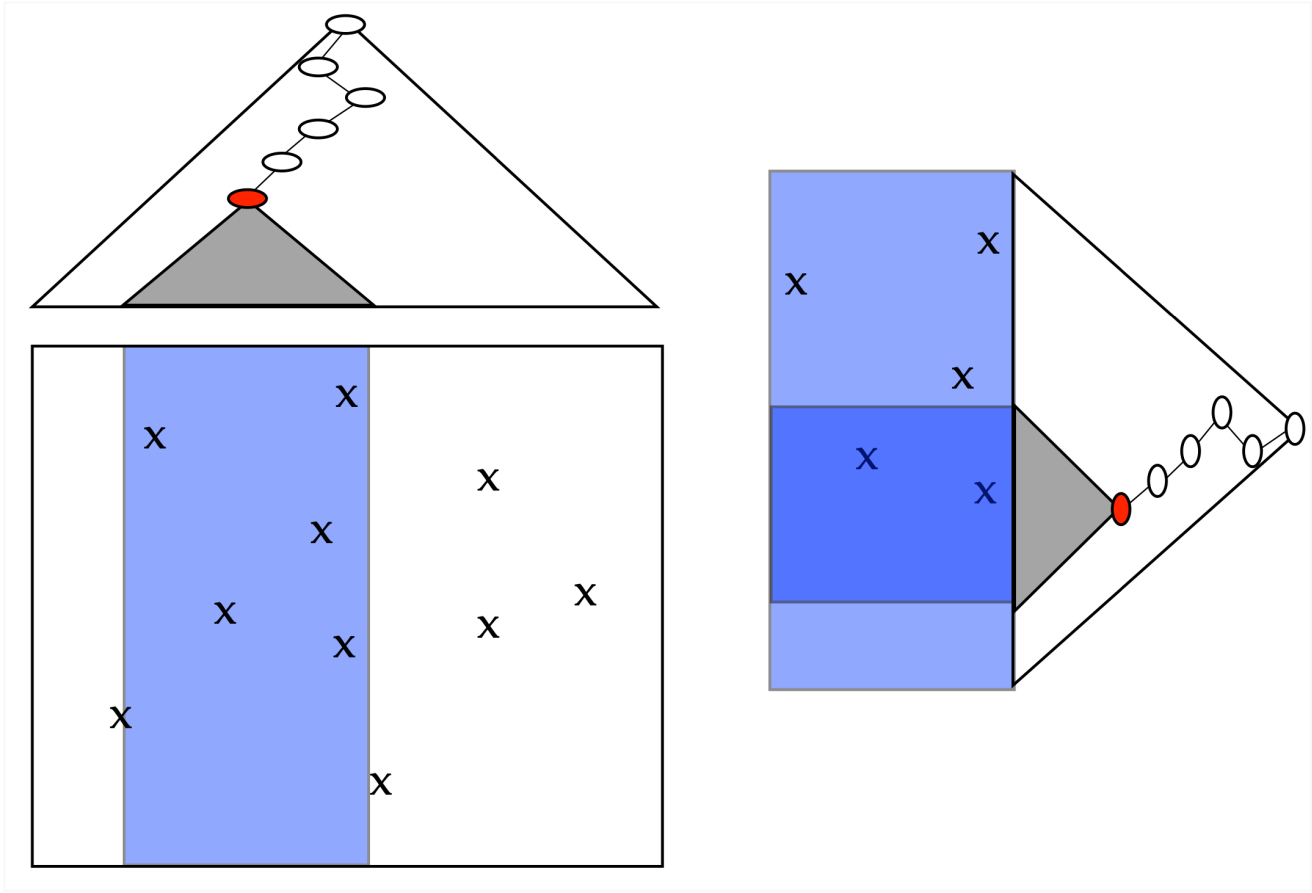
- 1D Range Tree
  - Leaf nodes are linked in sorted order using a doubly linked list
- Binary Tree of Binary Tree
  - One Binary Tree in X (x-Tree)
  - One Binary Tree in Y for each node in the X-Tree
- Complexity
  - Space: $O(n \log n)$
  - Time: $O(n \log n)$





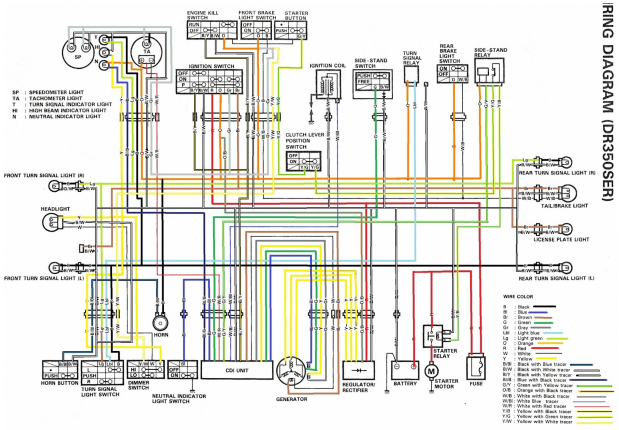
+ 2D Range Tree Search

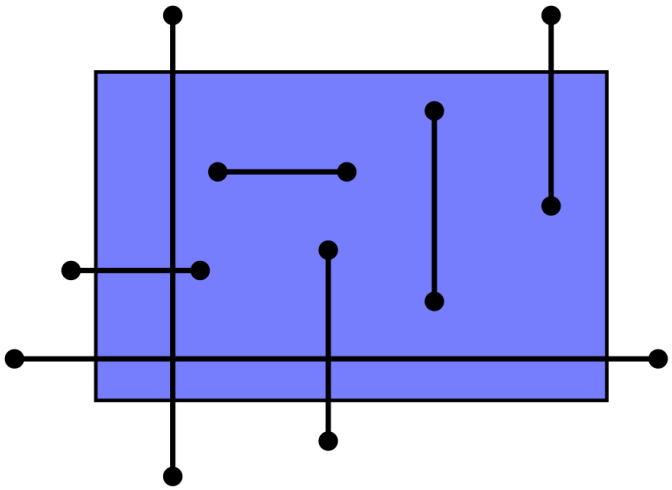
- Range Query $[x_1, x_2] \times [y_1, y_2]$
  - $O(\log^2 n + k)$ complexity



+ Interval Tree

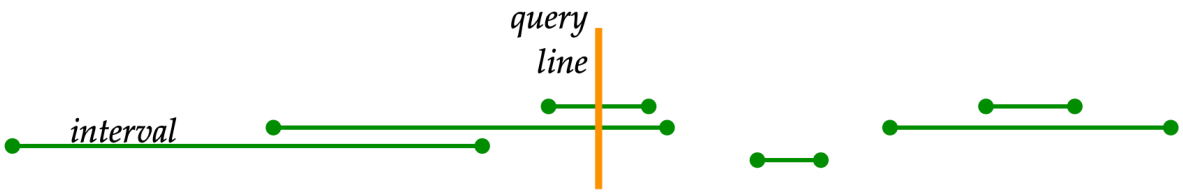
- Axis-Aligned segments
- Range Query
  - Return all segments that have any part of them inside the rectangle
  - Wiring Diagrams, Genes on genomes

