CS 240 – Data Structures and Data Management

Module 8: Range-Searching in Dictionaries for Points

Simon Abélard   Ahmad Biniaz   Éric Schost
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2019

References: Goodrich & Tamassia 21.1, 21.3

version *2019-06-12 12:29*

Outline

1. Range-Searching in Dictionaries for Points
   - Range Queries
   - Multi-Dimensional Data
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion

Outline

Range queries

- So far: *search*(k) looks for *one* specific item.
- New operation **RangeQuery**: look for *all* items that fall within a given range.
  - Input: A **range**, i.e., an interval $I = (x, x')$
    It may be open or closed at the ends.
  - Want: Report all KVPs in the dictionary whose key k satisfies $k \in I$

Example:

5	10	11	17	19	33	45	51	55	59

RangeQuery((18,45]) should return $\{19, 33, 45\}$

- Let s be the **output-size**, i.e., the number of items in the range.
- We need $\Omega(s)$ time simply to report the items.
- Note that sometimes $s = 0$ and sometimes $s = n$; we therefore keep it as a separate parameter when analyzing the run-time.

Range queries in existing dictionary realizations

Unsorted list/array/hash table: Range query requires $\Omega(n)$ time:
We have to check for each item explicitly whether it is in the range.

Sorted array: Range query in A can be done in $O(\log n + s)$ time:

RangeQuery((18,45])

5	10	11	17	19	33	45	51	55	59
			$\uparrow i$			$\uparrow i'$			

- Using binary search, find i such that x is at (or would be at) $A[i]$.
- Using binary search, find i' such that x' is at (or would be at) $A[i']$
- Report all items $A[i{+}1 \ldots i'{-}1]$
- Report $A[i]$ and $A[i']$ if they are in range

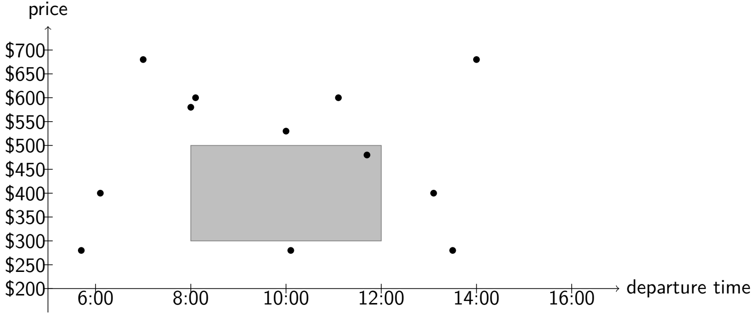
BST: Range query can similarly be done in time $O(\text{height}{+}s)$ time.
We will see this in detail later.

Outline

Multi-Dimensional Data

Range queries are of special interest for **multi-dimensional data**.

Example: flights that leave between 9am and noon, and cost \$300-\$500



- Each item has d **aspects** (coordinates): $(x_0, x_1, \cdots, x_{d-1})$
- Aspect values (x_i) are numbers
- Each item corresponds to a point in d-dimensional space
- We concentrate on $d = 2$, i.e., points in Euclidean plane

Multi-dimensional Range Search

(Orthogonal) d-**dimensional range query**: Given a **query rectangle** A, find all points that lie within A.

The time for range queries depends on how the points are stored.

- Could store a 1-dimensional dictionary (where the key is some combination of the aspects.)
  Problem: Range search on one aspect is not straightforward
- Could use one dictionary for each aspect
  Problem: inefficient, wastes space
- **Better idea**: Design new data structures specifically for points.
  - Quadtrees
  - kd-trees
  - range trees

Outline

Quadtrees

We have n points $S = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$ in the plane.

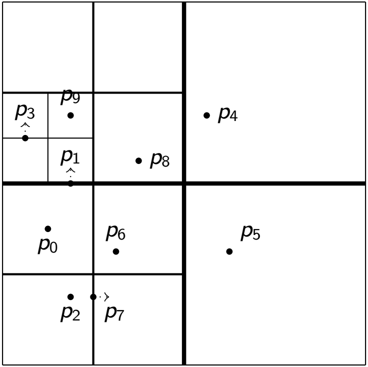
We need a **bounding box** R: a square containing all points.

