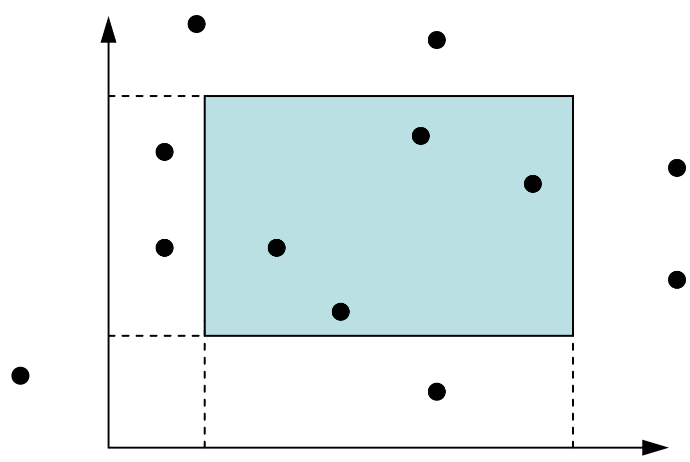
Busca em Regiões Ortogonais

Claudio Esperança
Paulo Roma

O problema

- O problema consiste em recuperar objetos – tipicamente pontos – que intersectam ou estão contidos numa região "simples" do espaço
- Freqüentemente a região é ortogonal (*orthogonal range*), isto é, uma região em espaço *d*-dimensional delimitada por hiperplanos perpendiculares aos eixos

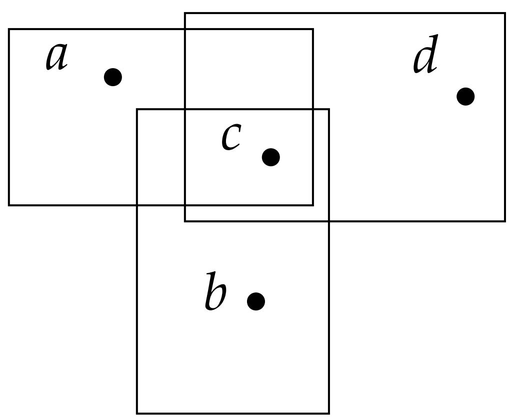


Estrutura de Dados

- Interessa-nos resolver o problema várias vezes para regiões diferentes
  - Pontos (ou outros objetos) são armazenados numa estrutura de dados
  - Tempo de pré-processamento
    - Tipicamente linear
  - Complexidade de espaço
    - Linear ou quase linear
- Várias estruturas de dados usadas na prática
  - Em memória: quadtrees, *k-d*-trees, *BSP*-trees
  - Em disco: quadtrees, *r*-trees, grid files

Coerência Espacial

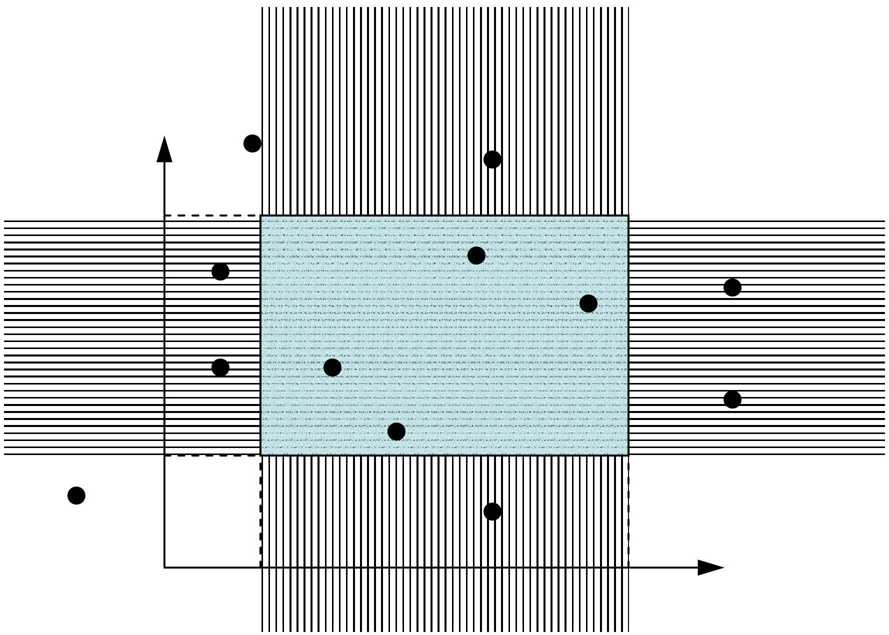
- Dado um conjunto de pontos *P* e uma região *R*, qual o espaço de todos os possíveis resultados?
  - A princípio, espaço é o conjunto de todos os possíveis subconjuntos de *P*
  - Na verdade, nem todos os subconjuntos são possíveis se *R* é uma região simples