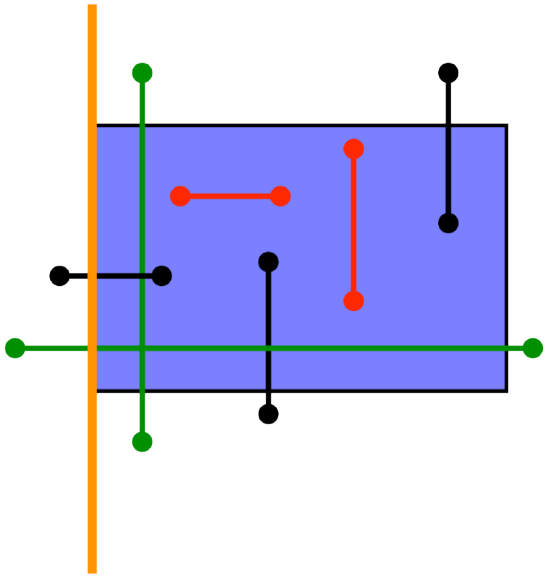




+ Interval Tree

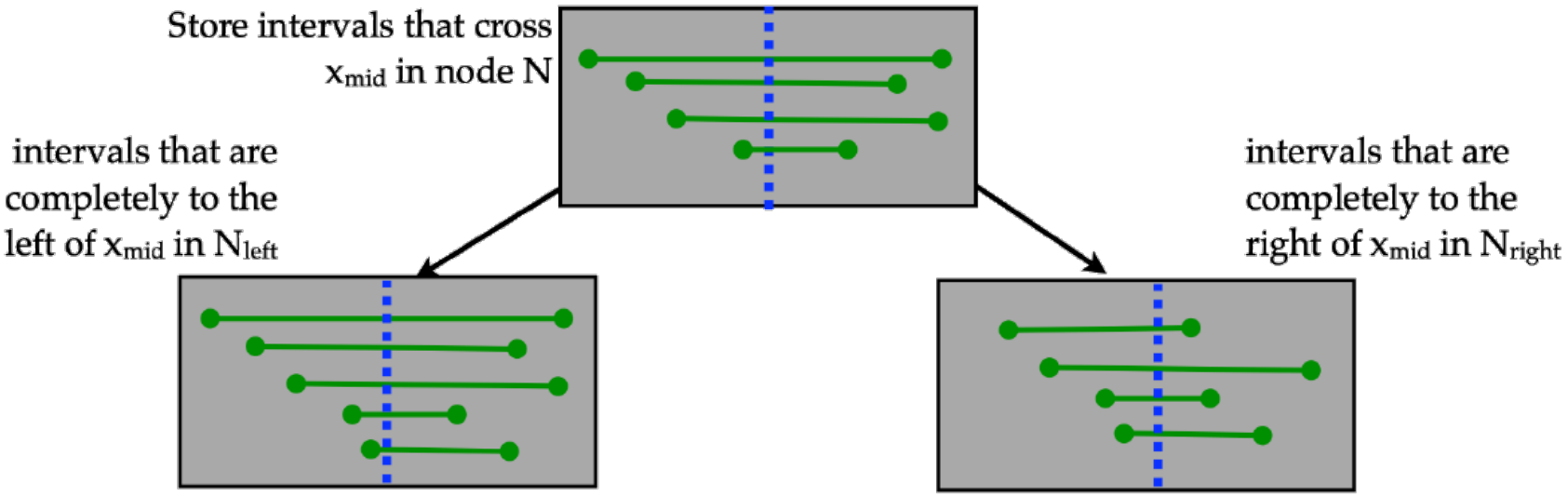
1. Segments with at least one endpoint in the rectangle
   - Black lines and Red lines
   - 2D Range Tree
   - Any point index structures
2. Segments with no endpoints in the range
   - Green lines
   - Consider just horizontal segments
   - They must cross a vertical side of the region
   - Subproblem
     - Given a vertical line, find segments that it crosses
     - Y is irrelevant for this subproblem





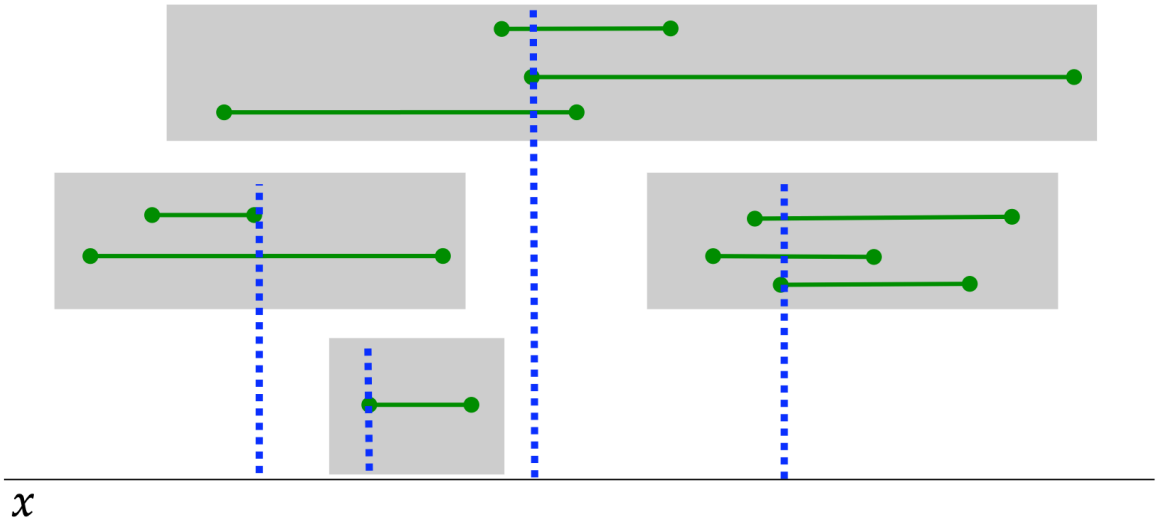
+ Interval Tree Construction

- Recursively build tree on intervals as follows:
  - Sort $2n$ endpoints
  - x_{mid} is the median point
    - Store the intervals crosses x_{mid}
    - Left child: intervals completely smaller than x_{mid}
      - Right end point smaller than x_{mid}
    - Right child: intervals completely larger than x_{mid}