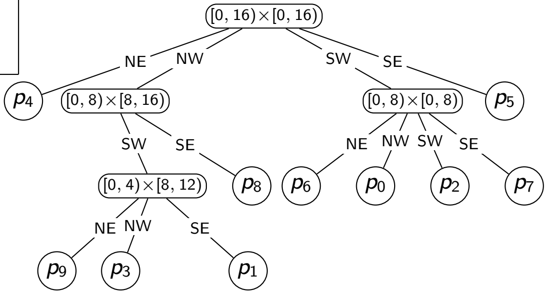
- Can find R by computing minimum and maximum x and y values in S
- The width/height of R should be a power of 2

Structure (and also how to *build* the quadtree that stores S):

- Root r of the quadtree is associated with region R
- If R contains 0 or 1 points, then root r is a leaf that stores point.
- Else *split*: Partition R into four equal subsquares (**quadrants**) $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
- Partition S into sets $S_{NE}, S_{NW}, S_{SW}, S_{SE}$ of points in these regions.
  - **Convention:** Points on split lines belong to right/top side
- Recursively build tree T_i for points S_i in region R_i and make them children of the root.

Quadtrees example

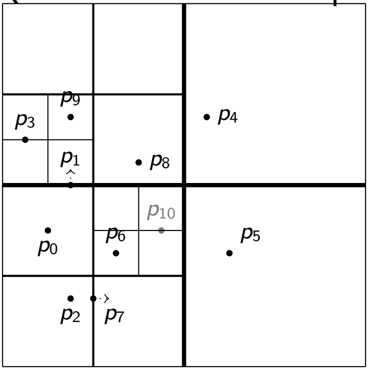




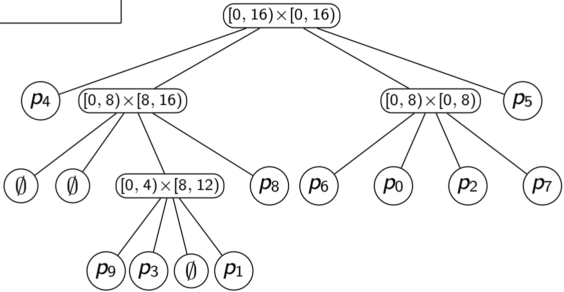
Quadtree Dictionary Operations

- *search*: Analogous to binary search trees and tries
- *insert*:
  - Search for the point
  - Split the leaf while there are two points in one region
- *delete*:
  - Search for the point
  - Remove the point
  - If its parent has only one point left: delete parent (and recursively all ancestors that have only one point left)

Quadtree Insert example



insert(p_{10})

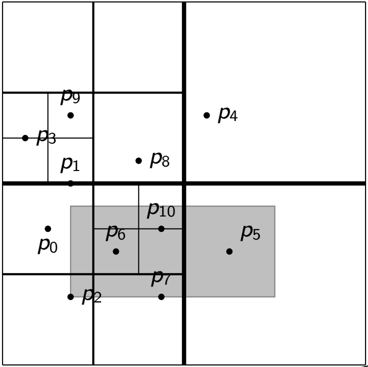


Quadtree Range Search

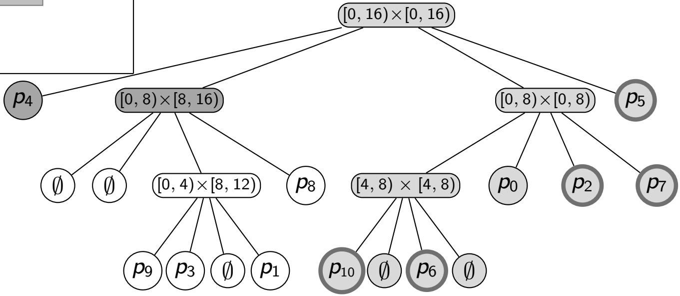
```
QTree::RangeSearch(r ← root, A)
r: The root of a quadtree, A: Query rectangle
    R ← region associated with node r
    if (R ⊆ A) then                  // inside node
            report all points below r; return
    if (R ∩ A is empty) then         // outside node
            return

            // The node is a boundary node, recurse
    if (r is a leaf) then
        p ← point stored at r
        if p is in A return p
        else return
    for each child v of r do
        QTree::RangeSearch(v, A)
```

Note: We assume here that each node of the quadtree stores the associated square. Alternatively, these could be re-computed during the search (space-time tradeoff).