O subconjunto { a, b, d } não é possível se R é um retângulo

Regiões retangulares

- Quando a região é um retângulo, pode-se decompor a consulta em *d* consultas unidimensionais



Consultas em 1D

- Dado um conjunto de pontos $P = \{ p_1, p_2, \ldots p_n\}$ sobre uma reta e um intervalo $R = [x_{\min}, x_{\max}]$, reportar todos os pontos p_i tais que $p_i \in R$
  - Busca binária claramente resolve o problema em $O(\log n + k)$, onde k é o número de pontos que satisfazem a consulta
  - Um problema análogo é o de *contar* os pontos que satisfazem a consulta, i.e., obter k
    - Busca binária resolve o problema em $O(\log n)$
- Busca binária, entretanto, tem os seguintes inconvenientes:
  - Não generaliza para $d > 1$
  - Requer um *array* $\rightarrow$ inserção de novos pontos em $O(n)$

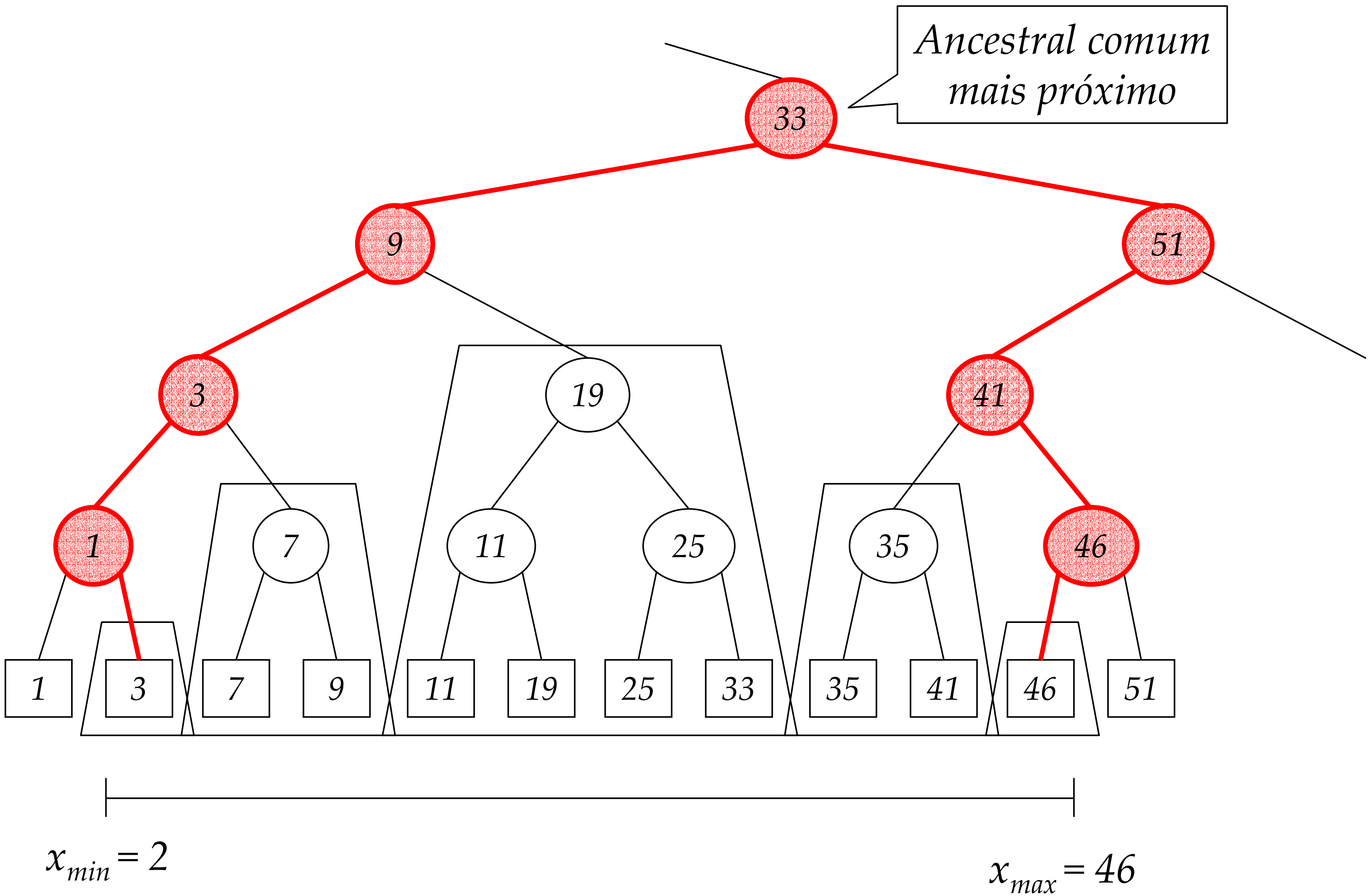
Subconjuntos canônicos

- Uma abordagem comum para o problema de busca é agrupar os pontos de P em sub-conjuntos "canônicos" $\{S_1, S_2, \dots S_k\}$ tais que $\bigcup S_i = P$
  - k guarda alguma relação com n que permite que qualquer consulta resulte em um pequeno número de subconjuntos – tipicamente $O(\log n)$
  - Não necessariamente os subconjuntos canônicos de P são disjuntos, mas a resposta é computada como uma coleção de subconjuntos disjuntos

Árvore binária de busca balanceada

- Uma partição em subconjuntos canônicos possível para o problema em 1D é uma ABBB
  - Pontos guardados em folhas da árvore
  - Valores dos nós internos podem ser arbitrários desde que não violem as propriedades de uma ABB
    - Ex. : maior dos descendentes à esquerda ou menor dos descendentes à direita
  - Cada subconjunto canônico corresponde aos descendentes de uma sub-árvore

Árvore binária de busca balanceada



Consulta em 1d usando ABB

- Encontram-se os caminhos para o menor e para o maior elemento do intervalo requisitado
- Encontra-se o ancestral comum mais próximo
- Fazem parte da resposta
  - Todas as sub-árvores à <u>direita</u> dos nós internos no caminho até o <u>menor</u> elemento
  - Todas as sub-árvores à <u>esquerda</u> dos nós internos no caminho até o <u>maior</u> elemento
- Como a árvore é balanceada e os caminhos são logarítmicos
  - Encontrar as sub-árvores é $O(\log n)$
  - Reportar todos os pontos é $O(\log n + k)$