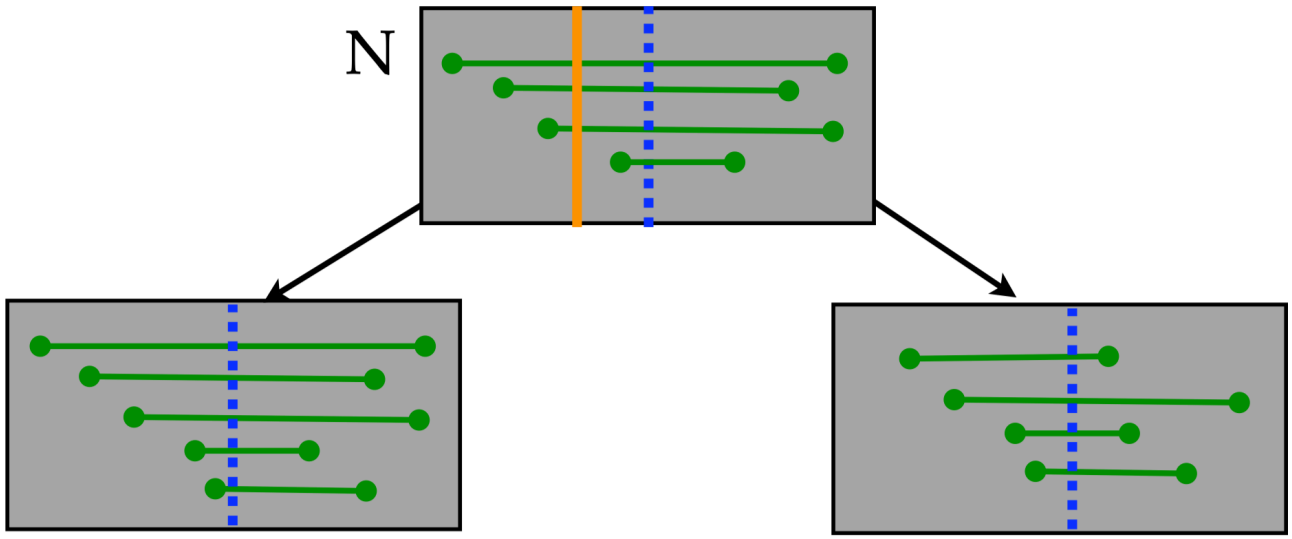
+ Interval Tree Construction

- Store x_{mid} with each node
- Approximately balanced because by choosing the median, we split the set of end points up in half each time
  - Height is $O(\log n)$
- $O(n)$ Storage
  - Each interval stored once
  - Fewer than n nodes (each node contains at least one interval)
- $O(n \log n)$ Construction time



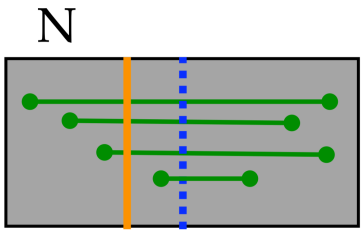
+ Interval Tree Search

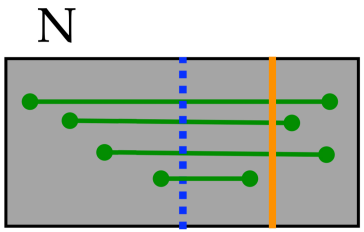
- Query x_q: a vertical line
  - Suppose we are at node N
    - If $x_q < x_{med}$, skip the right subtree
    - If $x_q \geq x_{med}$, skip the left subtree
    - Always have to search the intervals stored at current node
    - $logN$ nodes to check



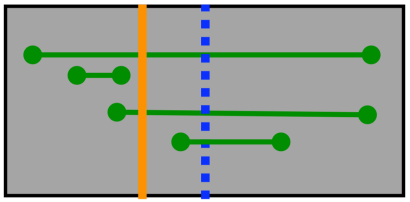
+ Interval Tree Search

- Current Node Search
  - Store each interval in two sorted list stored at node
    - List L sorted by increasing left endpoint
    - List R sorted by decreasing right endpoint
  - Search List depending on which side of x_{med} the query is on
    - If $x_q < x_{med}$, search L, output all until a left endpoint $> x_q$
    - If $x_q \geq x_{med}$, search R, output all until a right endpoint $< x_q$



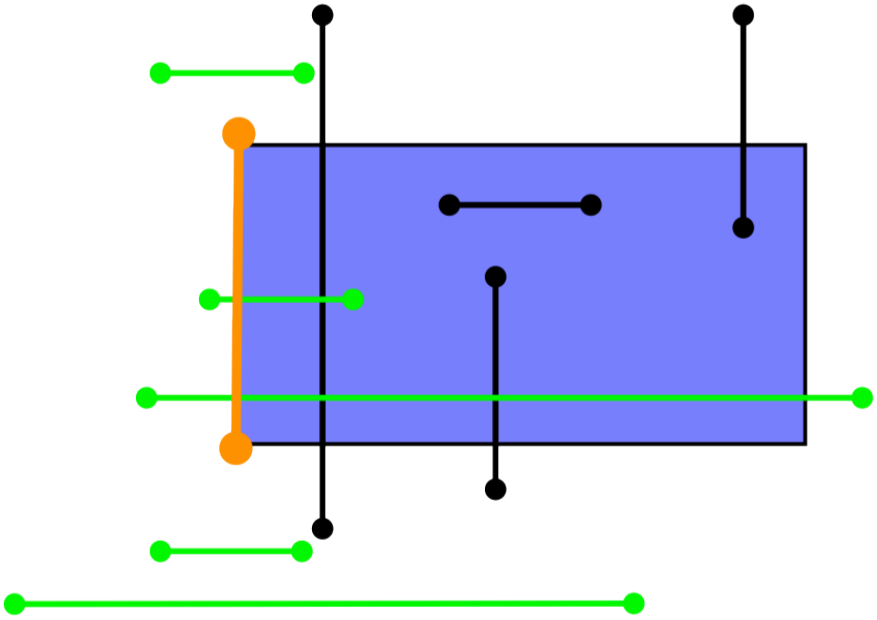


  - It works because we know each segment intersects x_{med}



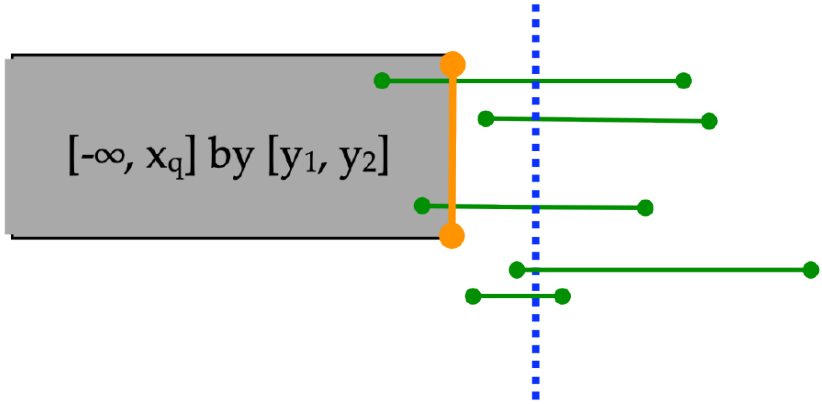
+ Interval Tree

- Vertical Segment Search
  - Instead of infinite vertical lines, we have finite segments as query
    - Somehow have to remove the ones that don't satisfy the y-constraints
    - Interval trees → candidates
      - How to remove the ones that don't satisfy the y-constraints?



