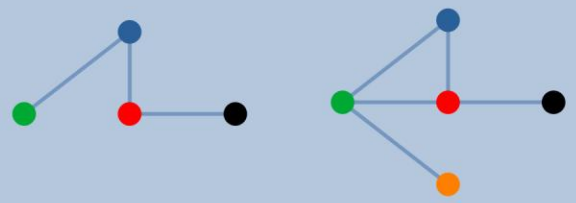
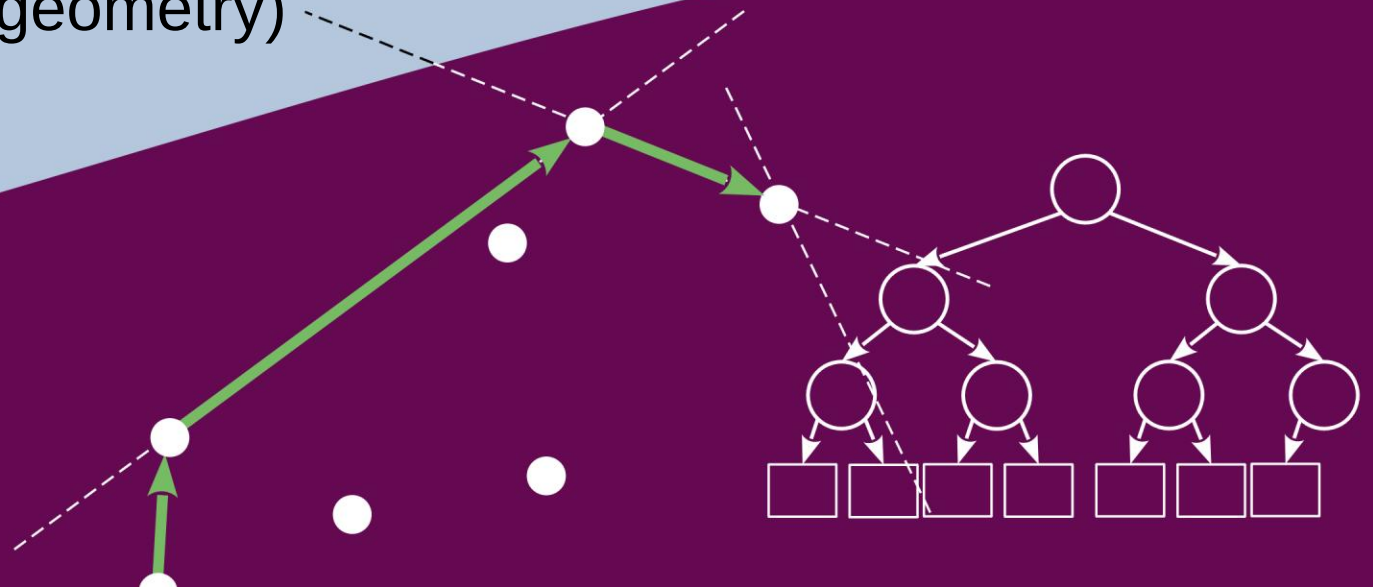
Advanced algorithms and data structures

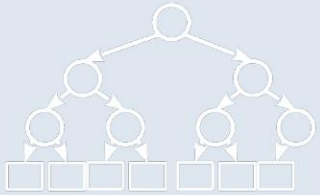
Week 4: Geometric algorithms
(computational geometry)



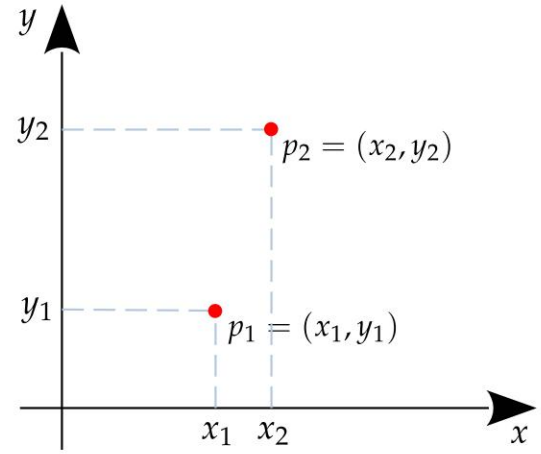


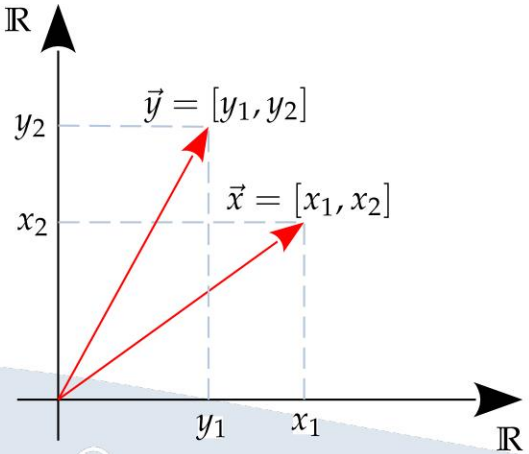
What is Computational Geometry?

- Definition: *the systematic study of data structures and algorithms in geometry*
- Examples:
  - **Computer graphics** - one of the most developed areas - computer games, visualization, modeling
  - **Spatial design of products** – layout on electronic boards, design integrated chips, packaging, **CAD/CAM**
  - **Robotics** – calculation of movements and trajectories
  - **GIS** – spatial projections, placement of objects, route calculations, object search, sections, ...



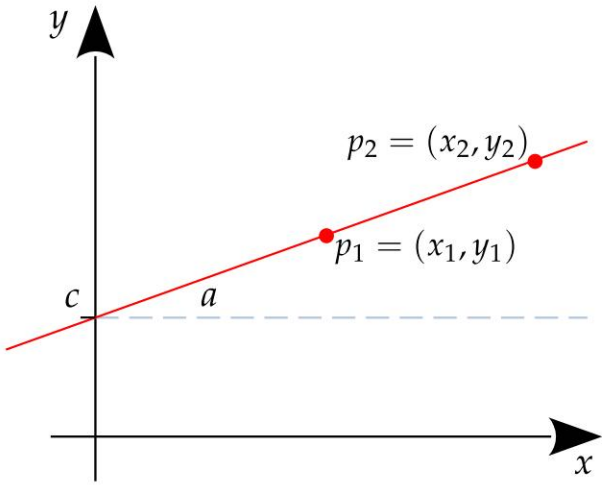
Basic geometric elements - point





- A point is an infinitesimal element of ÿ! space
  ÿ×ÿ×ÿ× ÿ×ÿ = (", #, $, … , !) =
- Real space ÿ! is abstract - we map it to Cartesian coordinate system
- A point can also be written in vector form
  ÿ = [" # $ … !]

Basic geometric elements - line



- A line is an infinite set of points in space that follow a linear equation

= {(,) : (,) ÿ ÿ# , + = }

- A linear equation can also be written as

= ÿ

- If we have two points in space through which a line passes

" = (", " , #) = (#, #)

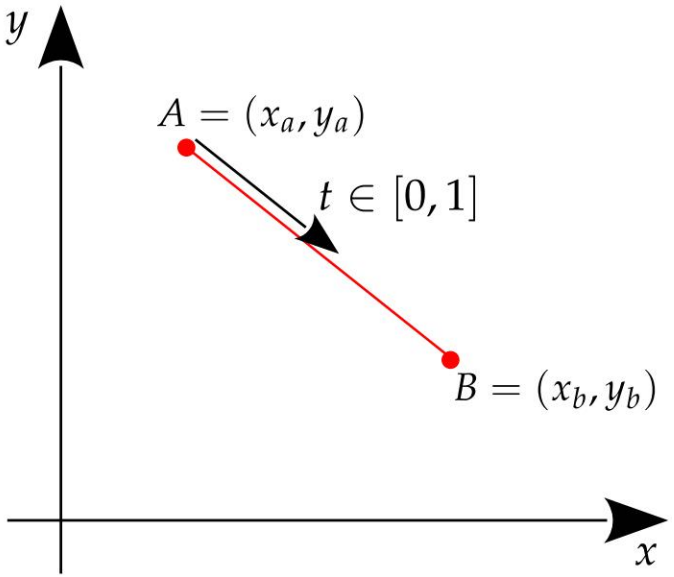
- The line can be written as

= " − # , = " + " − # "
" # " #

Basic geometric elements – line and segment

- The line can be parameterized

ÿ = - ", ÿ = - "
# #
= " + ÿ , = " + ÿ



- The parameter ÿ 0.1[determines the position of the point on the line:
  - for = 0 we are in !, while for = 1 we are in "
- We use the parametric approach to delimit the line
- The line passes through the (%, & ,) = (&, &)

points = • If we consider that the points A and B limit it

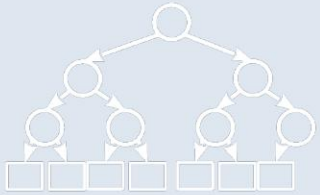
- For parameter values ÿ 0.1, we get a set of points representing line segment or length.