Quadtree range search example

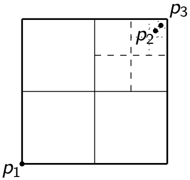


- Red: Search stopped due to $R \cap A = \emptyset$.
- Green: Search stopped due to $R \subseteq A$.
- Blue: Must continue search in children / evaluate.



Quadtree Analysis

- Crucial for analysis: what is the height of a quadtree?
  - Can have very large height for bad distributions of points



  - **spread factor** of points S:

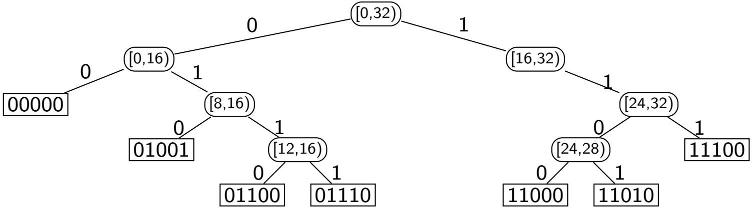
$$\beta(S) = \frac{\text{sidelength of } R}{\text{minimum distance between points in } S}$$

  - Can show: height h of quadtree is in $\Theta(\log \beta(S))$
- Complexity to build initial tree: $\Theta(nh)$ worst-case
- Complexity of range search: $\Theta(nh)$ worst-case even if the answer is $\emptyset$
- But in practice much faster.

Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:

"Points:"	0	9	12	14	24	26	28
(in base-2)	00000	01001	01100	01110	11000	11010	11100



Same as a trie (with splitting stopped once key is unique)

- Quadtrees also easily generalize to higher dimensions (octrees, *etc.*) but are rarely used beyond dimension 3.

Quadtree summary

- Very easy to compute and handle
- No complicated arithmetic, only divisions by 2 (bit-shift!) if the width/height of R is a power of 2
- Space potentially wasteful, but good if points are well-distributed
- Variation: We could stop splitting earlier and allow up to S points in a leaf (for some fixed bound S).
- Variation: Store pixelated images by splitting until each region has the same color.

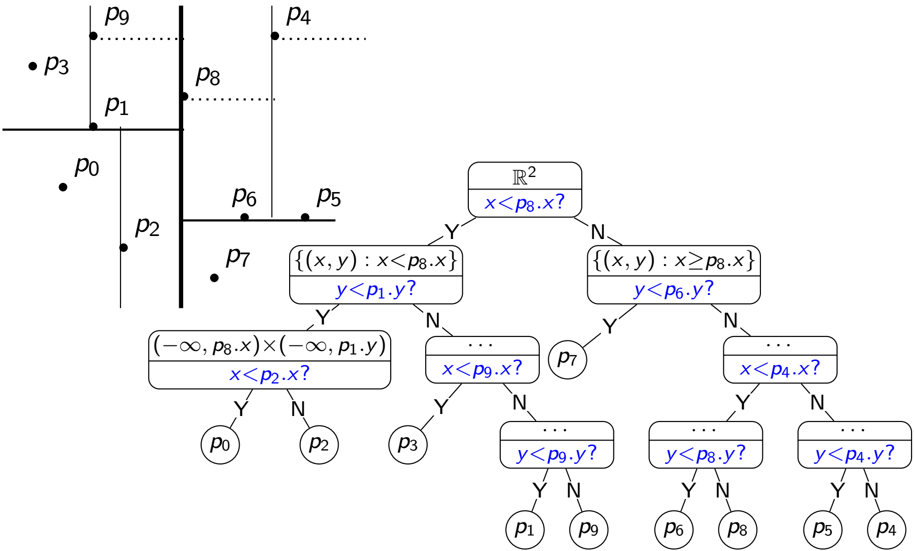
Outline

kd-trees

- We have n points $S = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$
- Quadtrees split square into quadrants regardless of where points are
- (Point-based) kd-tree idea: Split the region such that (roughly) half the point are in each subtree
- Each node of the kd-tree keeps track of a **splitting line** in one dimension (2D: either vertical or horizontal)
- **Convention:** Points on split lines belong to right/top side
- Continue splitting, switching between vertical and horizontal lines, until every point is in a separate region

  (There are alternatives, e.g., split by the dimension that has better aspect ratios for the resulting regions. No details.)

kd-tree example



For ease of drawing, we will usually not show the associated regions.