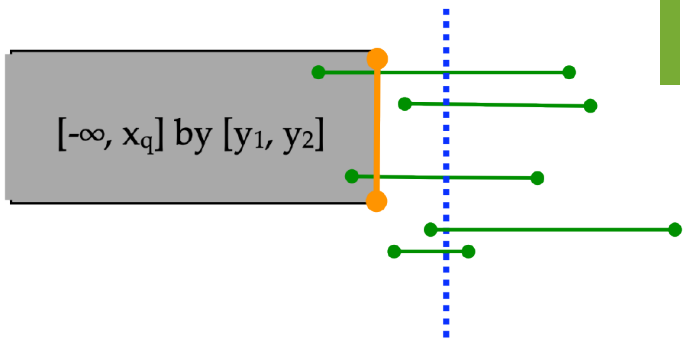
+ Interval Tree

- Vertical Segment Search
  - Use 2D Range Tree instead of sorted lists to hold segments at each node
    - Execute a range query on a semi-infinite range on the 2d-range tree on the end points stored at each node of the interval tree.
    - Two range trees R_{left} and R_{right} that store points to the left and to the right of x_{mid}
  - Time complexity
    - $O(\log^3 n + k)$ with two trees
    - $O(\log^2 n + k)$ with *fractional cascading*
  - Space complexity
    - Each interval stored at one node
    - Total space for the range trees is $O(nlog\, n)$



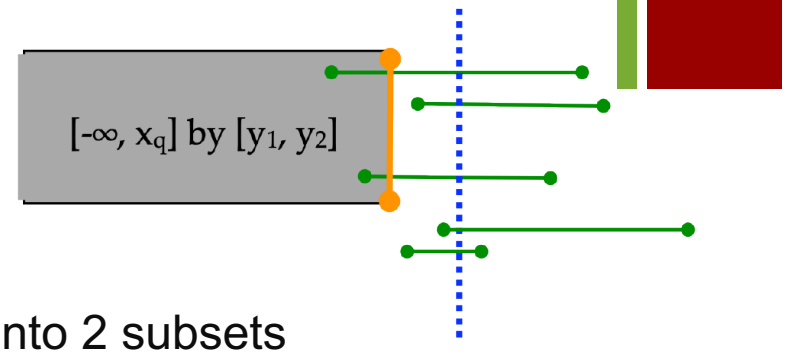
Priority Search Tree

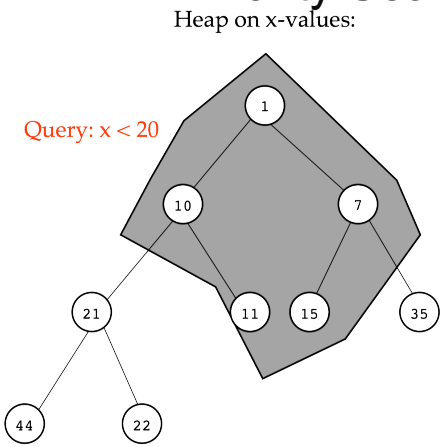
- Space Complexity $O(n)$
- 1-Side Range Query
  - $[-\infty, x], [y_1, y_2]$
  - Easy in 1D
    - Just walk the sorted list from left to right or right to left
  - 2D: $x < 20$ and $25 < y < 70$
  - Somehow integrate the the information about y-coordinate in the structure

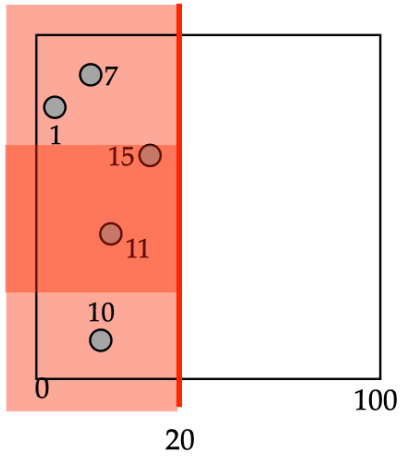


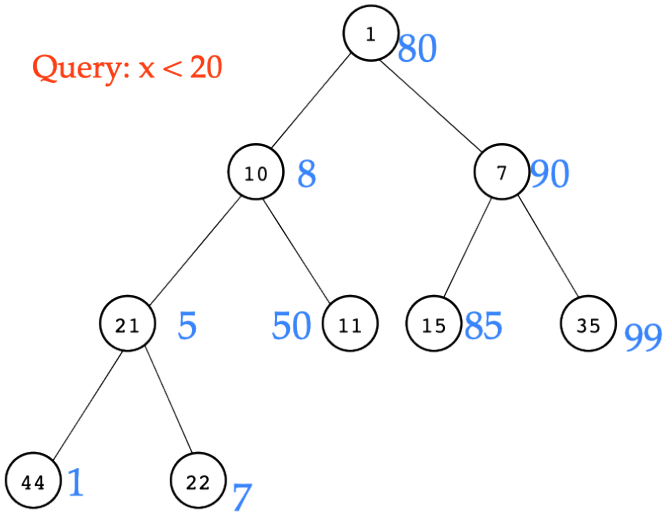
+ Priority Search Tree

- 1-Side Range Query
  - $[-\infty, x], [y_1, y_2]$
  - Find the highest/lowest value of x
    - Heap
      - Freedom in how to partition the set into 2 subsets
  - Find the range of y
    - Binary Search Tree
  - Combine them!
    - Priority Search Tree