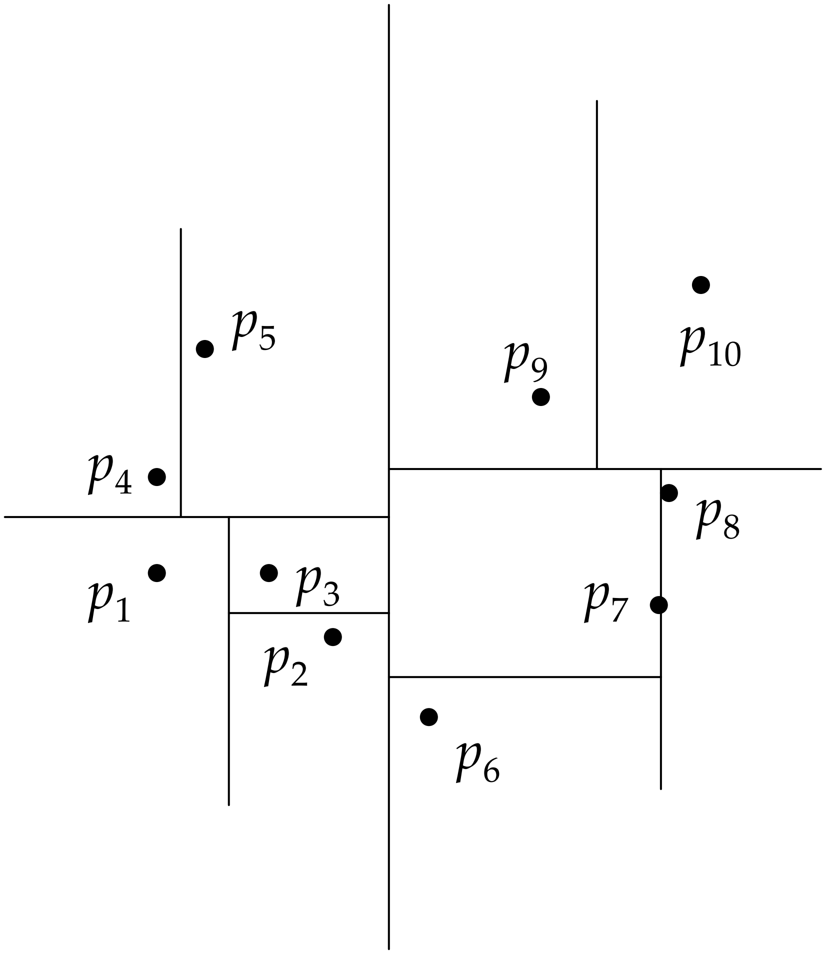
k-d-trees

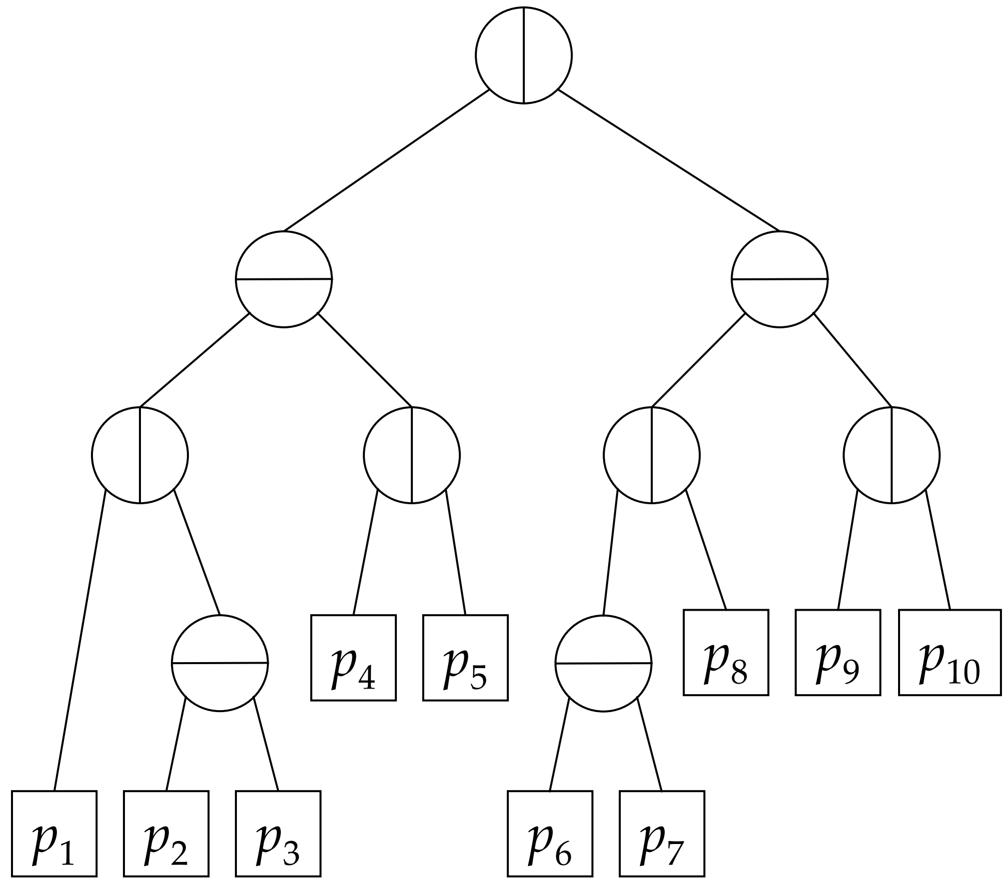
- Estrutura de dados proposta por Jon Bentley que estende a ABB para *k* dimensões
- Idéia é dividir o espaço por hiper-planos perpendiculares aos eixos coordenados alternando entre os eixos *x*, *y*, *z*, etc
  - Em algumas variantes, a escolha do eixo pode ser baseada em algum critério geométrico
    - Ex.: Maior dimensão do hiper-retângulo que contém todos os pontos da partição

k-d-trees

- Hiper-planos correspondem aos nós internos
  - Guarda-se qual a coordenada escolhida (*x*, *y*, *z*, etc) e o valor
    - Ex.: $x = 5$ indica que os nós à esquerda têm $x \leq 5$ e os à direita têm $x > 5$
- Nós-folha são hiper-retângulos, possivelmente ilimitados que contêm os pontos
  - Tipicamente um por folha, mas pode também ser um número constante máximo
    - Idéia de *bucket* → útil para armazenamento em disco