Constructing kd-trees

Build kd-tree with initial split by x on points S:

- If $|S| \leq 1$ create a leaf and return.
- Else $X :=$ *quick-select*$(S, \lfloor \frac{n}{2} \rfloor)$ (select by x-coordinate)
- Partition S by x-coordinate into $S_{x<X}$ and $S_{x \geq X}$
- Create left subtree recursively (splitting by y) for points $S_{x<X}$.
- Create right subtree recursively (splitting by y) for points $S_{x \geq X}$.

Building with initial y-split symmetric.

Run-time:

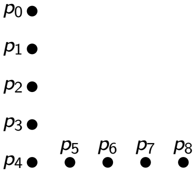
- Find X and partition S in $\Theta(n)$ expected time.
- $\Theta(n)$ expected time on each level in the tree
- Total is $\Theta(\textit{height} \cdot n)$ expected time
- This can be reduced to $\Theta(n \log n + \textit{height} \cdot n)$ *worst-case* time by pre-sorting (no details).

kd-tree height

Assume first that the points are in **general position** (no two points have the same *x*-coordinate or *y*-coordinate).

- Then the split always puts $\lfloor \frac{n}{2} \rfloor$ points on one side and $\lceil \frac{n}{2} \rceil$ points on the other.
- So height $h(n)$ satisfies the sloppy recurrence $h(n) \leq h(\frac{n}{2}) + 1$.
- This resolves to $h(n) \in O(\log n)$
- So can build the *kd*-tree in $\Theta(n \log n)$ time and $O(n)$ space.

If points share coordinates, then height can be infinite!



This could be remedied by modifying the splitting routine. (No details.)

kd-tree Dictionary Operations

- *search* (for single point): as in binary search tree using indicated coordinate
- *insert*: search, insert as new leaf.
- *delete*: search, remove leaf and unary parents.

Problem: After insert or delete, the split might no longer be at exact median and the height is no longer guaranteed to be $O(\log n)$ even for points in general position.

This can be remedied by allowing a certain imbalance and re-building the entire tree when it becomes too unbalanced. (No details.)

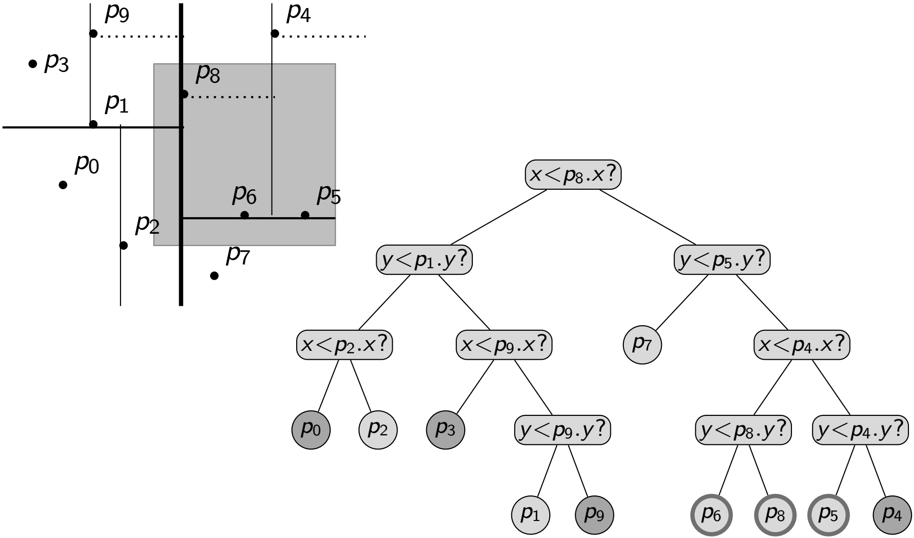
kd-tree Range Search

- Range search is *exactly* as for quad-trees, except that there are only two children.

```
kdTree::RangeSearch(r ← root, A)
r: The root of a kd-tree, A: Query rectangle
    R ← region associated with node r
    if (R ⊆ A) then report all points below r; return
    if (R ∩ A is empty) then return
    if (r is a leaf) then
        p ← point stored at r
        if p is in A return p
        else return
    for each child v of r do
        kdTree::RangeSearch(v, A)
```