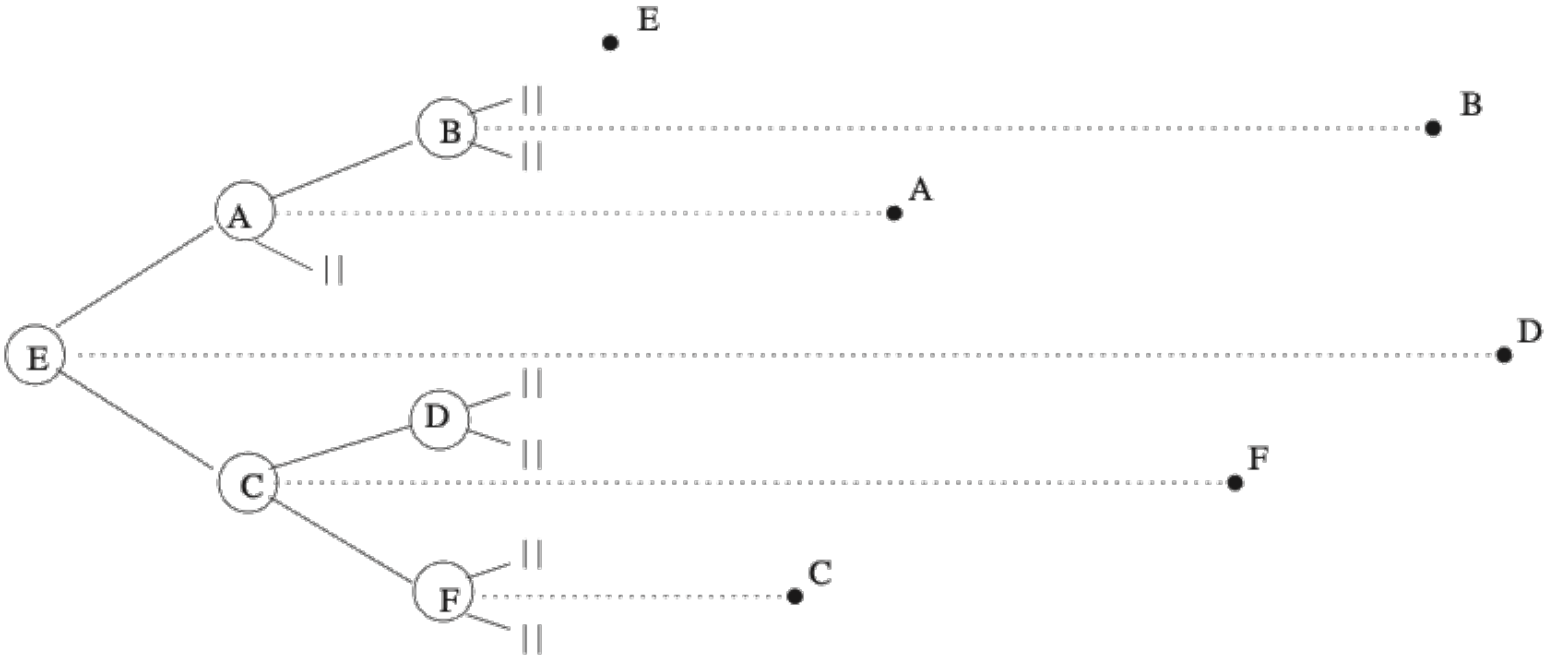


Priority Search Tree

- Construction
  - Given a set of points P
    - p_{minx} is the one with the smallest x
    - y_{mid}: median of the y-coordinates of $P/\{p_{minx}\}$
  - Store point p_{minx} and y_{mid} in a node N
    - y_{mid} does not need to correspond to point p_{minx}
- Split the points by y-coordinate
  - P_{left}=$\{p \in P\backslash\{p_{minx}\} \ \ and\ p.y \leq y_{mid} \ \}$
  - P_{right}=$\{p \in P\backslash\{p_{minx}\} \ and\ p.y > y_{mid} \}$
- Recursively build left/right subtrees
  - $O(n\ logn)$

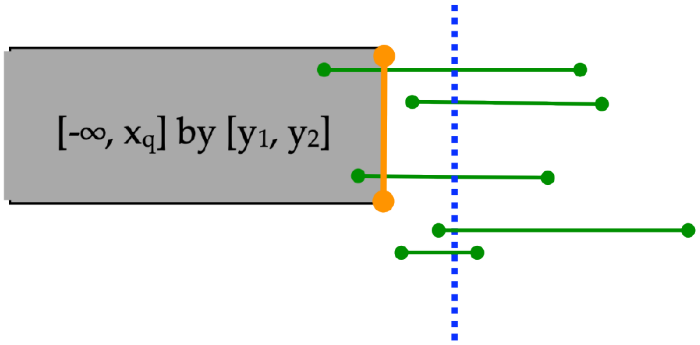
Priority Search Tree

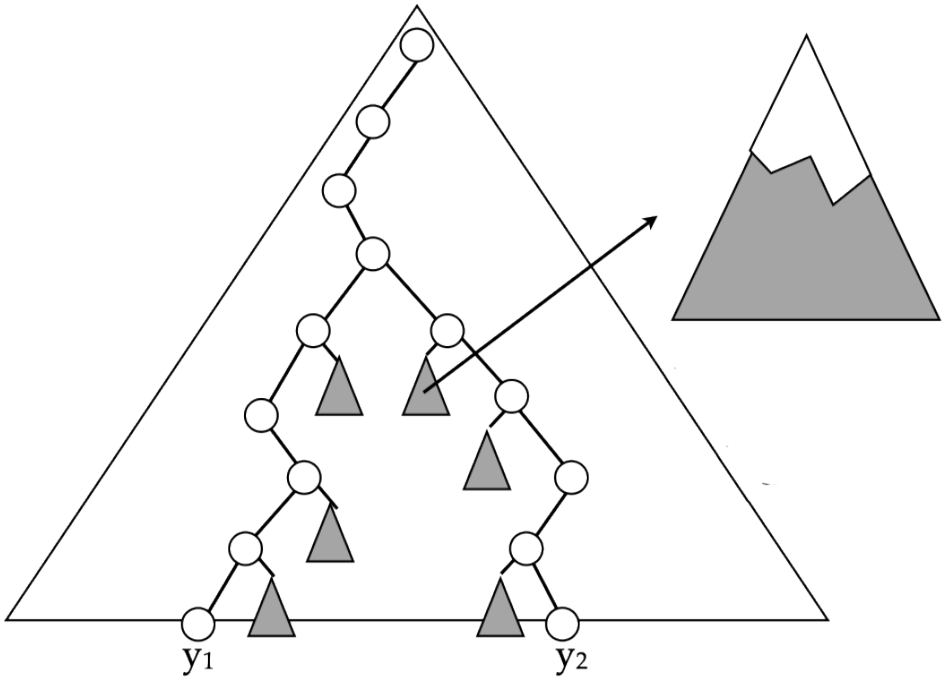
- Construction
  - Binary tree on y
  - Each sub-tree is a heap on x



+ Priority Search Tree

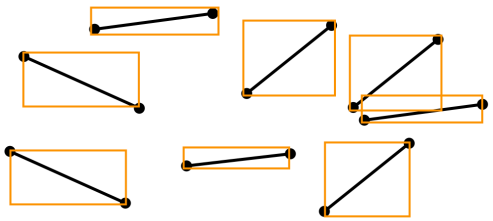
- Search
  - Range search for y can be done in a 1-d range tree
    - Points in the grey sub-trees all satisfy the y-constraints
    - Points along the search paths may or may not
      - Must check them
  - Each result sub-tree is a heap
    - Output the tops of the heap
  - $O(logn + k)$

