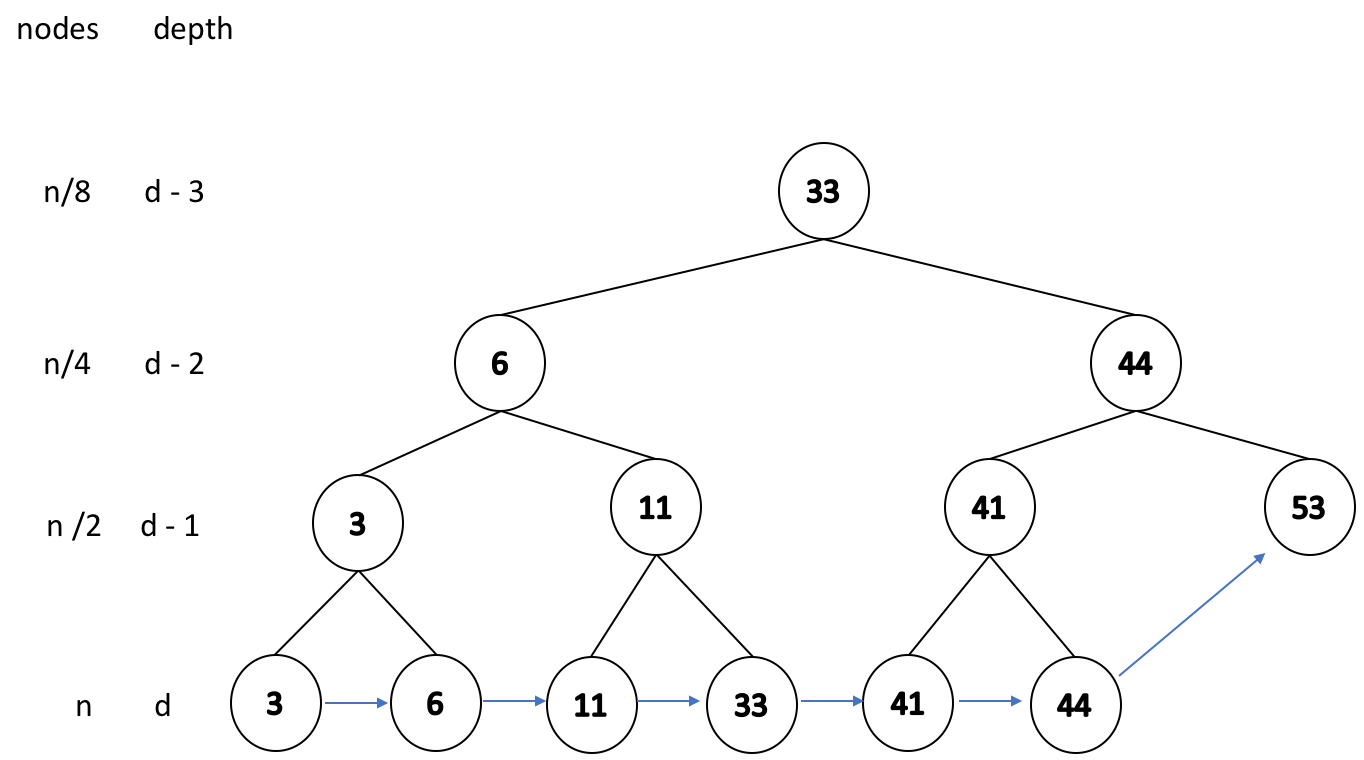
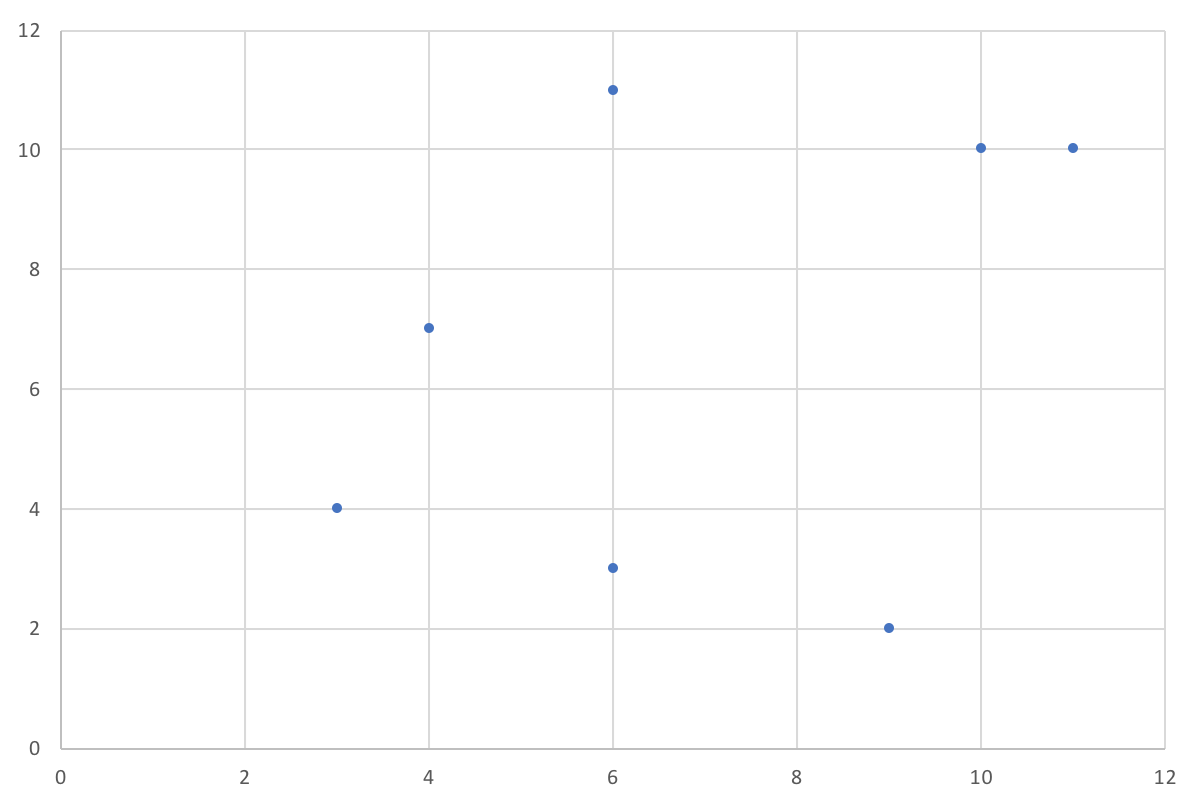
### **Range based search in a kD-Tree**

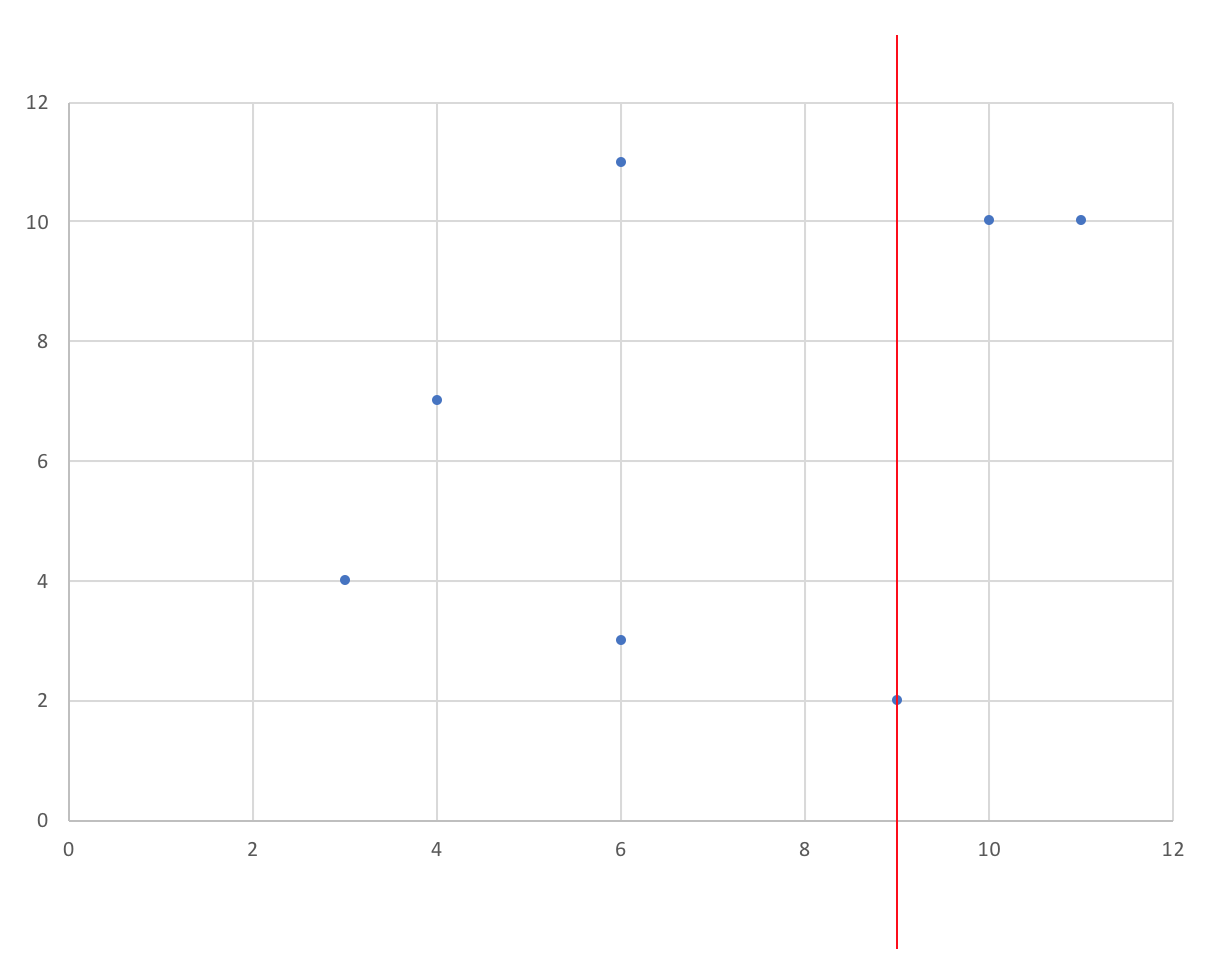
* + Given a set of points P = {p1,...,pn}, find the points in a range [a, b]
  + Build the kD-tree: use an AVL tree, but put all data in the leaves
  + Search is very similar with BST:
    - If dataToFind <= node.data, go left
    - If dataToFind > node.data, go right