Basic geometric elements - surface

- Consider a geometric element that has one dimension less than space in which it is located
  - For example a point (0 dimension) on a line (1 dimension)
- In three-dimensional space, it is a surface
- A surface is an infinite set of points that follow a linear equation

= {(, ,):(, ,) ÿ ÿ$, + + = }

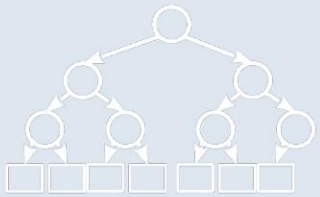
- We call a subset of points of a surface a geometric shape



Basic geometric elements – polygons

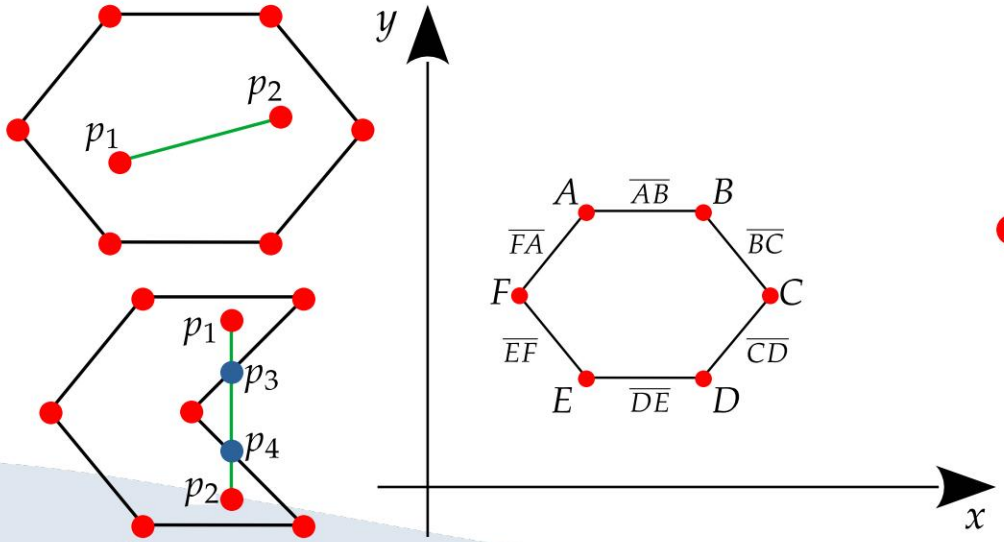
- A subset of the surface ÿ bounded by a set of points and line segments between those points is called a polygon
- A polygon composed of points is called a -gon
- The line segments bound the polygon forming the edge of the *polygon,* while the points in the polygon are called *the body of the polygon*
- When we enumerate points on the edge of a polygon, we usually do so in a clockwise direction
- The sum of the interior angles of a polygon is

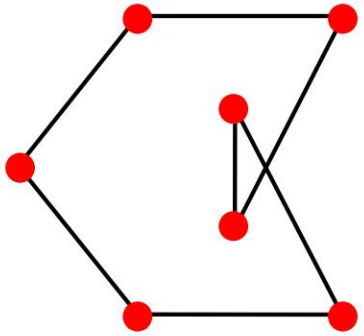
$$= ÿ(2 ÿ 180°)$$



Basic geometric elements – polygons

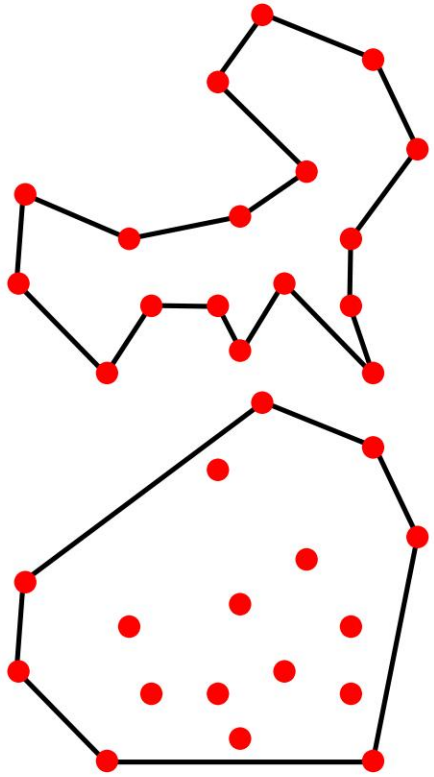
- A polygon is considered **convex** if for every pair of points ", # ÿ , length " # passes inside the polygon
- A polygon is considered **simple** if it does not interact with itself, for example intersecting with its edge





- **Regular** polygons are equiangular and equilateral, which means that their interior angles and the lengths of all line segments are equal.
  - **Example:** one-sided triangle, square, pentagon, hexagon, heptagon, ...

Convex Hull



• Let's imagine a set of points in ÿ#, (= { ", #, ... , ! }

• There is a subset of these points, such that it forms an edge of the which contains all points from (, polygon so that it is valid the set ÿ ÿ (:

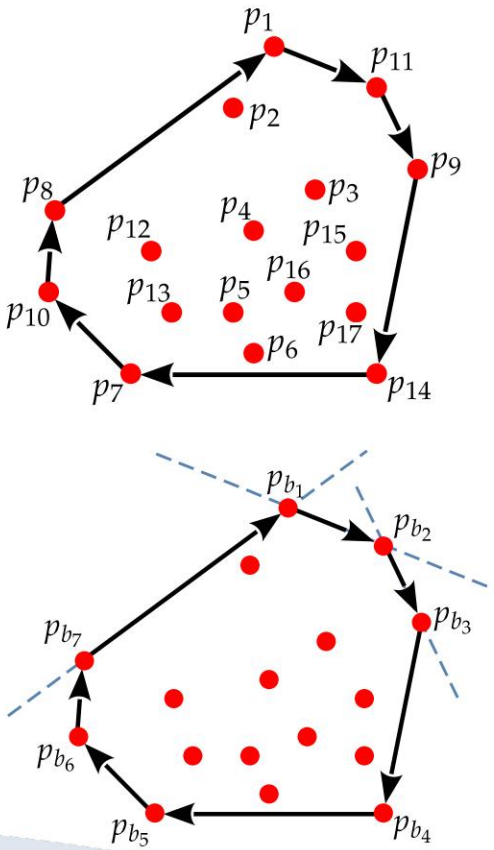
ÿ • If the polygon is convex, then it is a convex shells

• Let's imagine that • (a set of nails driven into a board

Let's put a rubber band around those nails

• The rubber band represents the edge of the polygon, which is a convex shell

Flat convex shell I

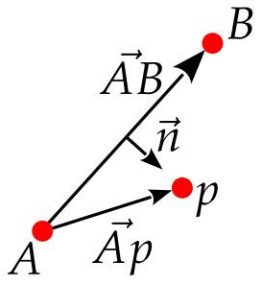


• We enumerate the points that belong to the edge of the polygon ( ) ÿ (clockwise • A simple way to determine that ( ) is a convex hull • We take each line segment of the edge of the polygon and see if they all other points in the polygon to the **right** of its line

• If yes, it is a convex shell

• How to determine that the point is **"to the right"** of the line? • And what does **"right" even mean?**

Flat convex shell I



- If we have two points = and then a vector can be defined

$$\longrightarrow = \&[\quad - \quad \% \quad \& \quad - \quad \%]$$

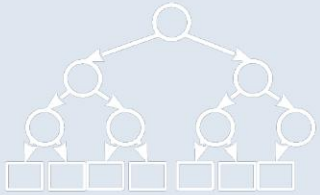
- With the system of equations

$$\longrightarrow \quad \rightarrow) \qquad \rightarrow \quad \longrightarrow)$$
$$ÿ \qquad = 0, = ÿ$$

- We get

$$= \&[\quad - \quad \% \quad \% \quad - \quad \&] \begin{bmatrix} - & \% \\ - & \% \end{bmatrix}$$
$$= ÿ \quad (\quad \% \&)(\quad - \quad \%) \; - \; (\quad \%)(\quad \& \quad - \quad \%)$$

- For everything > 0, we consider that the point is to the **right** of the line segment

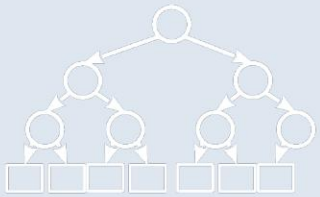


Flat convex shell I

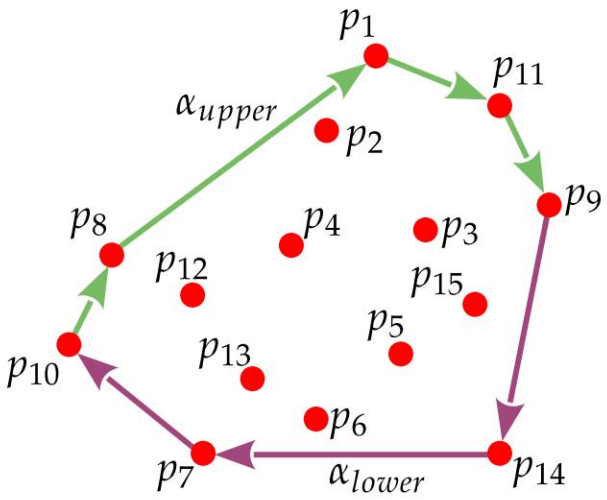
```
function SimpleConvexHull(P_p)
    α(P_p) ← ∅
    for each pair of points p_i, p_j ∈ P_p do
        bound ← 1
        for each point p_k ∈ P_p such that p_k ∉ {p_i, p_j} do
            if p_k is left from the line segment p_i p_j then
                bound ← 0
        if bound is 1 then
            add p_i p_j to α(P_p) taking care of the clockwise order
    return α(P_p)
```

Complexity = (!)

- Given that at the beginning we only !, have to find an edge ( ) such that it represents a convex shell, which is marked with ( )
- We take pairs of points from ", # ! Ÿ and we check if they are to the right of ! all other points from of their line segment " #
- If yes, then that line segment " # belongs to ( )
- Two big problems:
  - The algorithm does not ensure that the convex hull is closed
  - High complexity of the solution



Flat convex shell II



- Is it possible to have a different, faster algorithm for finding the convex shells?
- Let's divide the convex shell into an upper and a lower part

( ) = %&'() ÿ( ) *++() ( )

- Horizontally sort the points in

ÿ(+) = { + : ,! ÿ ,! + }

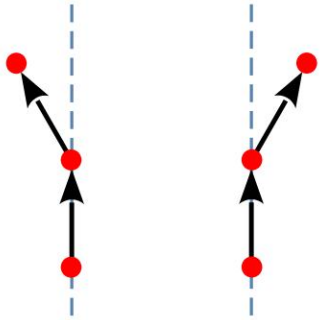
ÿ ÿ ,! , , " ÿ (+) : < , (,! ÿ • (,")

We notice that the first point in ÿ is the beginning, and the last point is the end of the upper part of the convex shell or

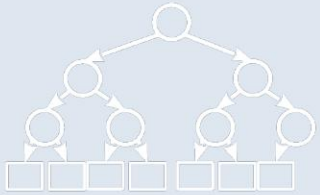
- Equally, only in ÿ(+) ( ) *++() reversed, the last point in ÿ is the beginning, and the first point in ÿ is the end of the lower part of the convex shell or

%&'() ( ) (+) ( )

Flat convex shell II



- We continue to go around both parts of the convex shell in a clockwise direction
- This means that the convex shell, and thus both of its parts, is inclined to the **right**
- In order to determine that part of the convex shell is inclined to the right, we need at least three points
- We use the same principle of determining whether the third point in a row is to the right or to the left of the line formed by the first two points