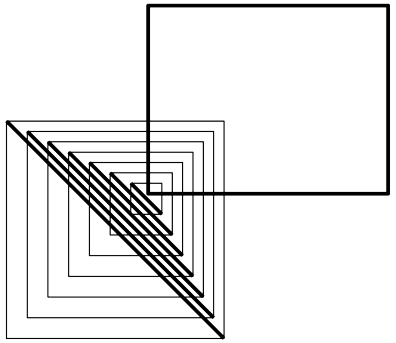




+ Segment Tree

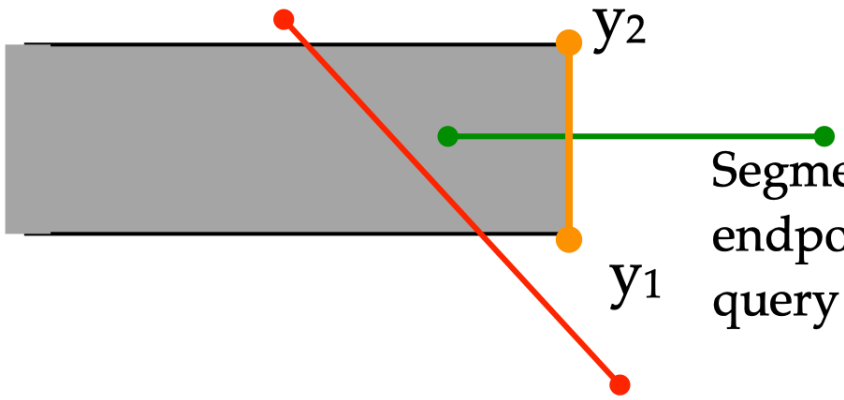
- Arbitrarily Oriented Segments
  - No longer assume they are parallel to the axes
  - Store the bounding boxes of each segment as a collection of 4 axis-parallel segments?
    - How to handle range queries on these kinds of segments
    - If a vertical line crosses a segment, it crosses its MBR
    - It may be that a vertical segment crosses a bounding box but doesn't cross the segment
- Vertical Segment Stabbing Query
  - Return all segments that intersect a vertical query segment
  - Assume segments don't cross





+ Segment Tree

- Interval Tree?
  - No longer true that a query like $[-\infty, x]$ by $[y_1, y_2]$ will find the endpoints of satisfying segments

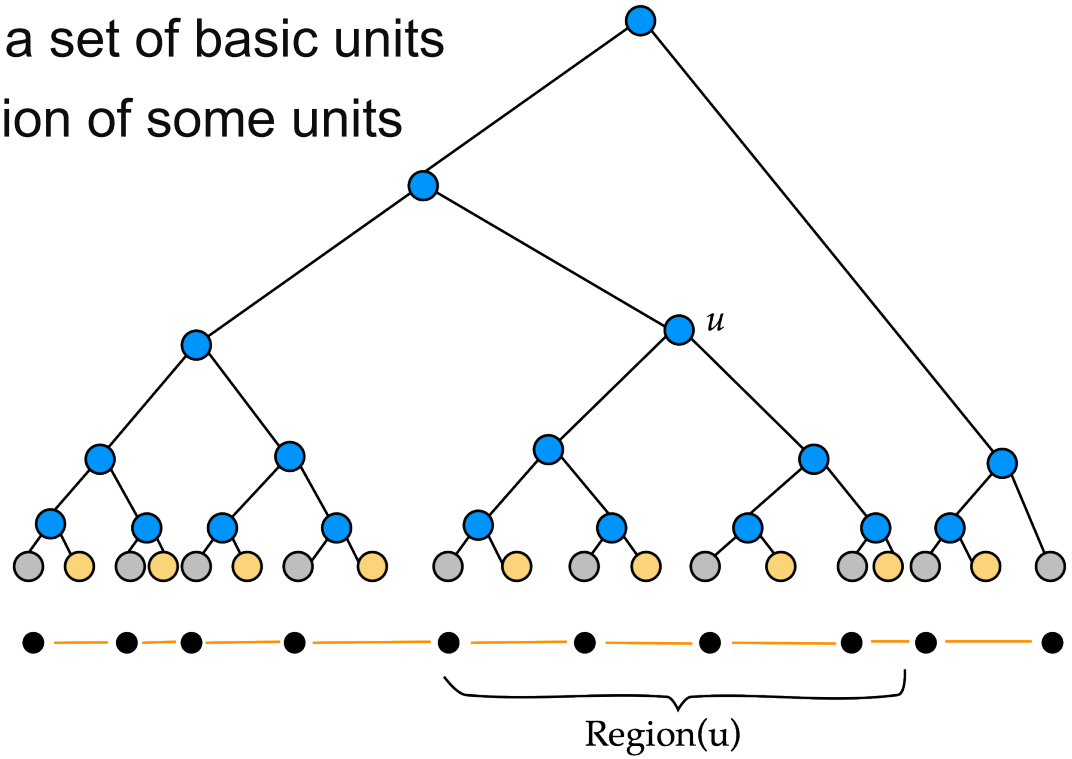


Segment intersects range and endpoint falls in half-infinite query (as in interval trees)

Segment intersects query, but there are no endpoints in the range

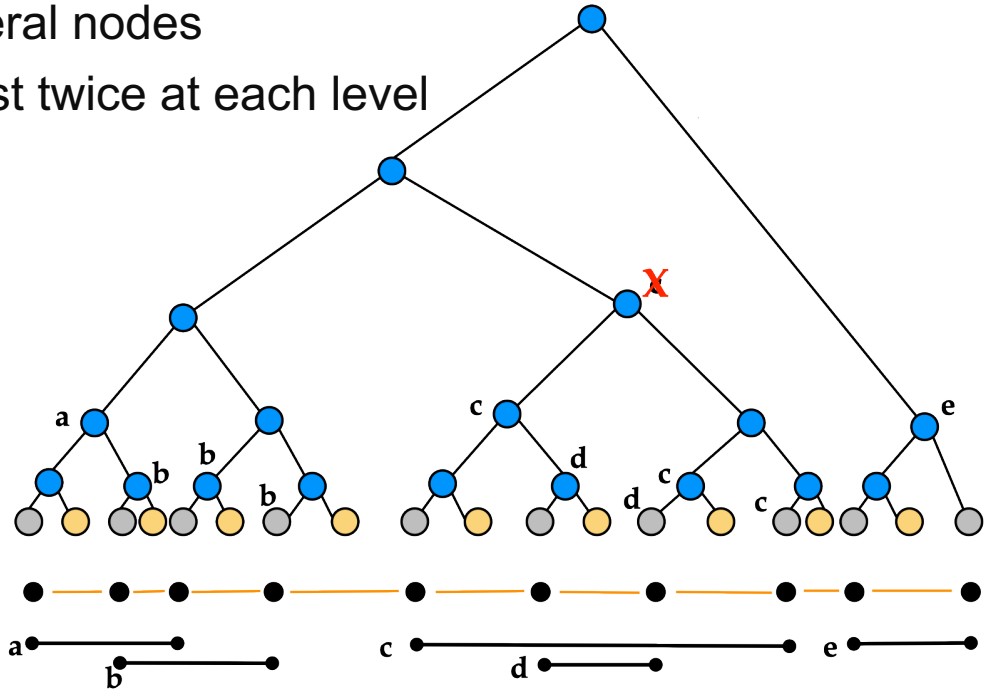
+ Segment Tree

- Binary Search Tree
  - Recursively partitions 1D space
  - Each leaf stores an elementary region
  - For internal node u, $Region(u)$ is the union of elementary regions in the subtree rooted at u
  - The space is divided into a set of basic units
  - Every segments is the union of some units