  + If the exact data was not found (eg. find 42), we get **one past the data**, an upper bound
  + We want to find all numbers between [11,41] efficiently
    - Need to jump from 11 to 41 efficiently
    - So we make a linked list on data nodes
  + This is like running a **binary search** on a linked list
  + Runtime for finding all integers in a range [a, b]
    - find lower bound **a** in the tree: **O(lg n)**
    - find upper bound **b** in the tree: **O(lg n)**
    - traverse the linked list to find the elements in the middle: **O(k)**, where **k** is the number of elements in this range
    - So total is **O(lg(n) + k)**
  + Analysis
    - runtime depends on the situation: **k** or **lg(n)** might dominate the term
    - No better than running a binary search on an array, but no worse

### **Range search in 2-dimensions**

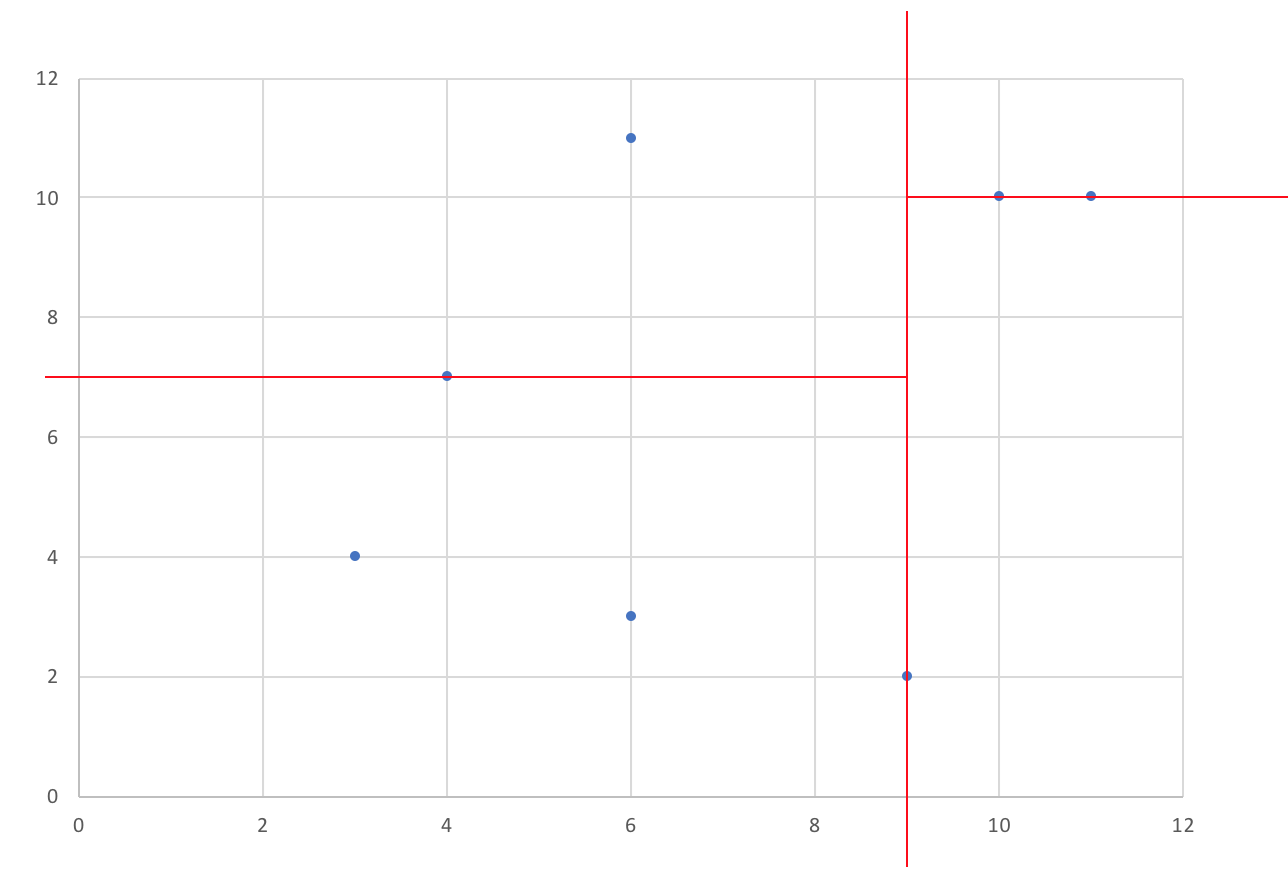
* + Given P = {p0, p1, …, pn}, where each pi is a point in 2-D space
    - find all the points in a rectangle {(x1, y1), (x2, y2)}
    - find the nearest point to (x, y)
  + Very widely asked questions in the real world. Eg: in image recognition
  + We still want to do a binary search, but a simple array will not suffice. We build a kD-tree.