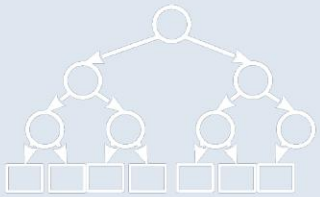


Flat convex shell II

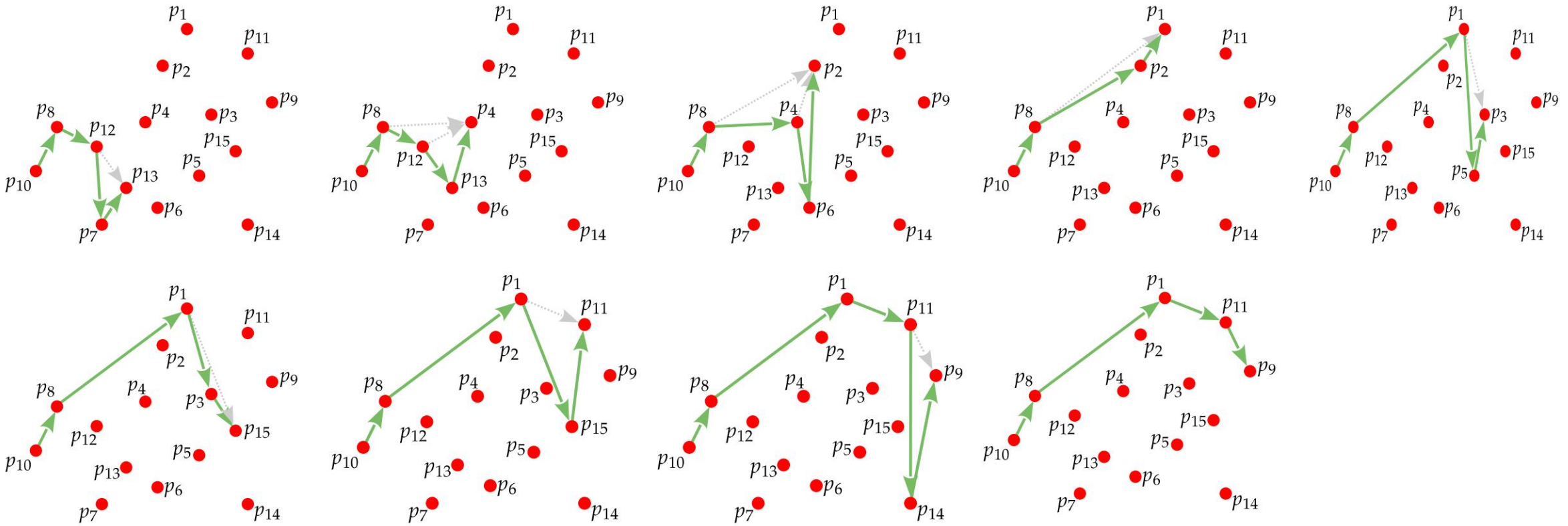
function CONVEXHULL(P_p)
  create $h(P_p)$ as the sorted list of points from P_p
  $\alpha_{upper}(P_p) \leftarrow \{p_{h_1}, p_{h_2}\}$ from $h(P_p)$
  for $i \leftarrow 3$ **to** n **do**
    add p_{h_i} to $\alpha_{upper}(P_p)$
    while $|\alpha_{upper}(P_p)| \geq 3$ and last three points incline left **do**
      remove the point before the last from $\alpha_{upper}(P_p)$
  $\alpha_{lower}(P_p) \leftarrow \{p_{h_n}, p_{h_{n-1}}\}$ from $h(P_p)$
  for $i \leftarrow n-2$ **downto** 1 **do**
    add p_{h_i} to $\alpha_{lower}(P_p)$
    while $|\alpha_{lower}(P_p)| \geq 3$ and last three points incline left **do**
      remove the point before the last from $\alpha_{lower}(P_p)$
  return $\alpha_{upper}(P_p) \cup \alpha_{lower}(P_p)$

Complexity = (log)

- First, we sort the list of points horizontally
- We add the first two points from ÿ to (!) in the upper convex shell
- In the loop, we start from the third point
  - If the last three points lean to the left, we remove the middle one from the upper convex shell
- When we reach the last point in, we get ÿ (! ,) the upper convex one shell
- For the lower one, the same procedure is done, but in reverse order through (!)



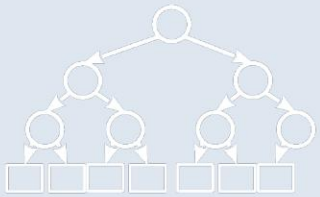
Flat convex shell II



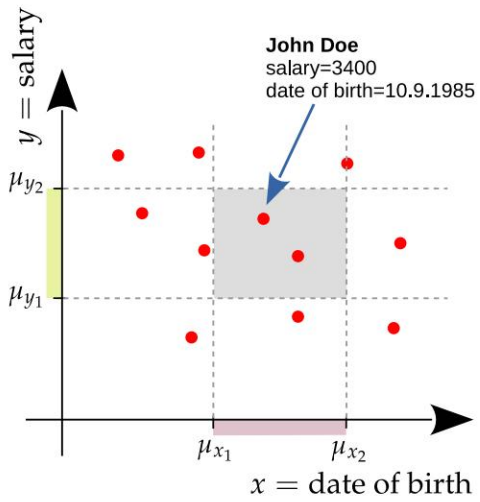
Flat convex shell II

- The algorithm goes through all n points from (, which would be the complexity
  ( )
- Since we need to use one of the sorting algorithms to create ÿ, the complexity of the sorting algorithm prevails and the total complexity is (log)

- The convex hull concept is used later in linear programming

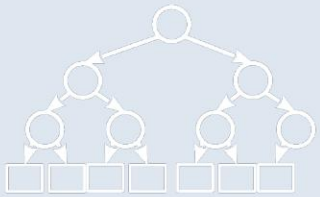


Geometric search

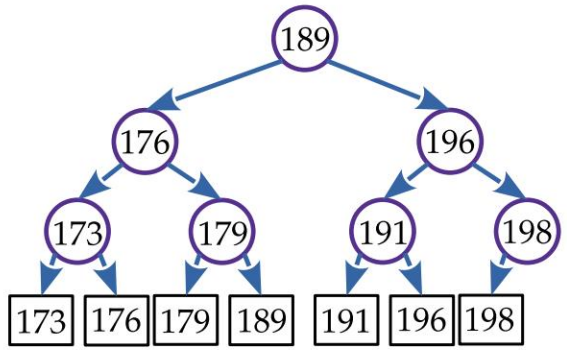


- Let's imagine that we have a set of data in the database that we can place in -dimensional space
- Some of the dimensions can be *height, weight, salary,* etc...
- Let's forget all the possibilities of the database for a moment
- B+ index trees help us retrieve data for a specific attribute value

- How to retrieve a dataset that has a specific attribute in a certain range of values?
- For example: let's find all people whose height is between 165 and 184 centimeters
- In general, we have a set of values (1-dimensional) from which we want to find values in the required range as quickly as possible



Range Search (1D)

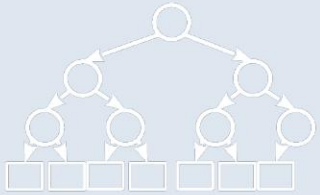


- We have a set of values , and the range we are [$, %]

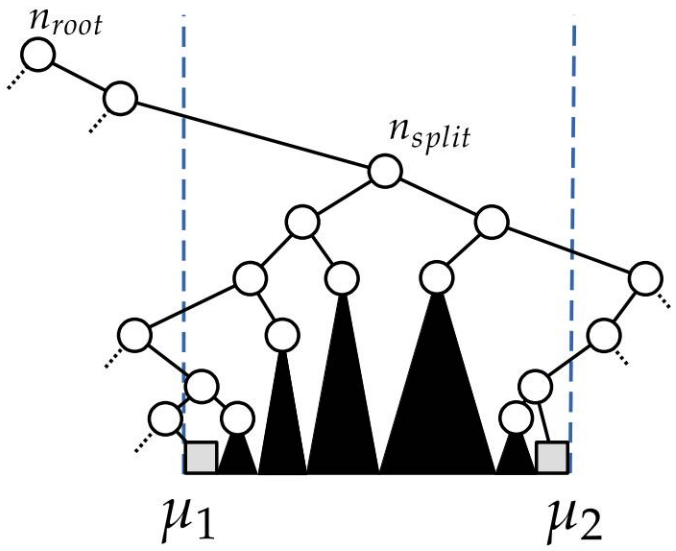
looking for • For example = {173, 176, 179, 189, 191, 196, 198}

- How to organize a set of values so that we have the ability to find the range $, % in the [most efficient way possible?
- We resort to using binary trees
- In this case, we use binary trees that have concrete values in their leaves - we call them **range trees**
  - Leaves have values from •

  Internal nodes are guiding *and* do not need to have the same values as leaves
  - Principle similar to that of B+ trees

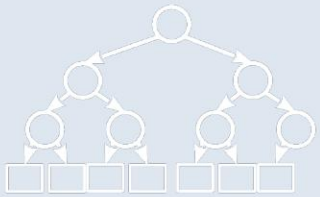


Range Search (1D)

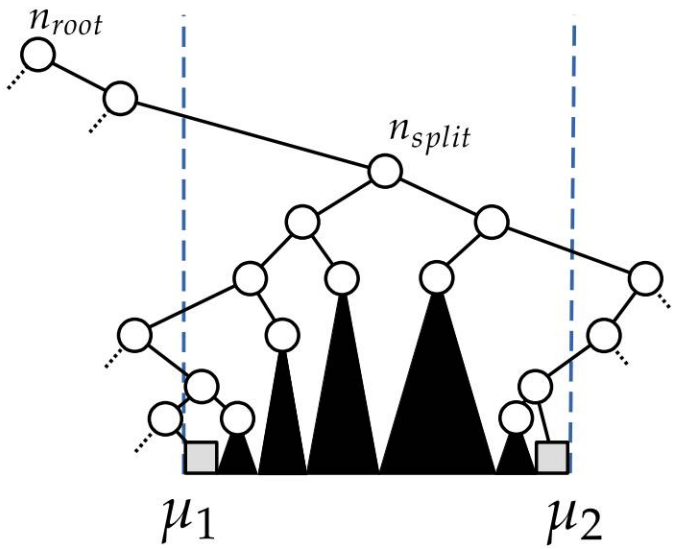


- We have a binary tree such that each internal node has a value such that:
  - In the left subtree are all values that are ÿ
  - In the right subtree are all values that are >

1. We start from the root node by looking for the first node that is in the required range [", #] the so-called *separating node* *(+,-
2. From the separating node, we move to *(+,+) to the left and to the right, thus obtaining two vertical paths from the node to the leaf: ▪ /