k-d-trees





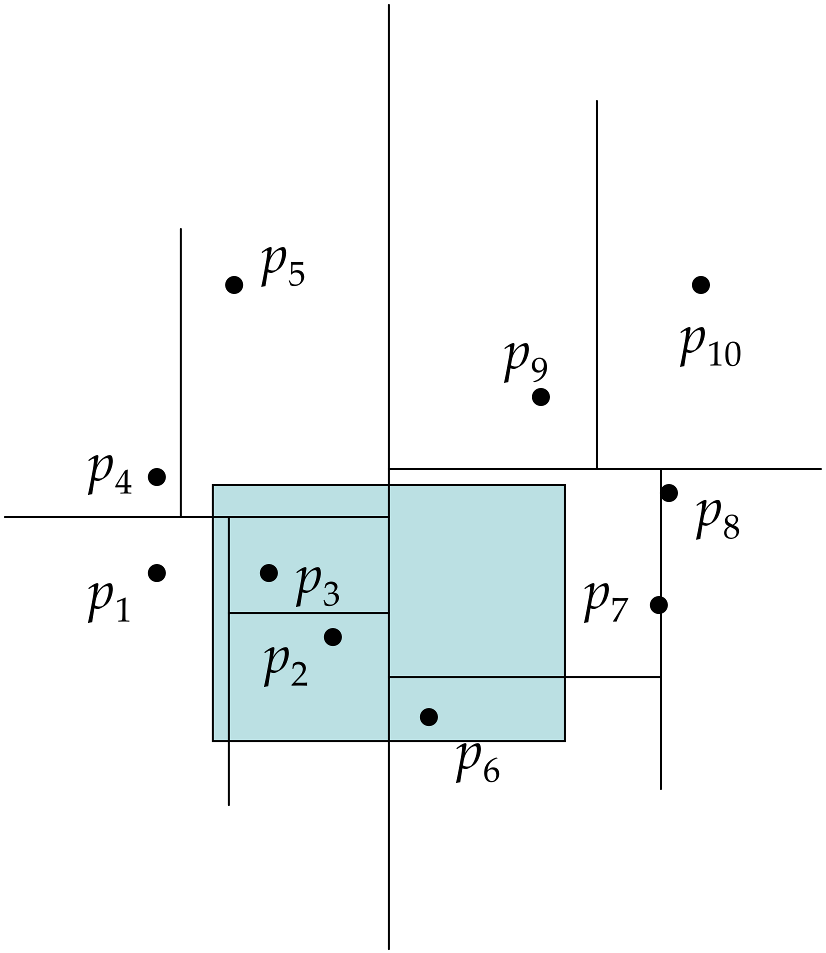
BSP-trees

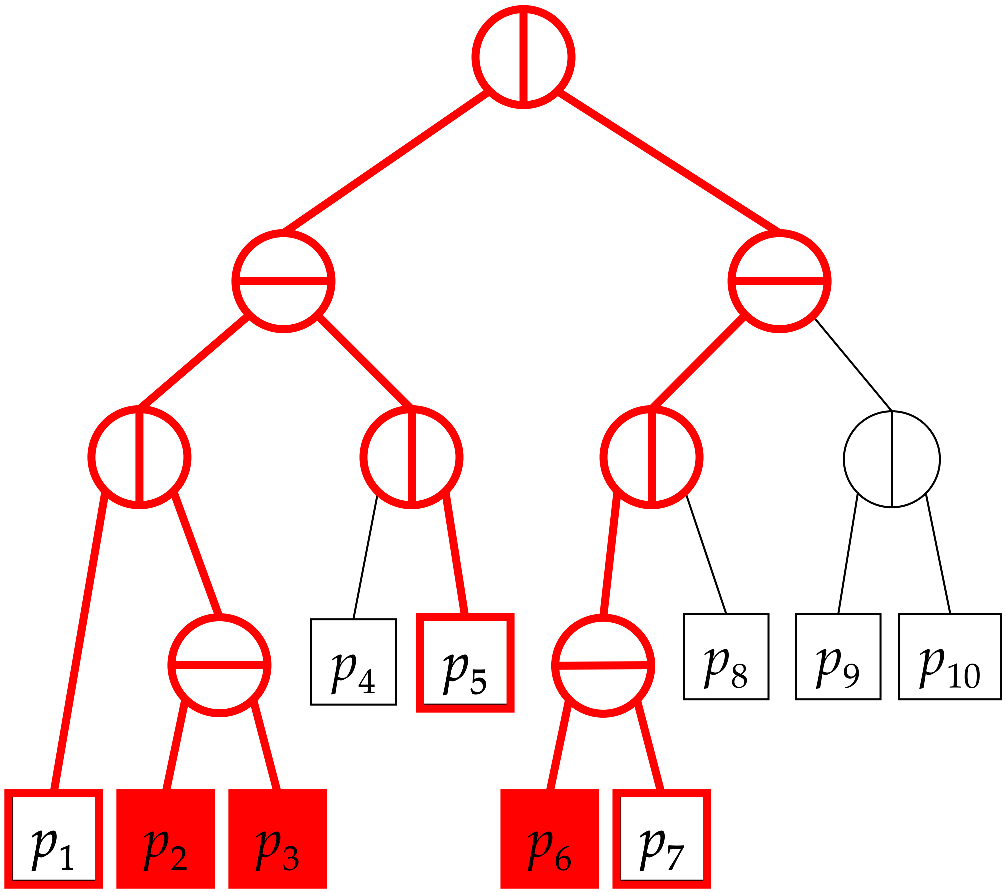
- *BSP-trees* (*Binary Space Partition Tree*) são mais gerais que *k-d-trees*
  - É relaxada a necessidade de se usar planos ortogonais
  - Células (nós-folha) são polígonos convexos
- Como escolher o valor da coordenada em cada nó interno?
  - O mais comum é escolher a mediana com relação ao eixo de partição
    - Medianas podem ser computadas em $O(n)$
    - Garantidamente cada partição resultante contém ~ $n/2$ pontos
    - Construção é feita em $O(n \log n)$