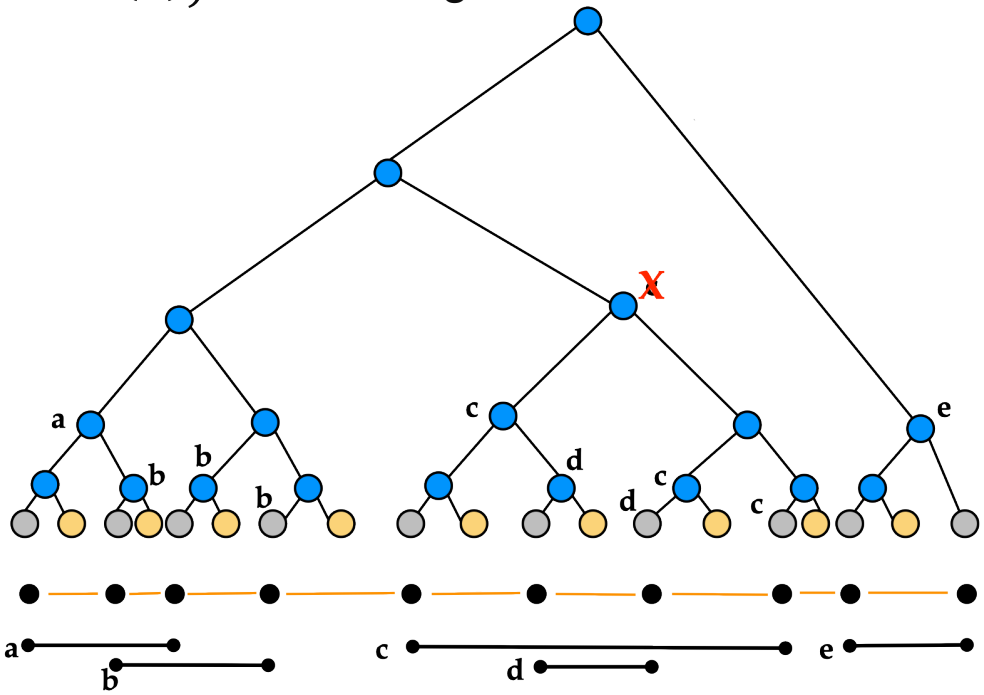
+ Segment Tree

- Store the segments in the Binary Search Tree
  - Store the segment s at any node u that
    - Segment covers the entire $Region(u)$
    - It doesn't cover the entire $Region(Parent(u))$
      - Propagate segments up until we reach a node whose Region is not a subset of the segment
    - Segments may be stored at several nodes
      - Each segment is stored at most twice at each level



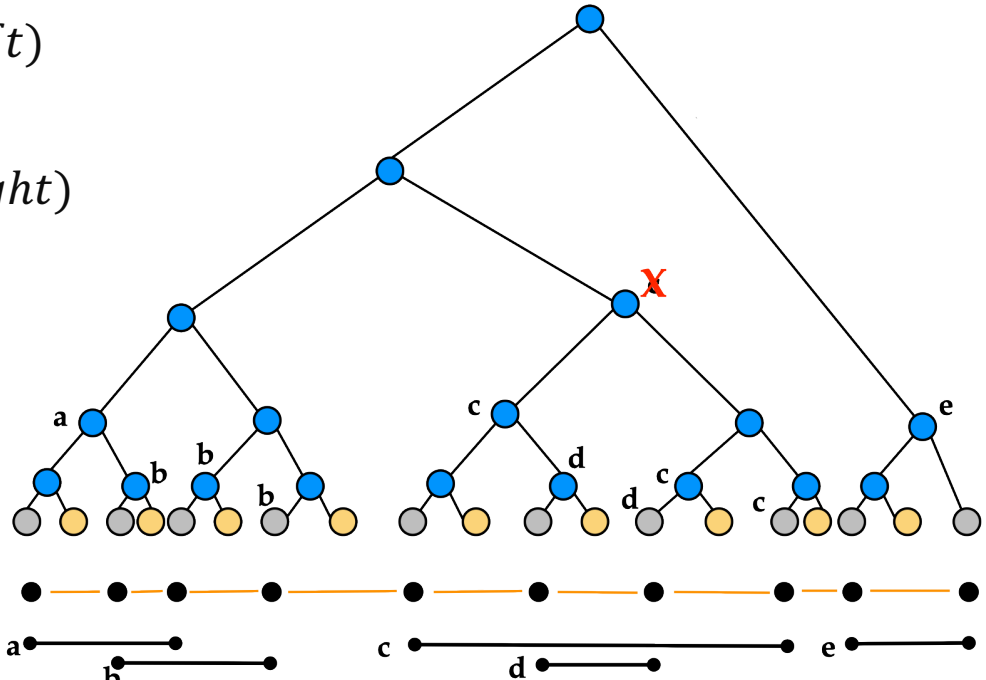
+ Segment Tree

- Vertical Line Query
  - Find the segment that intersect a given x
    - Binary search
    - Output every segment stored at the current node
      - If x falls into $Region\big(leftchild(u)\big)$, take the left branch
      - Else if x falls into $Region\big(rightchild(u)\big)$, take the right branch