Range Search (1D)



3. Moving to the left, we know that the nodes are smaller than and we have two options:

a) The node is within the required range !, " , which means

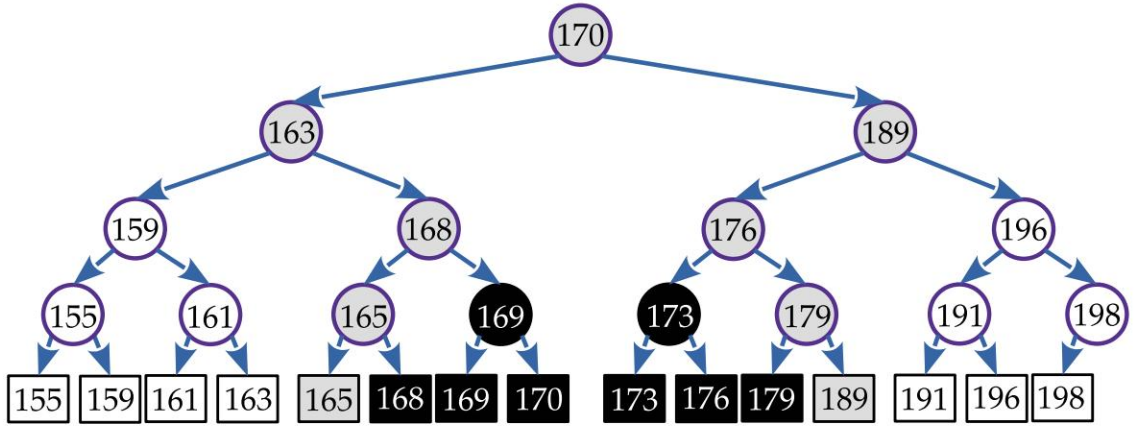
- Its right subtree is certainly in the required range and we automatically add all values to the result *(pruning).*
- We are not sure about its left subtree, so we move to left child

b) The node is not within the required range, which means

- Its left subtree is definitely not in range and we ignore it
- We are not sure about its right subtree, so we move to the right child

4. By moving to the right, we perform mirror operations

Range Search (1D)



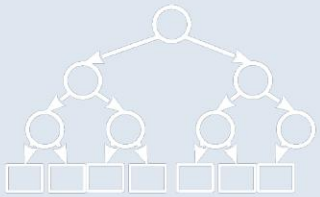
Complexity = (+ log)

- Example of tree and search range 165,184 [ ]
- The root node is also • The gray nodes are tested #$%&'
- Black nodes are automatically added to the result *(pruned)*
- White nodes are discarded as out of range
- Search complexity is (+ log), where is the number of automatically added values, and log is two passes through the binary tree

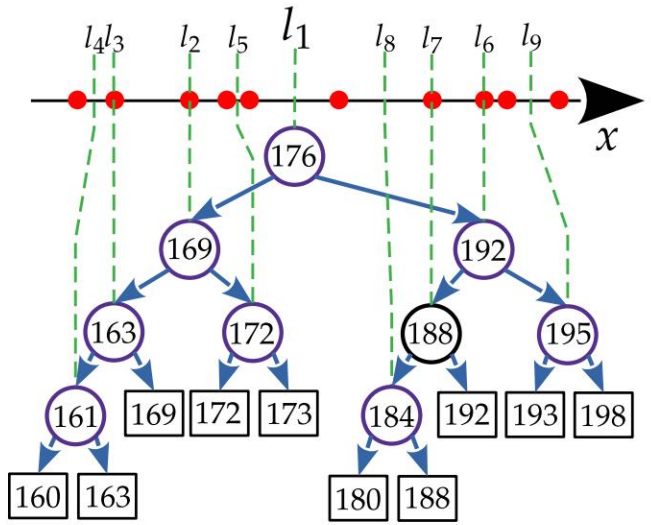
Range Search (1D)

```
function 1DRangeQuery(τ, μ1, μ2)
    rv ← ∅
    n_split ← FindSplittingNode(τ, μ1, μ2)
    if n_split is leaf then
        if v(n_split) is in the range [μ1, μ2] then
            add n_split to rv
    else
                                                ▷ left traversal
        n ← leftChild(n_split)
        while n is not leaf do
            if μ1 ≤ v(n) then
                add ReportSubtree(rightChild(n)) to rv
                n ← leftChild(n)
            else
                n ← rightChild(n)
        if n is leaf and v(n) is in the range [μ1, μ2] then
            add n to rv
                                                ▷ right traversal
        n ← rightChild(n_split)
        while n is not leaf do
            if v(n) ≤ μ2 then
                add ReportSubtree(leftChild(n)) to rv
                n ← rightChild(n)
            else
                n ← leftChild(n)
        if n is leaf and v(n) is in the range [μ1, μ2] then
            add n to rv
    return rv
```

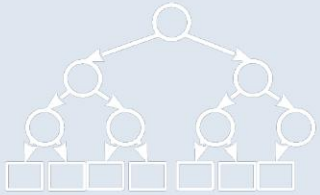
```
function FindSplittingNode(τ, μ1, μ2)
    n ← root(τ)
    while n is not leaf and (μ1 ≥ v(n) or μ2 ≤ v(n)) do
        if μ2 ≤ v(n) then
            n ← leftChild(n)
        else
            n ← rightChild(n)
    return n
```



Range Search (1D)



- How to create a binary tree that reflects a specific set of values ?
- With binary trees, we learned how to create a tree based on a sorted set of values - thoughtfully
- A similar principle is used here
  - The created node contains the median of all values that are in to its subtree - we want the tree to be balanced
  - Its left subtree contains only values that are ÿ of the value of the node
  - Its right subtree contains only values that are > of the value of the node
  - Using a recursive procedure, we create a binary tree up to its leaves - concrete values

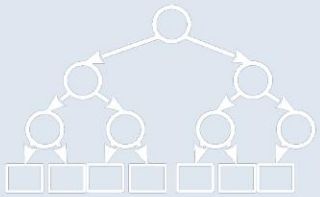


Range Search (1D)

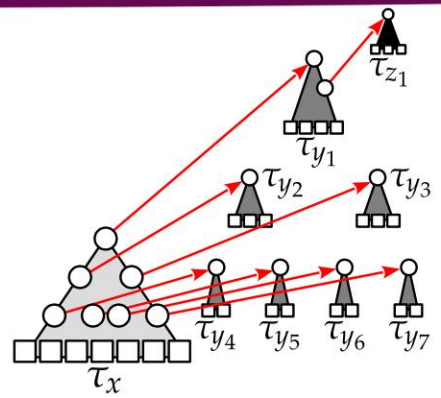
```
function CREATE1DRANGETREE(P_p)
    if |P_p| = 1 then
        return n ← leaf having the value from P_p
    else
        v_m ← median(P_p)
        P_{p_left} ← {p_i : p_i ∈ P_p ∧ p_i ≤ v_m}
        P_{p_right} ← {p_i : p_i ∈ P_p ∧ p_i > v_m}
        n_left ← CREATE1DRANGETREE(P_{p_left})
        n_right ← CREATE1DRANGETREE(P_{p_right})
        n ← create node having value v_m
        leftChild(n) ← n_left
        rightChild(n) ← n_right
    return n
```

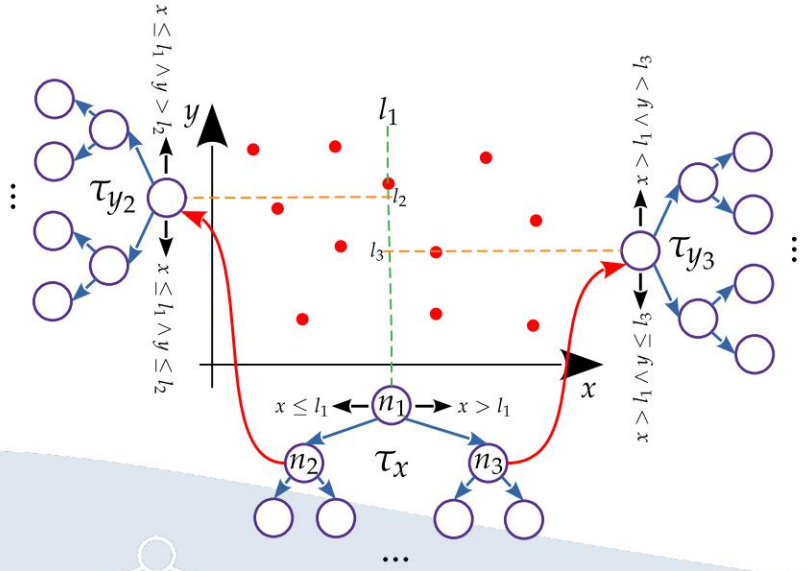
1. If we got only one value
   a) Let's return the node with that value (leaf)
2. If we got more values
   a) We calculate the median for the root node
   b) Divide the values into left (ÿ) and right (>)
   c) Recursively create left and right subtrees
   d) Place the created subtrees as left and right a child of the root node

- The complexity of creating a range tree is (log) for sorting due to searching for the median



Range Search (2D)



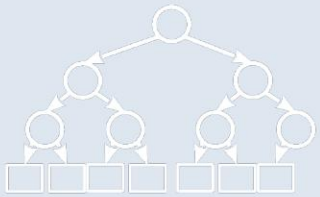


- To each node of the range tree, we add an subtree • %**01 additional dimension ( )
  This chaining can be continued for other dimensions
- %**01 represents a range tree for all points represented by a node but for a higher dimension ,, if , for -os, then it is formed for -os %**01
- Search complexity for two dimensions is now (+ log#) for chaining trees
- The complexity of creation is still (log) for sorting

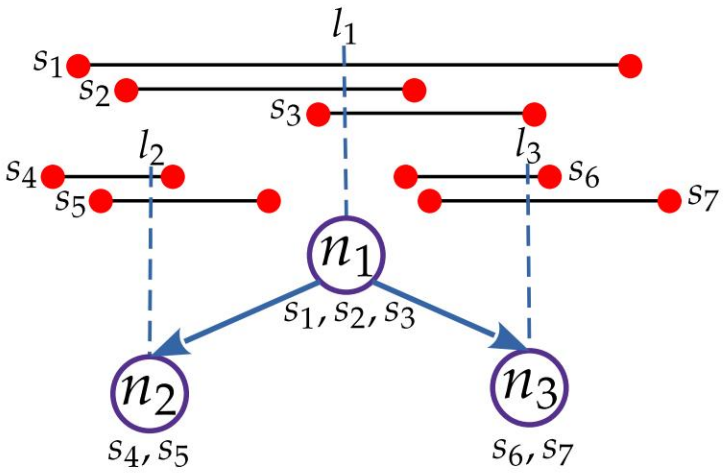
Interval search (1D)