Consulta em *k-d-trees*

- Dado um retângulo *R*, visita-se a raiz *T*
  - Se a região correspondente a *T* está contida em *R*, reportar *T* e terminar
  - Se *T* é um nó folha, reportar os pontos de *T* que satisfazem a consulta e terminar
  - Se a partição da esquerda de *T* intersecta *R*,
    - Visite o filho esquerdo de *T*
  - Se a partição da direita de *T* intersecta *R*,
    - Visite o filho direito de *T*

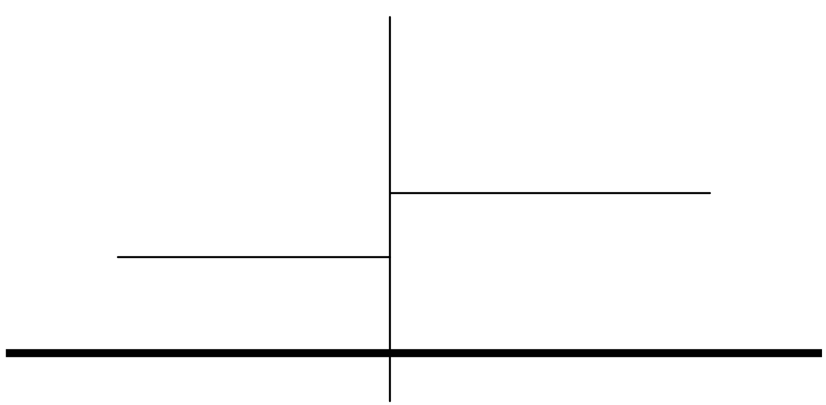
Consulta *k-d-trees*

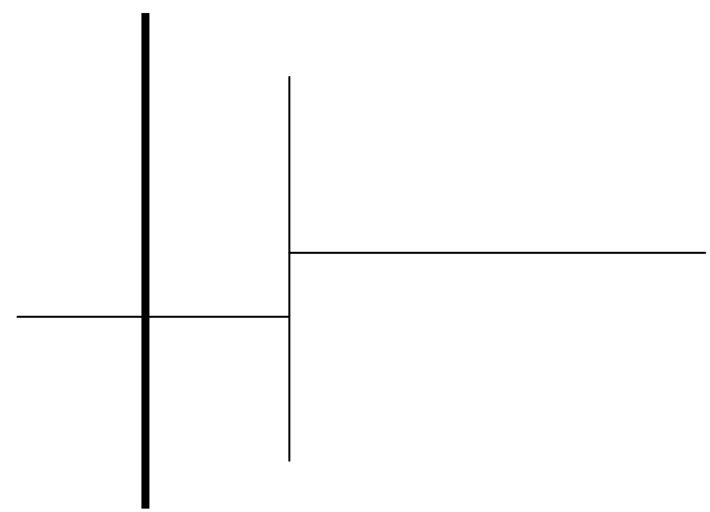




Complexidade da consulta em *k-d-trees*

- Quantos nós o algoritmo visita?
  - O algoritmo visita recursivamente uma partição se algum dos lados de *R* intersecta a partição
  - Considere os 4 *netos* de uma sub-árvore *T* e um dos lados de *R*. No máximo <u>2 netos</u> podem ser intersectados





Complexidade da consulta em *k-d-trees*

- Seja $Q(n)$ o número de nós visitados de uma k-d-tree T com n pontos guardados em seus nós folha
- Cada lado do retângulo intersecta no máximo 2 netos de T, cada qual com ~ $n/4$ pontos em seus nós descendentes
- Logo, a relação de recorrência é