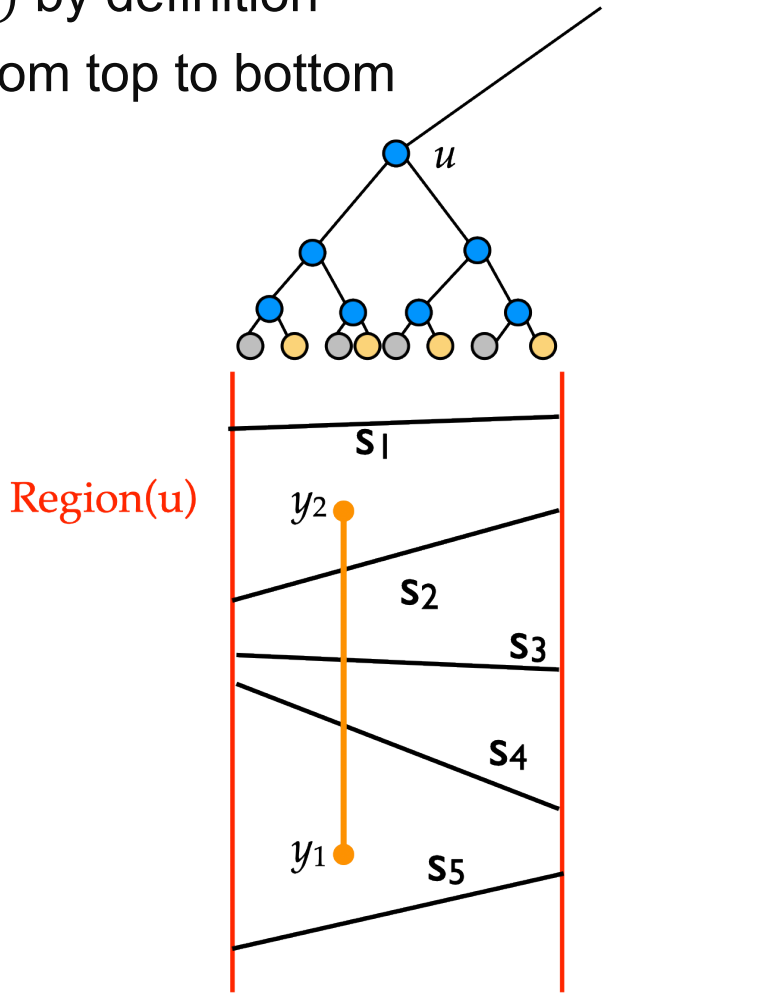
+ Segment Tree

- Construction
  - Sort the segments
  - Break into units
  - Build balanced BST on them
  - Insert the segment (x_1, x_2, u)
    - If $Region(u) \subset [x_1, x_2]$, $u.insert(x_1, x_2)$
    - If $[x_1, x_2]$ intersects $Region(u.left)$
      - Insert$(x_1, x_2, u.left)$
    - If $[x_1, x_2]$ intersects $Region(u.right)$
      - Insert$(x_1, x_2, u.right)$



+ Segment Tree

- Vertical Segment Stabbing Query
  - Segments stored at u all span $Region(u)$ by definition
  - The segments can be linearly ordered from top to bottom
  - Index them in binary search tree
  - Range query between y_1 and y_2
  - $O(\log^2 n + k)$

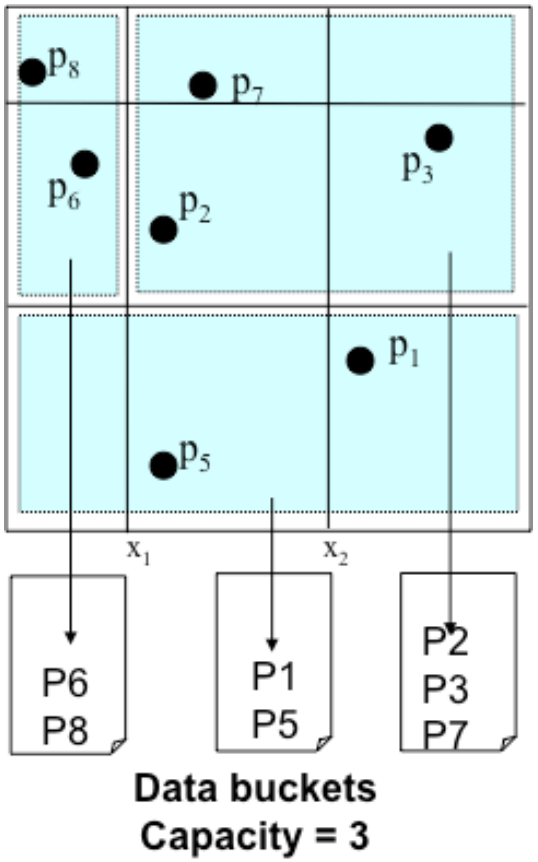


Data Access Methods

- One Dimensional
  - Hashing and B-Trees
- Line Data
  - Interval Tree, Segment Tree
- Point Data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: Point and Region Quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon Data
  - Transformation: End point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R^+-tree

Grid File

- Basic idea
  - Superimpose a k-dimensional grid on the space
    - Only scan the data in a grid
    - Essentially, a 2D hash function
  - Cells can be of varying sizes
  - Cell-to-bucket mapping: many-to-1
    - *What about 1-to-many?*
  - The grid definition (scales) is kept in memory
  - The grid directory is kept on disk
- Motivation
  - Fixed-grid not suitable for non-uniformly distributed data



Grid File

- To answer a point query:
  - Use the scales to locate the cell
  - Read the cell from disk
  - The loaded cell contains a reference (pointer) to data bucket
  - Read data bucket
  - On average two disk accesses

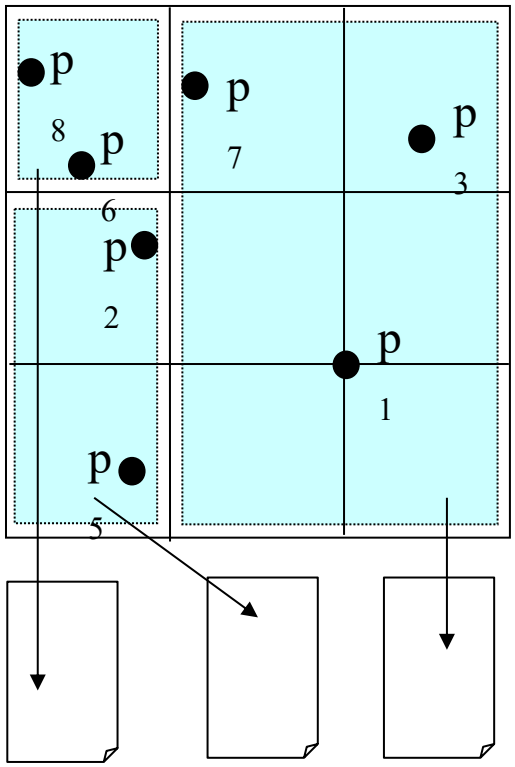
- To answer a range query:
  - Examine all cells that overlap the search region
  - Read the corresponding data buckets(s)

+ Grid File

- To insert a point:
  - Search (point query) the matching cell and data bucket
  - If there is sufficient space, insert into data bucket
  - Else: add a vertical or horizontal line to split, if necessary, and move data accordingly
- To delete a point:
  - Search …
  - Merge if necessary
- Problems:
  - Have to remember all the definitions of the grids
  - Why not uniformally?

+ EXCELL

- EXtendible CELL
- Motivation
  - Fixed grid is easier to manage and more efficient to use
- Basic Ideas
  - All cells are of the same size
    - No need to keep grid definition (scales) in memory
    - But it's still necessary to remember how to map grid cells to buckets



Data buckets
Capacity = 3

+ EXCELL

- Search:
  - Efficient for point and range queries
- To insert a point:
  - Search (point query) the matching cell and data bucket
  - If there is sufficient space, insert into data bucket
  - Else: add a vertical or horizontal line to split, if necessary, and move data accordingly
  - When split, the number of cells is doubled

Data Access Methods

- One Dimensional
  - Hashing and B-Trees
- Line Data
  - Interval Tree, Segment Tree
- Point Data
  - Hashing: GRID and EXCELL
  - Hierarchical
    - Quadtree: Point and Region Quadtrees
    - kd-Tree
    - Z-values and B-tree
- Polygon Data
  - Transformation: End point mapping and z-values
  - Overlapping: R-tree and R*-tree
  - Clipping: R^+-tree