- Let's imagine a set of intervals = { , = ([,!, ," ,] ,) ,!, ,"", , ÿ ÿ } which are parallel to the -axis – actually horizontal line segments
- Dots (,!, ,) (,", , are called the endpoints of the interval
- We would like to have a structure in which we save a set of intervals, so that we can quickly and efficiently find the intervals that are in the query window

  2 = [3!, 3"] × [4!, 4"]
- The solution is in a composite binary tree that stores values in nodes - we call them **interval trees**
- In this case, we do not have sheets representing specific values

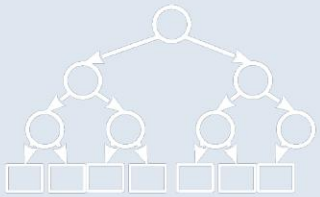


Interval search (1D)

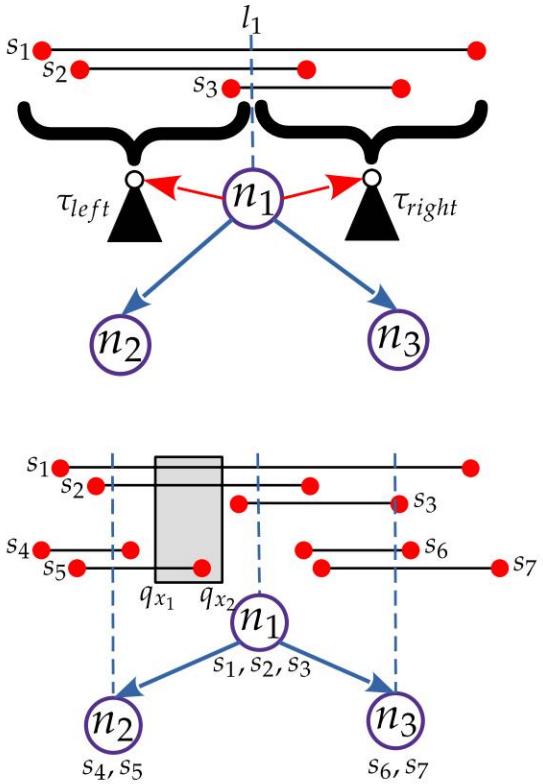


- We calculate the median) of all endpoints of the interval from the set of intervals for which we create a tree
  - For the root node the set of intervals is •

  For each node below the root it is a subset of – the set of intervals stored in the parent
- Let's create a node with the value of the calculated median)
  - Let's save all the intervals that intersect • U in the node (! this is how we save the intervals I(!) = { !, ",) }
- We recursively create left and right subtrees
  - All intervals that are both extreme go into the left subtree points < (
  - All intervals that are both extreme go into the right subtree points > (



Interval search (1D)



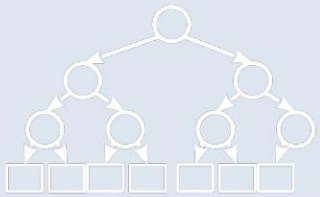
- Two additional data structures are added to each node (2D range trees for example)
  - In the left #$%& , the left endpoints of the intervals saved in that node (ÿ) are stored
  - In the right node (>) the right endpoints of the intervals saved in that '()*&
  - Endpoints in '()*& trees have back references to intervals in nodes to avoid any additional searching
- We do this because the search results in two possible scenarios:
  1. The interval completely intersects, which means that its left endpoint is < and its right endpoint is +!, [+!, +"]
  2. An interval has a beginning or end at [+!, +" ,] which means that it is left or right an end point in that range
  3. We should also make sure that the interval is in the range [,!, ,"]
- This can be determined through additional structures %*+' or ,&-.'
  - That is why the interval tree is called composite

Interval search (1D)

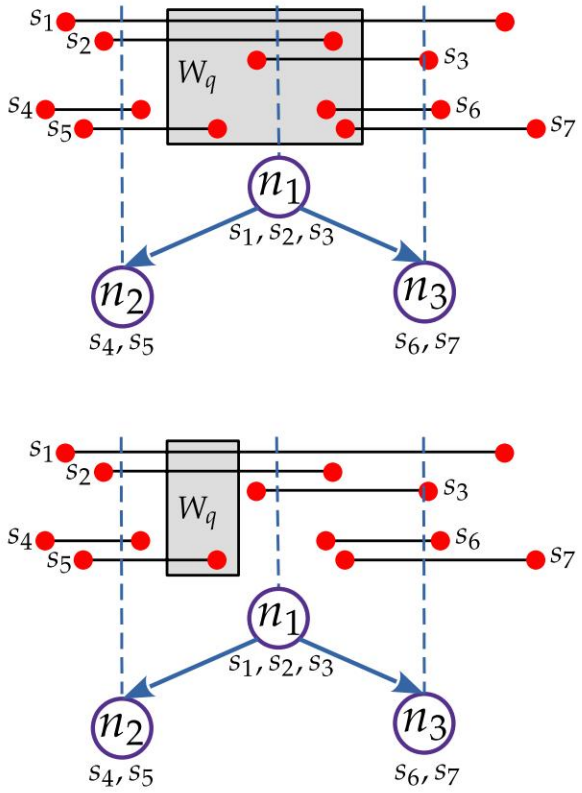
function CREATEINTERVALTREE(I)
  if $I = \emptyset$ **then**
    $n \leftarrow$ empty leaf
  else
    $x_{med} \leftarrow median(endpoints(I))$
    $I_{left} \leftarrow \{s_i : s_i \in I \wedge x_{i_1}(s_i) < x_{med} \wedge x_{i_2}(s_i) < x_{med}\}$
    $I_{right} \leftarrow \{s_i : s_i \in I \wedge x_{i_1}(s_i) > x_{med} \wedge x_{i_2}(s_i) > x_{med}\}$
    $I_{med} \leftarrow I \setminus (I_{left} \cup I_{right})$
    $n \leftarrow$ create a node having value x_{med}
    $intervals(n) \leftarrow I_{med}$
    $leftChild(n) \leftarrow$ CREATEINTERVALTREE(I_{left})
    $rightChild(n) \leftarrow$ CREATEINTERVALTREE(I_{right})
    if $I_{med} \neq \emptyset$ **then**
      $P_{left} \leftarrow \{((x_{i_k}(s_i), y_i(s_i)), s_i) : s_i \in I_{mid} \wedge x_{i_k}(s_i) \leq x_{med}\}$
      $P_{right} \leftarrow \{((x_{i_k}(s_i), y_i(s_i)), s_i) : s_i \in I_{mid} \wedge x_{i_k}(s_i) > x_{med}\}$
        ▷ We include the back reference to s_i
      $\tau_{left}(n) \leftarrow$ CREATE2DRANGETREE(P_{left})
      $\tau_{right}(n) \leftarrow$ CREATE2DRANGETREE(P_{right})
  return n

- Due to the need for sorting, the basic interval tree has log (complexity • If we add to that

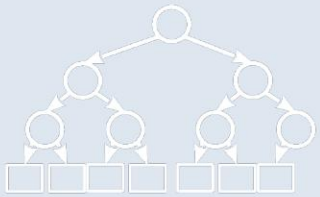
two range trees per log 2 log node, is ultimately log# (((we get))(t • What))
( )



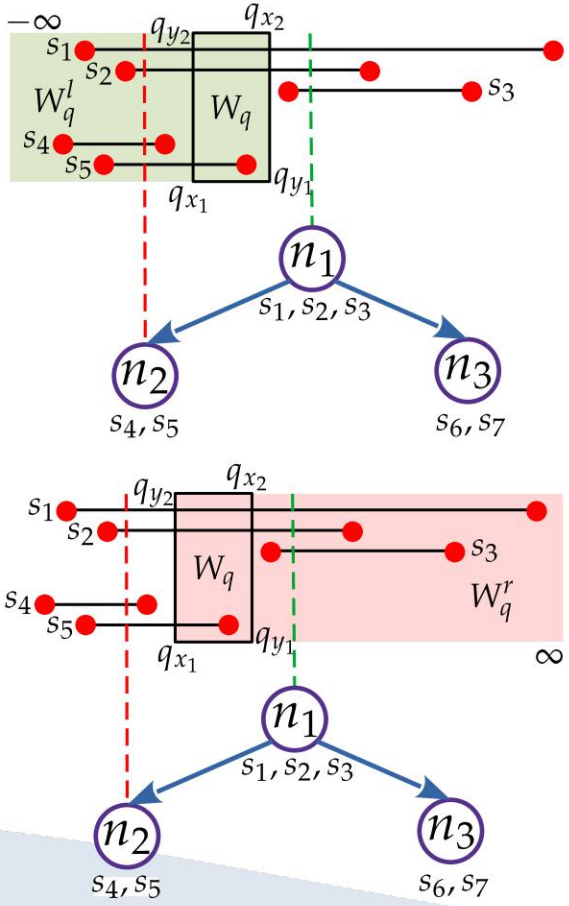
Interval search (1D)



- Searching an interval tree is similar to searching a range tree. If we are currently in node 2 (starting from node 1), then we have three basic cases:
  - the median of node !"# $ is within the range %!, %" [ ]
    - both trees are consulted in order to select intervals that '()*&, and #$%& end in window +!, +" [ ] × [,!, ,"]
  - the median node !"# $( ) is on the left side of the range %!, %" [ ]
    - the right additional tree is consulted in order to select the intervals that '()*&, end on the right side in the window +!, +" [ ] × [,!, ,"]
    - for the left subtree of the node, we consider that there are no required intervals
    - we continue in the right subtree
  - the median !"# $( ) is to the right of the range %!, %" [ ] node • the left additional tree is %&, × in order to select the intervals that #$ consulted, the left side ends in the window]+!, +" [,!, ,"
    - for the right subtree of the node (we consider that there are no required intervals
    - we continue in the left subtree



Interval search (1D)



- But what happens to intervals that completely intersect the range, like say # in the example? "
- Instead of searching endpoints exclusively in the range 3!, 3" , let's expand the search [ ]
  - We search for the left endpoints in the window (ÿÿ, 3-]× [4., 4-]
  - We search for the right endpoints in the window [3rd, ÿ)× [4., 4-]