- We assume again that each node stores its associated region.
- To save space, we could instead pass the region as a parameter and compute the region for each child using the splitting line.

kd-tree: Range Search Example



Red: Search stopped due to $R \cap A = \emptyset$. Green: Search stopped due to $R \subseteq A$.

kd-tree: Range Search Complexity

- The complexity is $O(s + Q(n))$ where
  - s is the output-size
  - $Q(n)$ is the number of "boundary" nodes (blue):
    - ★ *kdTreeRangeSearch* was called.
    - ★ Neither $R \subseteq A$ nor $R \cap A = \emptyset$

- **Can show:** $Q(n)$ satisfies the following recurrence relation (no details):

$$Q(n) \leq 2Q(n/4) + O(1)$$

- This solves to $Q(n) \in O(\sqrt{n})$
- Therefore, the complexity of range search in kd-trees is $O(s + \sqrt{n})$

kd-tree: Higher Dimensions

- kd-trees for d-dimensional space:
  - At the root the point set is partitioned based on the first coordinate
  - At the subtrees of the root the partition is based on the second coordinate
  - At depth $d-1$ the partition is based on the last coordinate
  - At depth d we start all over again, partitioning on first coordinate
- **Storage**: $O(n)$
- **Height**: $O(\log n)$
- **Construction time**: $O(n \log n)$
- **Range query time**: $O(s + n^{1-1/d})$

This assumes that points are in general position and d is a constant.

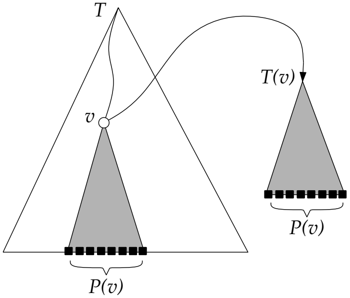
Outline

Towards Range Trees

- Both Quadtrees and kd-trees are intuitive and simple.
- But: both may be very slow for range searches.
- Quadtrees are also potentially wasteful in space.

New idea: **Range trees**

- Somewhat wasteful in space, but much faster range search.
- Have a binary search tree T (sorted by x-coordinate); this is the **primary structure**
- Each node v of T has an **associate structure** $T(v)$: a binary search tree (sorted by y-coordinate)



- Must understand first: How to do (1-dimensional) range search in binary search tree?

BST Range Search

BST::RangeSearch$(r \leftarrow root, k_1, k_2)$
r: root of a binary search tree, k_1, k_2: search keys
Returns keys in subtree at r that are in range $[k_1, k_2]$

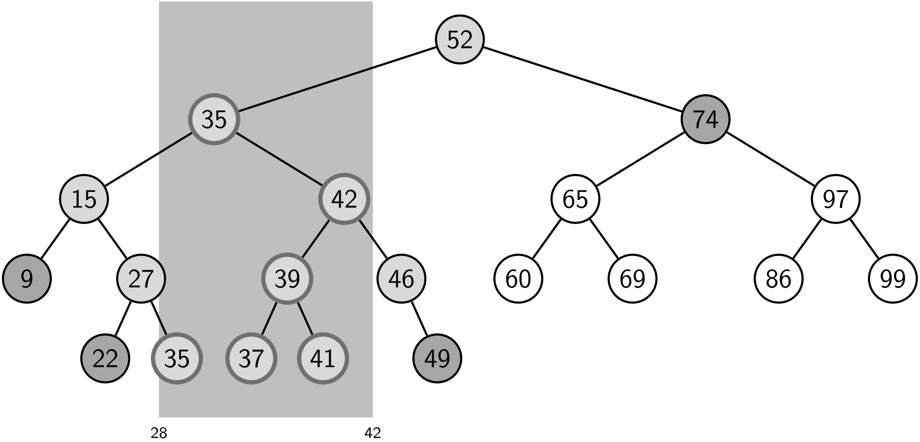
```
if r = NIL then return
if k1 ≤ r.key ≤ k2 then
    L ← BST::RangeSearch(r.left, k1, k2)
    R ← BST::RangeSearch(r.right, k1, k2)
    return L ∪ r.{key} ∪ R
if r.key < k1 then
    return BST::RangeSearch(r.right, k1, k2)
if T.key > k2 then
    return BST::RangeSearch(r.left, k1, k2)
```

Keys are reported in in-order, i. e., in sorted order.