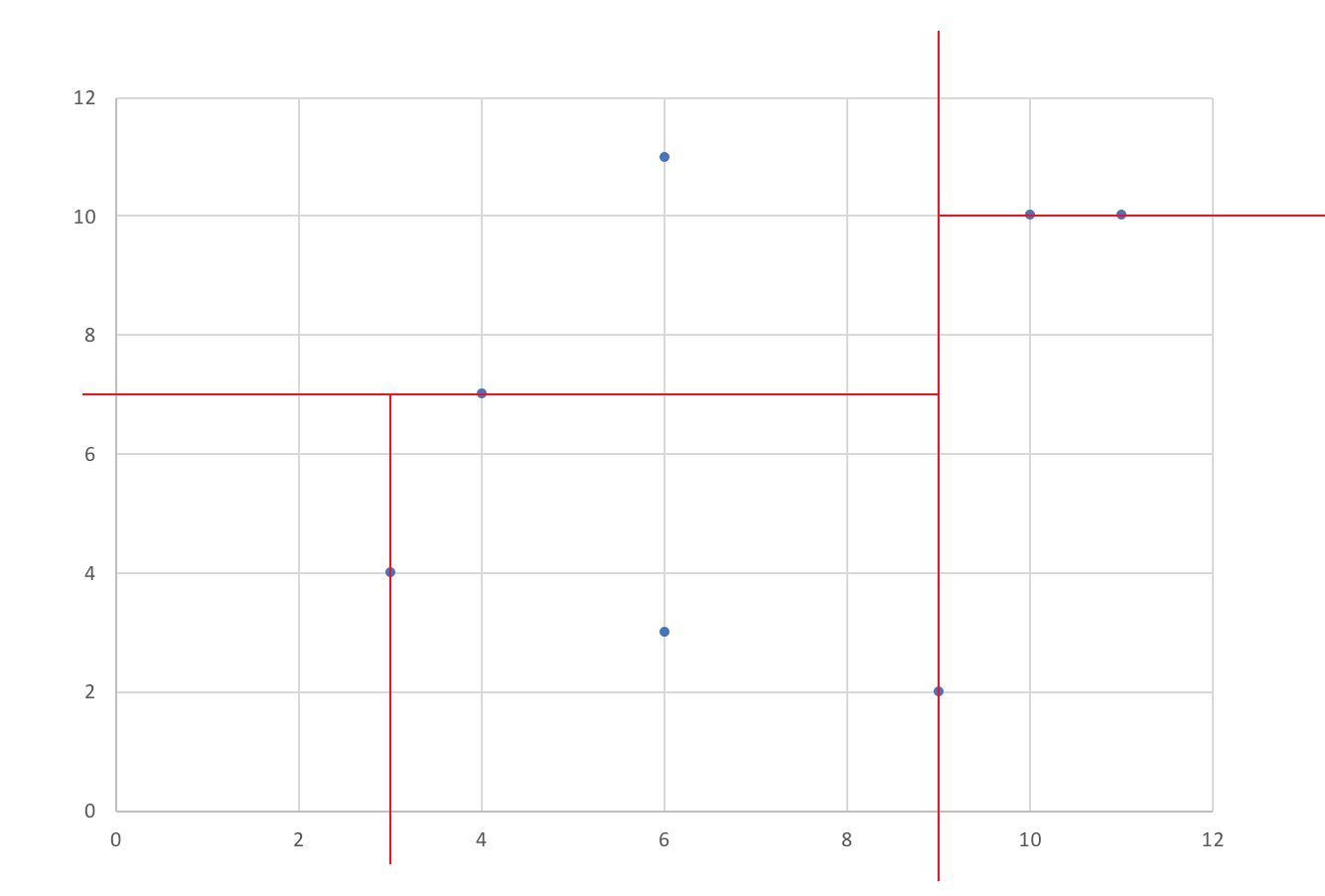
* + - Build a root node: find a proper **x** value to divide the space into 2 parts