Interval search (1D)

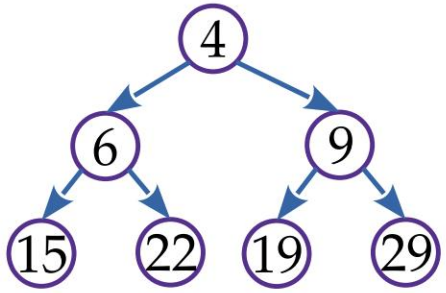
function $\text{IntervalQuery}(n, [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}])$
  $iv \leftarrow \emptyset$
  if $q_{x_1} \le x_{med}(n) \le q_{x_2}$ **then**
    $p \leftarrow \text{2DRangeQuery}(\tau_{left}(n), (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}])$
    $p \leftarrow p \cup \text{2DRangeQuery}(\tau_{right}(n), [q_{x_1}, \infty) \times [q_{y_1}, q_{y_2}])$
    $iv \leftarrow$ all intervals of the endpoints in p
    $move \leftarrow both$
  else if $x_{med}(n) < q_{x_1}$ **then**
    $p \leftarrow \text{2DRangeQuery}(\tau_{right}(n), [q_{x_1}, \infty) \times [q_{y_1}, q_{y_2}])$
    $iv \leftarrow$ all intervals of the endpoints in p
    $move \leftarrow right$
  else if $q_{x_2} < x_{med}(n)$ **then**
    $p \leftarrow \text{2DRangeQuery}(\tau_{left}(n), (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}])$
    $iv \leftarrow$ all intervals of the endpoints in p
    $move \leftarrow left$
  if $move \in \{both, left\}$ and exists $lc \leftarrow leftChild(n)$ **then**
    $iv \leftarrow iv \cup \text{IntervalQuery}(lc, [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}])$
  if $move \in \{both, right\}$ and exists $rc \leftarrow rightChild(n)$ **then**
    $iv \leftarrow iv \cup \text{IntervalQuery}(rc, [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}])$
  return iv

- Given that we have two levels of trees, the first level is an interval tree, and the second is two two-dimensional range trees, search complexity 2 log# log =
  ((log$ • For some other data)) (structures, it can)

be different. i +9:- ;,<=- Let's say is a log for an ordinary list it

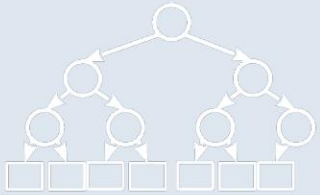
( )

Priority *tree*



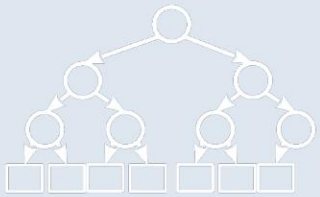
- Using a two-dimensional span tree in an interval tree seems inefficient – a two-dimensional span tree has two layers (three in total with the interval tree)
- Interval tree queries are specific: (ÿÿ, 0/]× 10, 1/ [ ] [00, ÿ)× [10, 1/]
- A *heap* can be used as data for this purpose structure
  - A heap is a sorted structure, where the root of the heap is the smallest or the largest member
  - We are talking about an ascending or descending pile
- If we sort the points along the -axis, we can store them in a pile
  - This solves the horizontal part of the query (ÿÿ, 1"] and [• 1#, ÿ)

For (ÿÿ, 1"] we need an ascending sorted heap – we descend from the root of the pile, until we come across ( ) > 1"



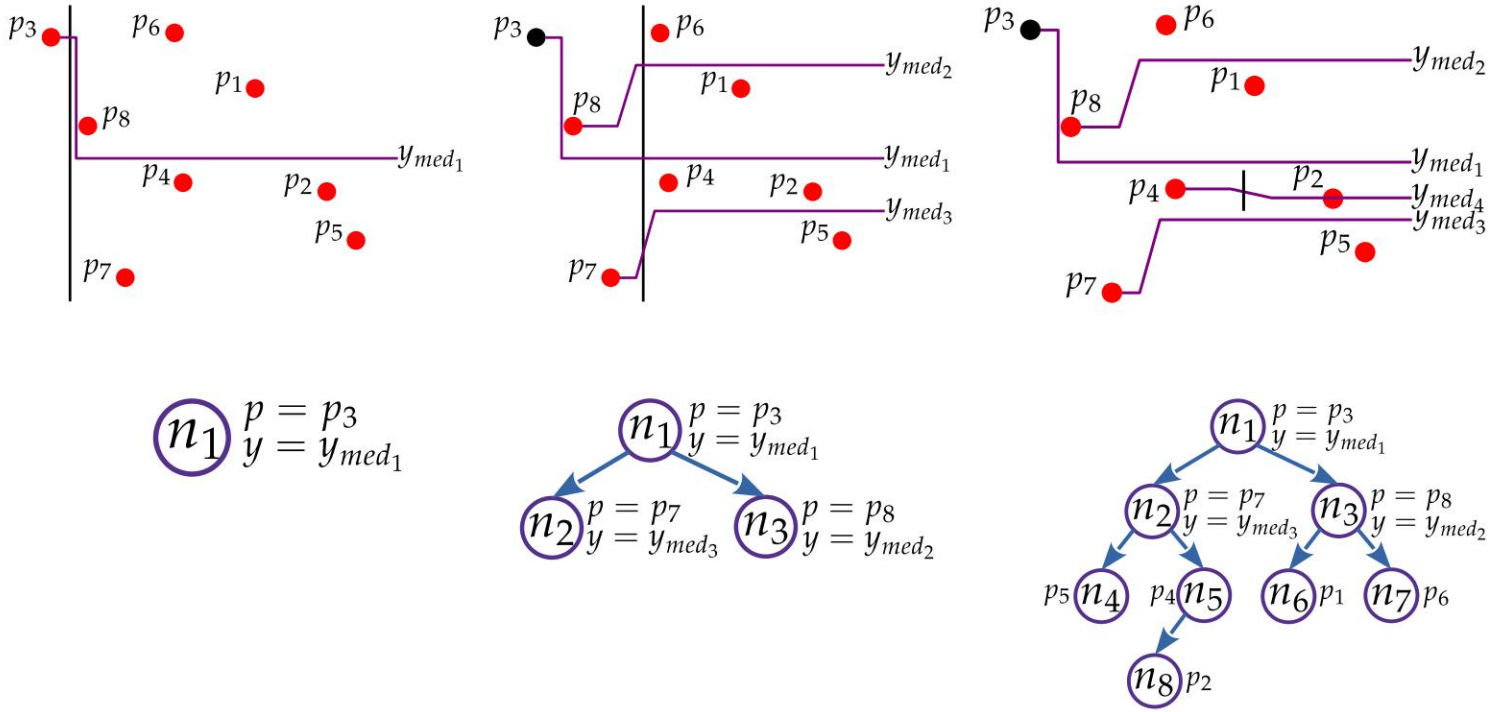
Priority tree

- What about the -axis? The crowd alone will not support that second dimension.
- We use the fact that heap partitioning can be arbitrary as long as the heap is sorted - parent smaller or larger than children
- We incorporate the concept of a binary tree - **a priority** tree - into the heap
  - We always take ( ) represents a point stored in a node - it follows the heap principle, the value of the node from the next point with the smallest ( ) value
  - We add the parameter ( ) to the node
    - The structure of the priority tree by parameter ( ) follows the principle of a binary tree
      - Left child • Right child – has parameter - has ( ) ÿ ( ) - less than or equal to the parent parameter . ( ) > ( ) - greater than the parent
    - When we create a priority tree and decide in which subtree we put the points
      - We remove the node point from the set of points
      - We calculate the median /$0 value of the remaining points - we set the parameter (n) to /$0
      - Remaining points with ( ) ÿ • /$0 they go to the left subtree
        Remaining points with > ( ) /$0 they go to the right subtree

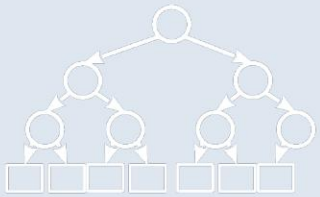


BE CAREFUL! (( )) ÿ ( ) – the value of the point in the node and the parameter of the node are not the same!

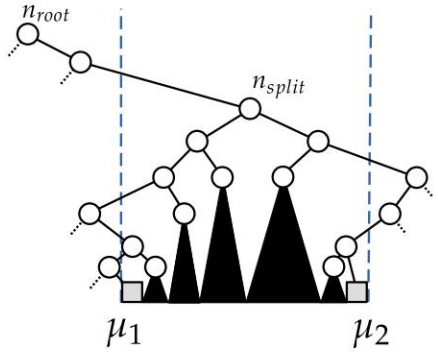
Priority tree



function CREATEPRIORITYTREE(P)
  $p_{min} = \underset{p_i \in P}{arg\ min}\ x(p_i)$
  $n \leftarrow$ new node
  $p(n) \leftarrow p_{min}$
  if $P \setminus p_{min} \neq \emptyset$ **then**
    $y_{med} \leftarrow median(P \setminus p_{min})$
    $P_L \leftarrow \{p : p \in P \setminus p_{min}, y(p) \leq y_{med}\}$
    $P_R \leftarrow \{p : p \in P \setminus p_{min}, y(p) > y_{med}\}$
    $y(n) \leftarrow y_{med}$
    if $P_L \neq \emptyset$ **then**
      $leftChild(n) \leftarrow$ CREATEPRIORITYTREE(P_L)
    if $P_R \neq \emptyset$ **then**
      $rightChild(n) \leftarrow$ CREATEPRIORITYTREE(P_R)
  return n

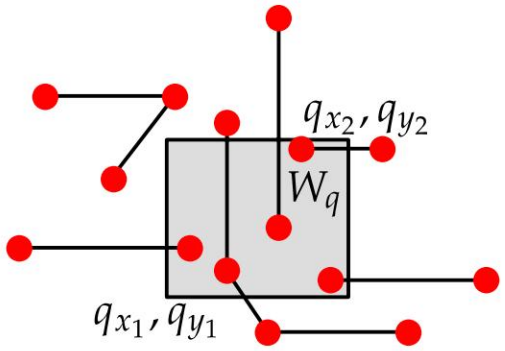


Priority tree



- Given that the value of the node point (( )) of the tree is not the same as the node parameter (( )) – the node parameter is used in forming a binary tree
  - (( )) is used when checking whether a specific point is in the query window, that is, inside [&!, &"]
  - ( ) is used to navigate the priority tree
- Let's look for the separation node :+%2; – the first one below the root that intervals 4. is within ÿ (:+%2;) 4- ÿ • We separate the search into two vertical paths
  - The concept is the same as range searches
- All the time we observe that (( )) ÿ - the principle of the heap 3-
- For the nodes of all paths that we pass from the root, including the nodes of all subtrees that we automatically add, we check each point of the node to see if it is within the query window(ÿÿ, 3-]× [4., 4-]