Note: If there are *duplicates*, then this finds all copies that are in range. (Normally dictionaries do not contain duplicates, but we will soon apply this as part of range trees where duplicates may occur.)

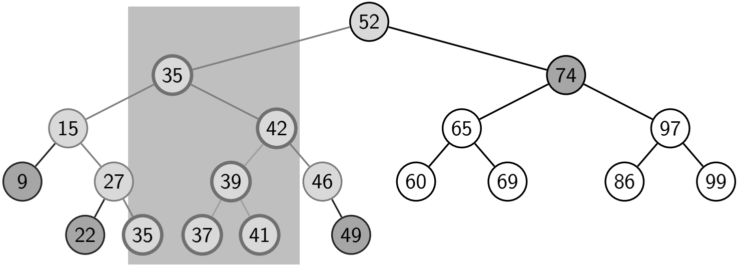
BST Range Search example

BST::RangeSearch$(T, 28, 42)$



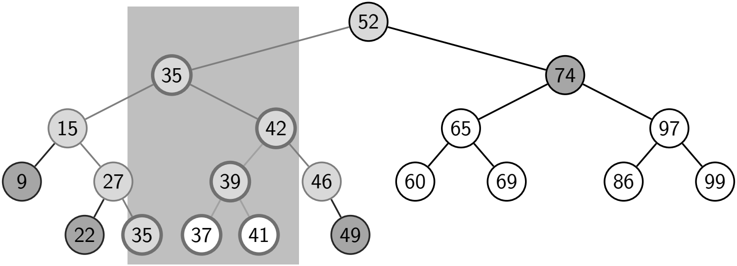
Note: Search from 39 was unnecessary: *all* its descendants are in range.

BST Range Search re-phrased



- Search for left boundary k_1: this gives path P_1
  In case of equality, go *left* to ensure that we find all duplicates.
- Search for right boundary k_2: this gives path P_2
  In case of equality, go *right* to ensure that we find all duplicates.
- This partitions T into three groups: outside, on, or between the paths.

BST Range Search re-phrased

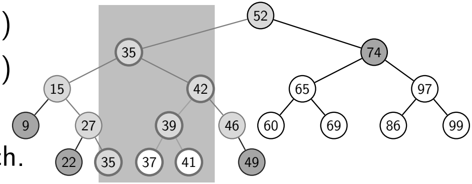


- boundary nodes: nodes in P_1 or P_2
  - For each boundary node, test whether it is in the range.
- outside nodes: nodes that are left of P_1 or right of P_2
  - These are *not* in the range, we stop the search at the topmost.
- inside nodes: nodes that are right of P_1 and left of P_2
  - We stop the search at the topmost (**allocation node**).
  - All descendants of an allocation node are *in* the range.
    For a 1d-range-search, report them.

BST Range Search analysis

Assume that the binary search tree is balanced:

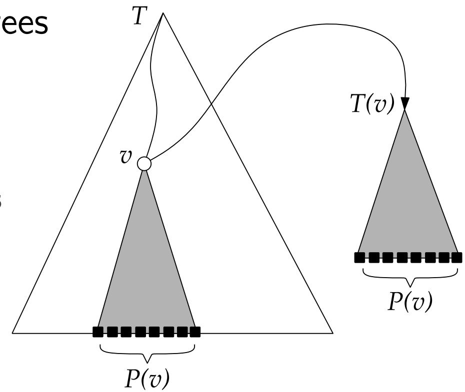
- Search for path P_1: $O(\log n)$
- Search for path P_2: $O(\log n)$
- $O(\log n)$ boundary nodes
- We spend $O(1)$ time on each.