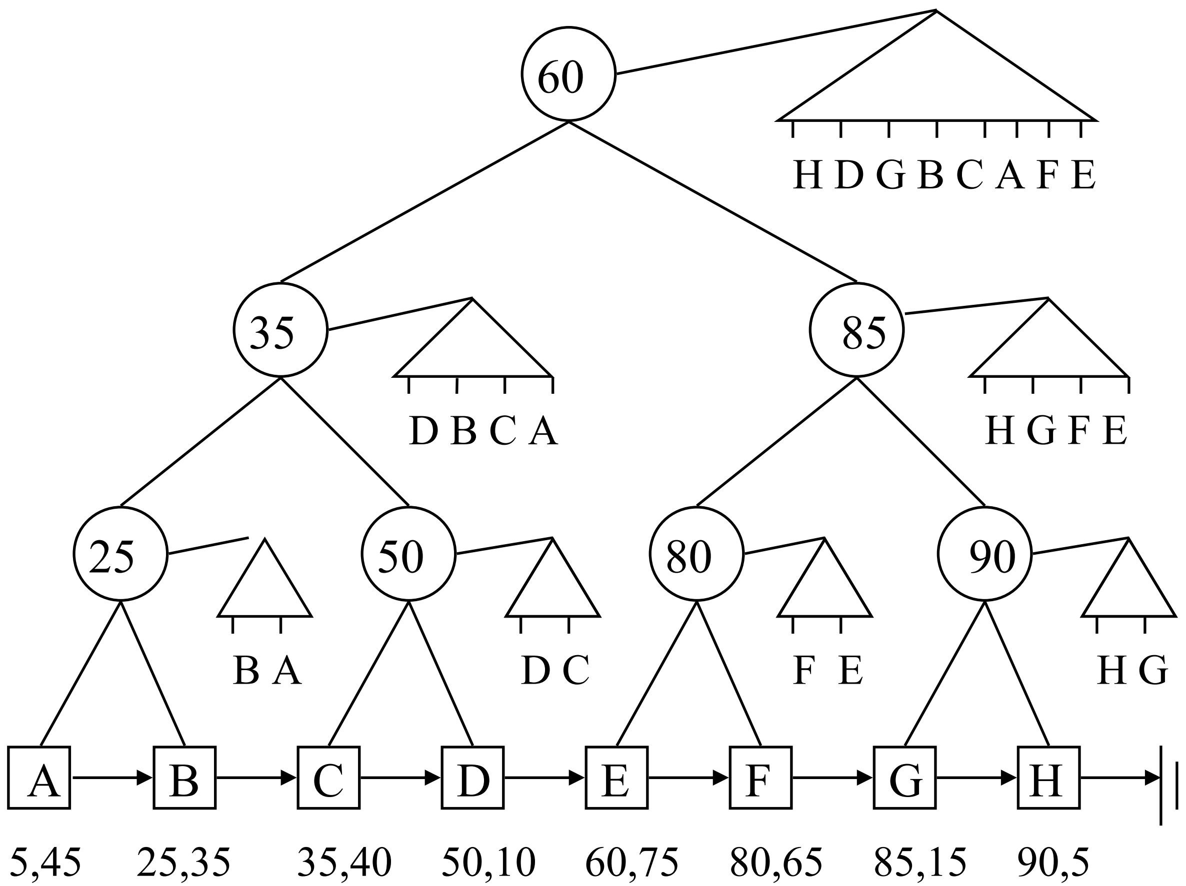
$$Q(n) = \begin{cases} 1 & \text{para } n = 1 \\ 2Q(n/4) & \text{para } n > 1 \end{cases}$$

- Usando, por exemplo, o teorema mestre, esta recorrência resolve-se em $O(n^{\log_4 2}) = O(\sqrt{n})$
- Portanto, o problema de *contar* os pontos que satisfazem a consulta (2D) tem complexidade $O(\sqrt{n})$ e problema de *reportar todos* os pontos tem complexidade $O(\sqrt{n} + k)$, onde k é o tamanho do resultado

Orthogonal Range Trees

- *k-d-trees* têm a vantagem de serem estruturas dinâmicas mas o tempo de busca de regiões ortogonais é ruim - $O(\sqrt{n} + k)$
- Árvores de regiões ortogonais suportam busca de regiões retangulares(2D) em $O(\log^2 n + k)$
  - É uma árvore de busca multi-nível
  - Usa-se uma árvore binária de busca balanceada para indexar a coordenada x
  - Os pontos descendentes de cada sub-árvore são re-indexados usando uma árvore binária de busca ordenada cujas chaves são as coordenadas y

Orthogonal Range Trees



Complexidade de tempo

- Os subconjuntos canônicos relativos à coordenada x podem ser obtidos em $O(\log n)$
  - O número de subconjuntos (sub-árvores) é $O(\log n)$
  - Cada subconjunto corresponde a uma sub-árvore para a coordenada y que é pesquisada a um custo $O(\log n)$
  - Portanto, o custo total é $O(\log^2 n)$
  - Para enumerar todos os k pontos que satisfazem a consulta gasta-se $O(\log^2 n + k)$

Complexidade de Espaço

- A árvore para a coordenada x ocupa espaço $O(n)$, uma vez que é uma árvore binária comum
- Levando em conta todas as sub-árvores auxiliares para a coordenada y, a estrutura ocupa espaço $O(n \log n)$
  - Cada ponto aparece nas sub-árvores auxiliares (y) de todos os seus nós (x) ancestrais
  - Como a árvore é balanceada, cada folha tem $\log n$ nós ancestrais
- Em d dimensões, a *range tree* ocupa espaço $O(n \log^{d-1} n)$