Searching for line segments



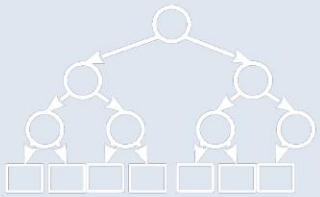
- Let's imagine a two-dimensional example where instead of intervals we have line segments

$$= \{ , = (\quad ,!, ,! \quad)(\quad ,", ," \quad) : (\quad ,/, ,/ \quad) \text{ ÿ ÿ\#} \}$$

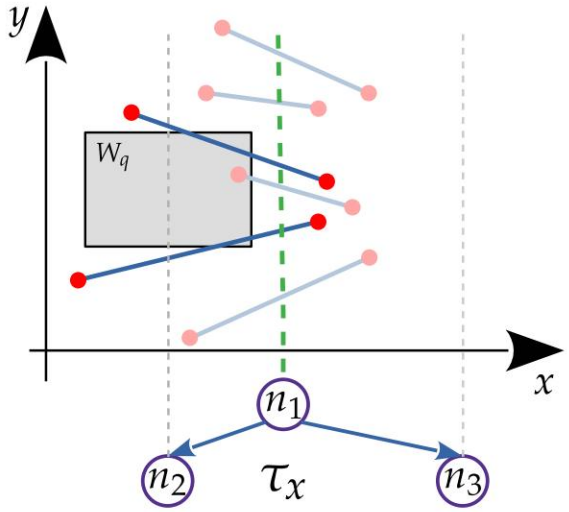
- Then we define the search area

$$= [\quad 3!, 3" \quad] \times [\quad 4!, 4" \quad]$$

  - An example of the printed circuit board we want find all lines that pass through a certain area • An example of a city plan where you want to find streets that are within a search area

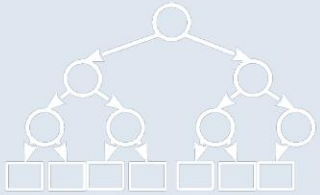


Searching for line segments

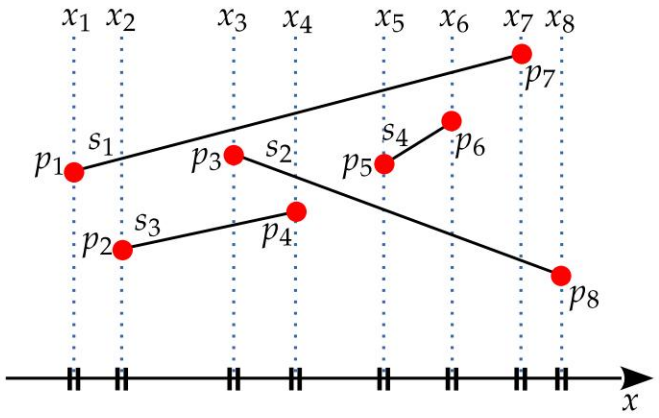


- Can we use a tree for such a search? interval?
- Let's imagine that we have a query window that is to the left of the median node (" in the example).
  - We check whether the left ends of the line segments of the node are
    - within ÿÿ,
  - For example, we see at least two line segments with their left ends not in the required area, but they still intersect the query window

- Although an interval tree will work correctly for most cases, some will still produce a *false negative* result



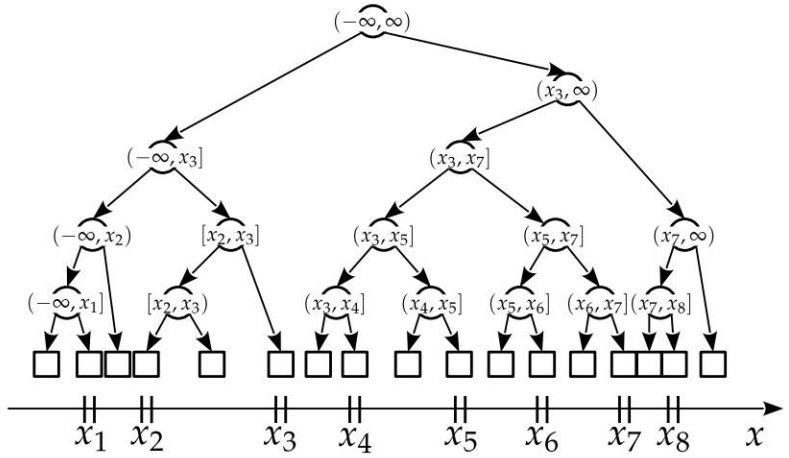
Searching for line segments



- Instead, let's apply the positional approach *(locus approach)*
- We start from the -axis
  - We break the set of line segments into elementary ones intervals - these are intervals along the -axis in which there are no changes in the number of line segments
  - Changes occur at the end points of the linear ones of segments – we use end point projections on - axis : $_{\&}$ = ($_{\&}$)
  - We break the example in the picture into the following elementary intervals

(ÿÿ, ! ,) [!, ! ,] (!, " ,) [", " , ...] (2, 3 , 3, 3 , (3, ÿ]

Searching for line segments



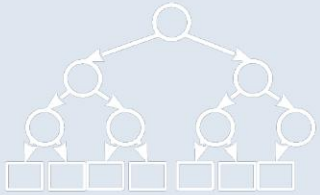
(ÿÿ, ! ,) [!, ! ,] (!, " ,) [", " , ...] (2, 3 , 3, 3 , (3, ÿ]

- Let's turn each of the elementary intervals into one leaf of a balanced binary tree - the leaves are values (concept of B+ index tree), and the internal nodes are just for reference

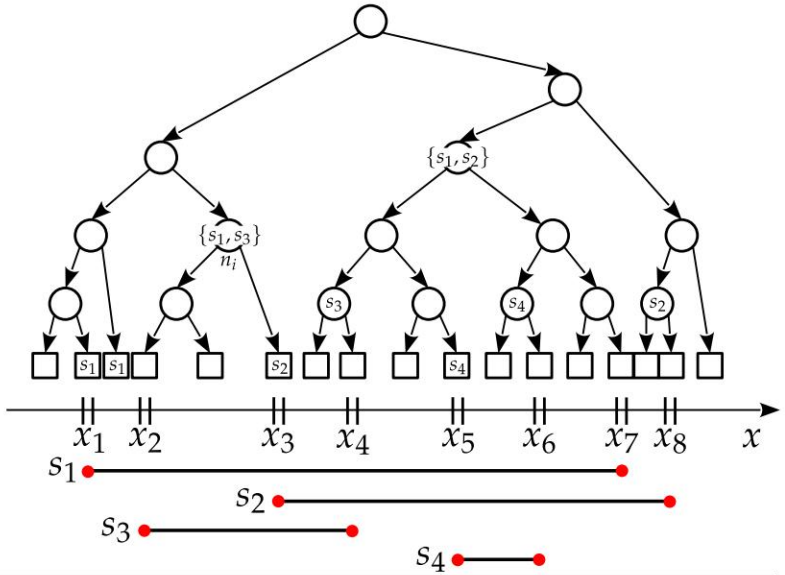
- We make the tree above those leaves in a thoughtful way
  - We want the tree to be balanced so that the search is + log/ - ( ) again coming back to the concept of searching for span

- The internal nodes of the tree aggregate the intervals of their subtrees = (4) ÿ (5)
  - This results in intervals in nodes of the same level not overlapping and having no gaps

- The root node aggregates all elementary intervals, which results in ÿÿ, ÿ( )



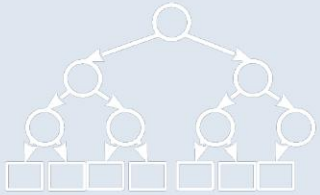
Searching for line segments



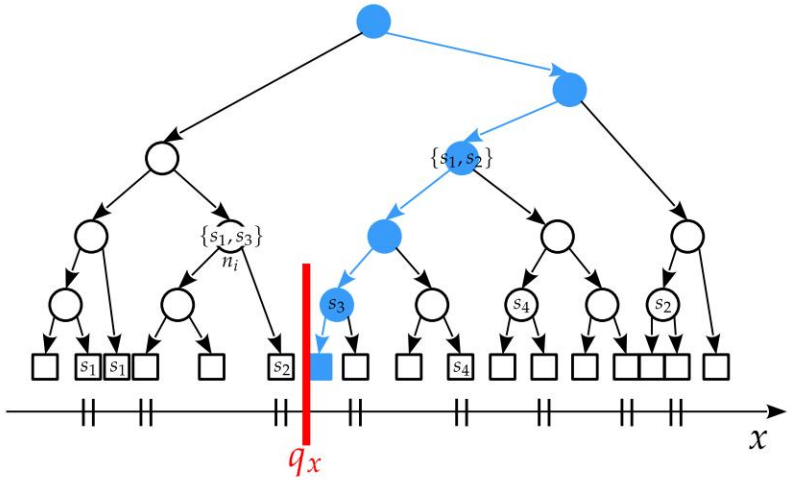

```
procedure InsertSegment(n, [x1, x2])
    if I(n) ⊆ [x1, x2] then
        add [x1, x2] to S(n)
    else
        if I(leftChild(n)) ∩ [x1, x2] ≠ ∅ then
            InsertSegment(leftChild(n), [x1, x2])
        if I(rightChild(n)) ∩ [x1, x2] ≠ ∅ then
            InsertSegment(rightChild(n), [x1, x2])
```

- Then we assign line segments to nodes trees
  - Logic dictates that each line segment in assign your sheet to the elementary interval
  - It is possible (and probable) that several leaves contain the same line segment & - thus we cause high complexity of saving the tree *(storage complexity)*
  - We assign such a line segment to an internal node that contains all leaves that have that line segment &
  - Using node intervals for such allocation we can start from the root node
    &, interval is • For contained while is completely an
  - represents a canonical set of line segments that completely pass through the

interval • We obtain **a tree of segments**



Searching for line segments

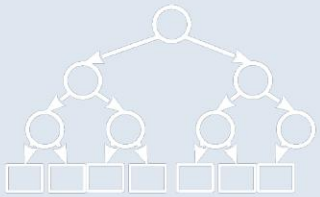


- If we want to find line segments through which they pass
through < = 3× 4.[, 4-]
- We need to find the elementary interval through which it passes
3
- We perform a classic traversal of the tree, passing through the nodes where ÿ ( )
3
- The sheet represents the requested elementary interval
- We denote the vertical path from the root node to the leaf as • The set of all line segments that