- We spend $O(1)$ time per topmost outside node.
  - They are children of boundary nodes, so this takes $O(\log n)$ time.
- We spend $O(1)$ time per allocation node v.
  - They are children of boundary nodes, so this takes $O(\log n)$ time.
- For 1d-range-search, also report the descendants of v.
  - We have $\sum_{\text{allocation nodes } v} \#\{\text{descendants of } v\} \leq s$ since subtrees of allocation nodes are disjoint. So this takes time $O(s)$ overall.

Run-time for 1d-range-search: $O(\log n + s)$. This is no faster overall, but allocation nodes will be important for 2d-range-search.

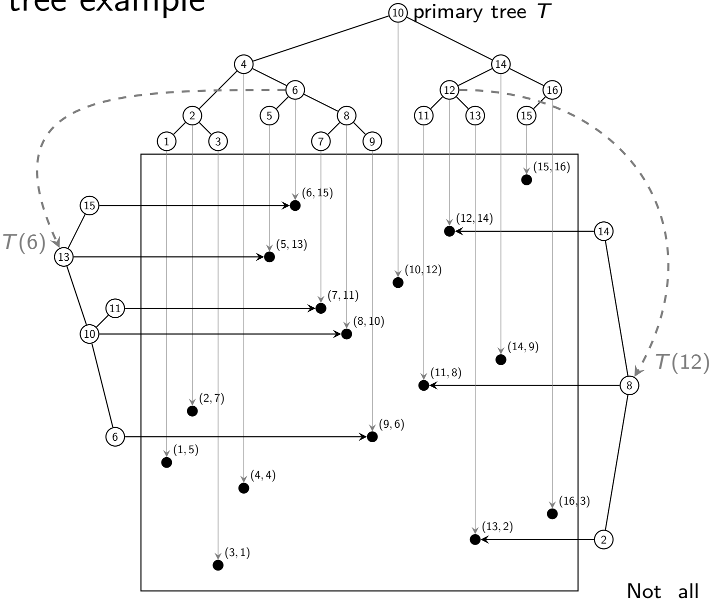
2-dimensional Range Trees



A range tree is a **tree of trees** (a *multi-level* data structure)

- **Primary structure**: Balanced binary search tree T that stores P and uses *x-coordinates* as keys.
- Each node v of T stores an **associate structure** $T(v)$:
  - Let $P(v)$ be all points in subtree of v in T (including point at v)
  - $T(v)$ stores $P(v)$ in a balanced binary search tree, using the *y-coordinates* as key
  - Note: v is not necessarily the root of $T(v)$

Range tree example



Not all associate trees are shown.

Range Tree Space Analysis

- Primary tree uses $O(n)$ space.
- Associate tree $T(v)$ uses $O(|P(v)|)$ space (where $P(v)$ are the points at descendants of v in T)
- **Key insight**: $w \in P(v)$ means that v is an ancestor of w in T
  - Every node has $O(\log n)$ ancestors in T
  - Every node belongs to $O(\log n)$ sets $P(v)$
  - So $\sum_v |P(v)| \leq n \cdot O(\log n)$

Therefore: A range tree with n points uses $O(n \log n)$ space.