Construção da *range tree*

- Observe que as árvores auxiliares (y) podem ser armazenadas mais simplesmente como *arrays* ordenados
- Construir a árvore 1-D para a coordenada x leva tempo $O(n \log n)$
- Se construirmos os *arrays* y da maneira *top-down* habitual, a construção da *range tree* levará $O(n \log^2 n)$
- Evita-se isso construindo os *arrays* y de maneira *bottom-up*
  - Os *arrays* correspondentes aos nós folhas são trivialmente construídos
  - Para construir o *array* do nó imediatamente acima, simplesmente intercala-se os dos dois nós filhos a um custo $O(n)$

Range trees multidimensionais

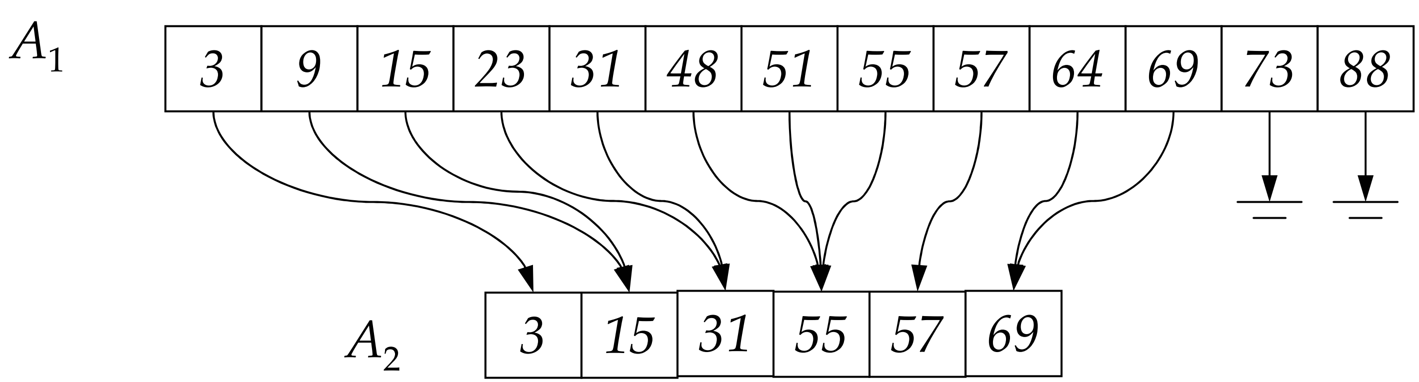
- Resumindo, a idéia é construir a *range tree* de forma recursiva
  - Constrói-se uma árvore para a primeira dimensão
  - Associada a cada sub-conjunto canônico é construída uma *range tree* de dimensão d-1
    - Se é a última dimensão, pode-se construir um *array* ordenado em vez de uma árvore
  - O tempo de busca dos subconjuntos canônicos será $O(\log^d n)$ enquanto que a enumeração de todos os pontos leva $O(\log^d n + k)$

Fractional Cascading

- O tempo de busca em *range trees* 2D pode ser melhorado para O (log n) através de uma técnica chamada *fractional cascading* (cascateamento fracionário?)
- Cada vez que descemos na árvore x e descobrimos um subconjunto canônico temos que buscar os pontos que satisfazem o intervalo y na sub-árvore y associada
  - Observe que sempre fazemos uma busca por y_{min} e y_{max}
  - A idéia do *fractional cascading* é evitar repetir a pesquisa em y várias vezes
  - A busca é feita apenas em uma árvore y (a do ancestral comum mais próximo)
  - As demais árvores y são pesquisadas em $O(1)$

Fractional cascading

- Exemplo simples: queremos fazer uma busca de intervalo em 2 arrays ordenados A_1 e A_2
  - Tempo: O(log n_1+log n_2+k)
- Suponhamos que A_2 seja um subconjunto de A_1
  - Podemos melhorar o tempo criando um ponteiro entre cada elemento E de A_1 e o menor elemento de A_2 que é maior ou igual a E
  - Ao encontrar o menor valor que satisfaz o intervalo em A_1, basta seguir o ponteiro para obter o 1º elemento de A_2 que precisa ser testado
    - Tempo: O(log n_1+log n_2+k+1)



Layered range trees

- Podemos usar a idéia em *range trees* criando ponteiros entre os elementos do array ordenado y correspondente a um nó interno da range tree x e os elementos dos arrays ordenados dos filhos
  - Tempo : $O(2 \log n + k) = O(\log n + k)$