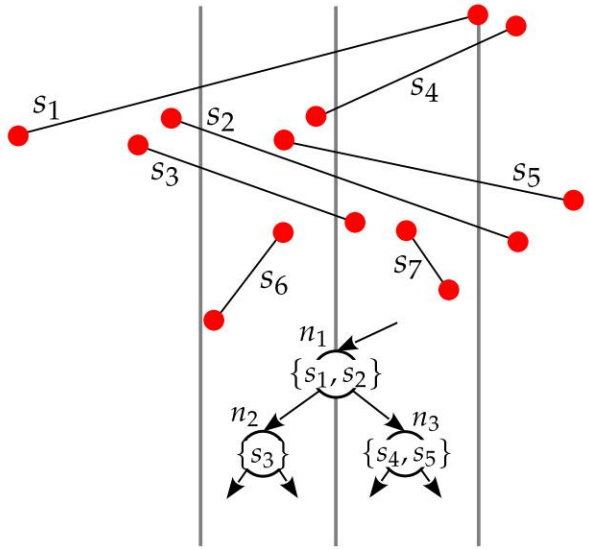
pass through 3 are

E ( ) ÿ
=ÿ

- In our example !, ",{@ }



Searching for line segments



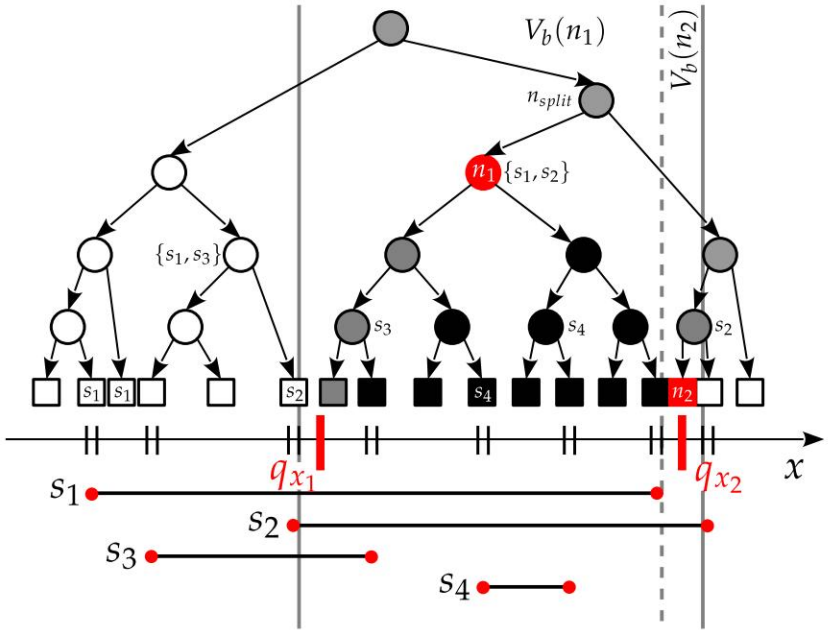
- If we take the interval of a node ( ) on = ( )×(ÿÿ, defines a vertical block & ( ) ÿ)
- All line segments in ( ) pass completely horizontally through the entire & • (In) the example in the picture, we

notice another property that is related to the principle of interval aggregation

- AND !) = AND ") ÿ A @( )
- Line segments thus (! completely pass through and A! ,) and through • Line ") - AND @)
  segments that completely pass through A (,) 2

  where 2 is in the subtree whose root is B, **they partially pass** through AB ( )

Searching for line segments



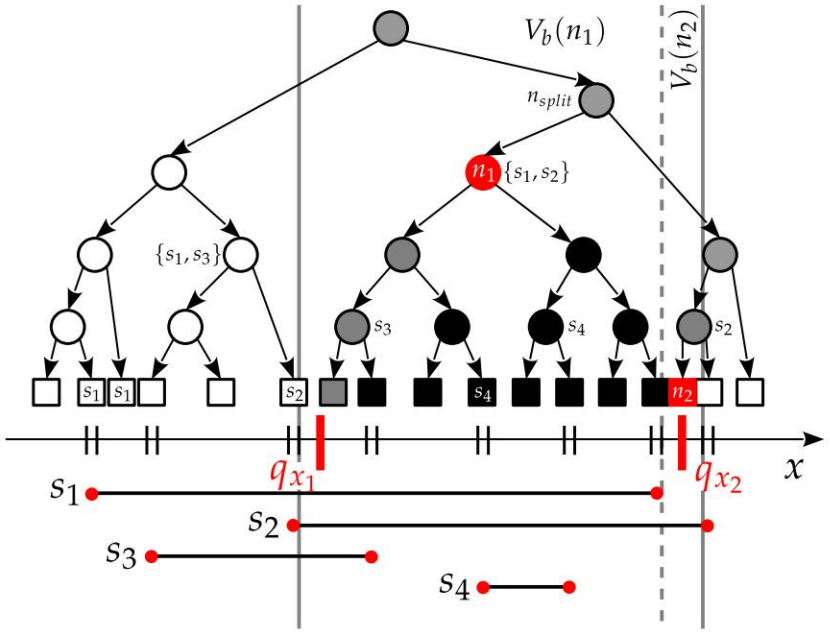
- Let's go back to window search < [10, 1/] = [00, 0/] ×
- The concept is the same as the range search
- We find the splitting node &!'" which is in this case:
  - Contains the entire $6 interval = [1#, 1"]
  - Left child 4 contains • 1# ÿ (4)
  - Right child 5 contains • After 1" ÿ (5)

that we get:

- Two vertical paths, the left 4 and the right 5 - in example gray nodes
- The set of subtrees 7 that are surely in the interval $6 - in for example nodes of black color

- The result is a set of vertical blocks

Searching for line segments

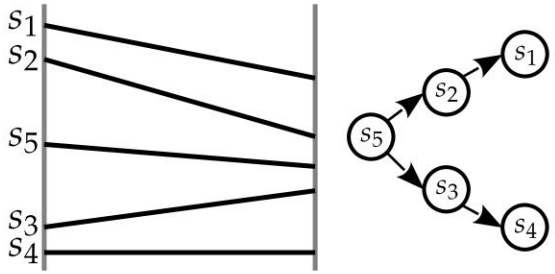


- In a set of vertical blocks that are the result of:
  - We can only have one vertical block
    - When 01#(& is in the elementary interval – in the sheet
    - When 01#(& is the root of a subtree, all of which are elementary intervals in 2#
  - The first and last vertical blocks contain 1# and 1"
- A set of line segments that completely or partially intersects sets of line segments of <8 is the segment tree canonical subtrees that form a set of vertical blocks - see red nodes

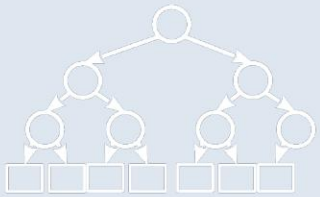
E ( ) ÿ

>ÿ 9ÿ :ÿB;

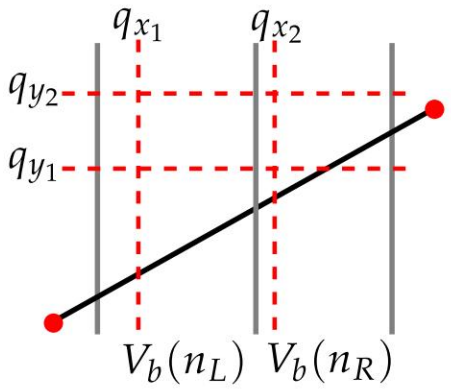
Searching for line segments



- What about vertical search, how we find
  = [ ] ?
  << 10, 1/
- So far we know that the canonical set of line segments ( ) contains line segments that completely horizontally intersect the vertical block C( )
- If we organize the canonical set of segments in balanced binary tree, we can use the same principle of geometric search as with ranges
  - Problem: In order to clearly know the vertical arrangement of the lines segments, they **must not intersect !**
- We associate such a tree with a node that has ( ) ÿ ÿ and denote it with ( )



Searching for line segments



- The first and last vertical blocks contain 3" and 3"
- In these vertical blocks, we must not only look at the place where the line segment intersects the edges of the vertical block

- The intersection of the line segment with must be taken into account in order to avoid *false positive* addition that is, of the line segment to the final result

3", 3!,

Searching for line segments

```
function ReportSubtree(n, W_q = [q_x1, q_x2] × [q_y1, q_y2])
    rv ← ∅
    if n is not nil then
        add VerticalQuery(τ(n), n, W_q) to rv
        add ReportSubtree(leftChild(n), W_q) to rv
        add ReportSubtree(rightChild(n), W_q) to rv
    return rv
function FindSplittingNode(τ, I_q = [q_x1, q_x2])
    n ← root(τ)
    while n is not leaf do
        lc ← leftChild(n), rc ← rightChild(n)
        if q_x1 ∈ I(lc) and q_x2 ∈ I(rc) then
            return n
        else
            if I_q ⊆ I(lc) then
                n ← lc
            else
                n ← rc
    return n
```

```
function SegmentTreeQuery(τ, W_q = [q_x1, q_x2] × [q_y1, q_y2])
    rv ← ∅
    n_split ← FindSplittingNode(τ, [q_x1, q_x2])
    if n_split is leaf then
        if S(n_split) ≠ ∅ then
            add VerticalQuery(τ(n_split), n_split, W_q) to rv
    else
        n ← leftChild(n_split)
        while n is not leaf do
            add VerticalQuery(τ(n), n, W_q) to rv
            lc ← leftChild(n), rc ← rightChild(n)
            if q_x1 ∈ I(lc) then
                add ReportSubtree(rc, W_q) to rv
                n ← lc
            else
                n ← rc
        add VerticalQuery(τ(n), n, W_q) to rv
        n ← rightChild(n_split)
        while n is not leaf do
            add VerticalQuery(τ(n), n, W_q) to rv
            lc ← leftChild(n), rc ← rightChild(n)
            if q_x2 ∈ I(rc) then
                add ReportSubtree(lc, W_q) to rv
                n ← rc
            else
                n ← lc
        add VerticalQuery(τ(n), n, W_q) to rv
    return rv
```

Questions