Range Trees: Dictionary Operations

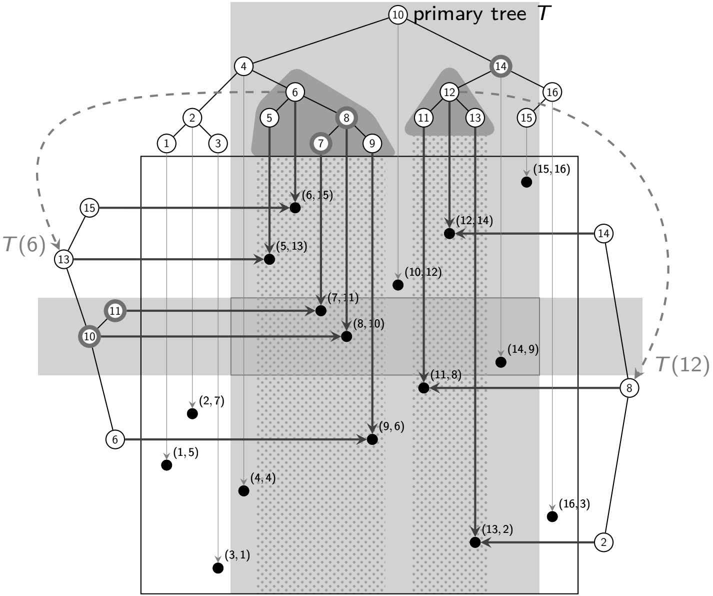
- *search*: as in a binary search tree
- *insert*: First, insert point by x-coordinate into T.
  Then, walk back up to the root and insert the point by y-coordinate in *all* associate trees $T(v)$ of nodes v on path to the root.
- *delete*: analogous to insertion
- **Problem:** We want the binary search trees to be balanced.
  - This makes *insert*/*delete* very slow if we use AVL-trees.
    (A rotation at v changes $P(v)$ and hence requires a re-build of $T(v)$.)
  - Instead of rotations, can do something similar as for kd-trees:
    Allow certain imbalance, rebuild entire subtree if violated.
    (No details.)

Range Trees: Range Search

Range search query for $A = [x_1, x_2] \times [y_1, y_2]$ is a two stage process:

- Perform a range search (on the x-coordinates) for the interval $[x_1, x_2]$ in primary tree T (*BST::RangeSearch*(T, x_1, x_2))
- Obtain boundary, topmost outside and allocation nodes as before.
- For every boundary node, test to see if the corresponding point is within the region A.
- For every allocation node v:
  - Let $P(v)$ be the points in the subtree of v in T.
  - We know that all x-coordinates of points in $P(v)$ are within range.
  - Recall: $P(v)$ is stored in $T(v)$.
  - To find points in $P(v)$ where the y-cordinates are within range as well, perform a range search in $T(v)$: *BST::RangeSearch*$(T(v), y_1, y_2)$

Range tree range search example



Range Trees: Query Run-time

- $O(\log n)$ time to find boundary and allocation nodes in primary tree.
- There are $O(\log n)$ allocation nodes.
- $O(\log n + s_v)$ time for each allocation node v,
  where s_v is the number of points in $T(v)$ that are reported
- Two allocation nodes have no common point in their trees
  $\Rightarrow$ every point is reported in at most one associate structure
  $\Rightarrow \sum_{\text{allocation node } v} s_v \leq s$

Time for range-query in range tree is proportional to

$$\sum_{\text{allocation node } v} (\log n + s_v) \in O(\log^2 n + s)$$

(There are ways to make this even faster, but they are beyond the scope of the course.)

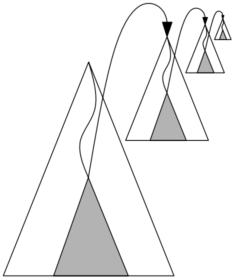
Range Trees: Higher Dimensions

- Range trees can be generalized to d-dimensional space.

Space	$O(n\,(\log n)^{d-1})$	**kd-trees**: $O(n)$
Construction time	$O(n\,(\log n)^{d-1})$	**kd-trees**: $O(n \log n)$
Range query time	$O(s + (\log n)^{d})$	**kd-trees**: $O(s + n^{1-1/d})$

(Note: d is considered to be a constant.)

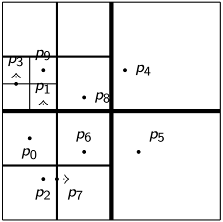
- Space/time trade-off compared to kd-trees.



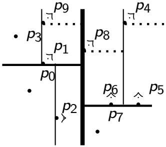
Outline

Range query data structures summary

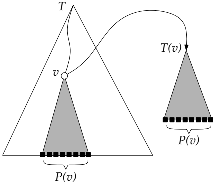
- Quadtrees
  - simple (also for dynamic set of points)
  - work well only if points evenly distributed
  - wastes space for higher dimensions



- kd-trees
  - linear space
  - query-time $O(\sqrt{n} + s)$
  - inserts/deletes destroy balance
  - care needed if not in general position



- range trees
  - query-time $O(\log^2 n + s)$
  - wastes some space
  - inserts/deletes destroy balance



Convention: Points on split lines belong to right/top side.