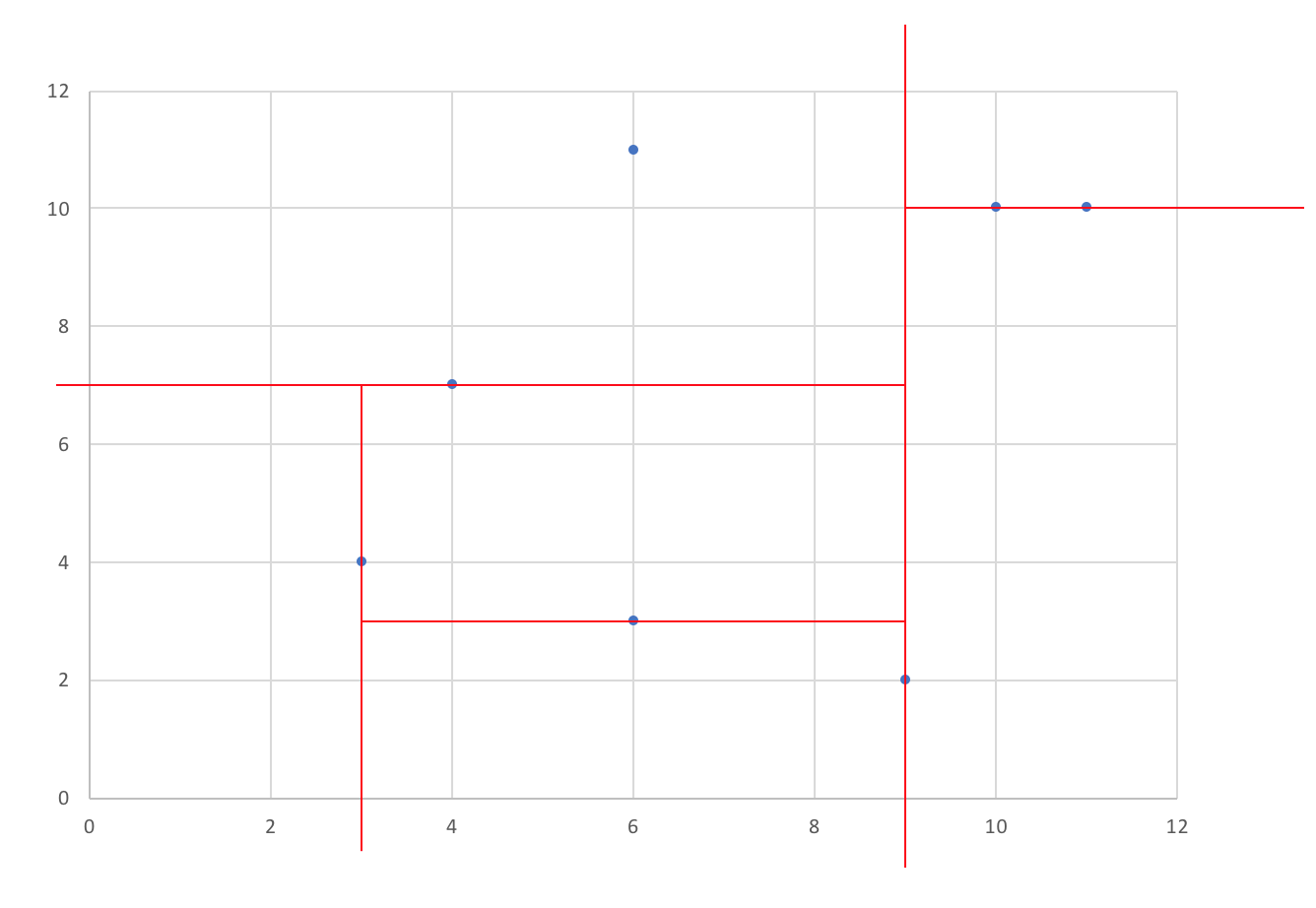
* + - Continue partitioning: find a proper **y** value for each subspace, and further divide each subspace into 2



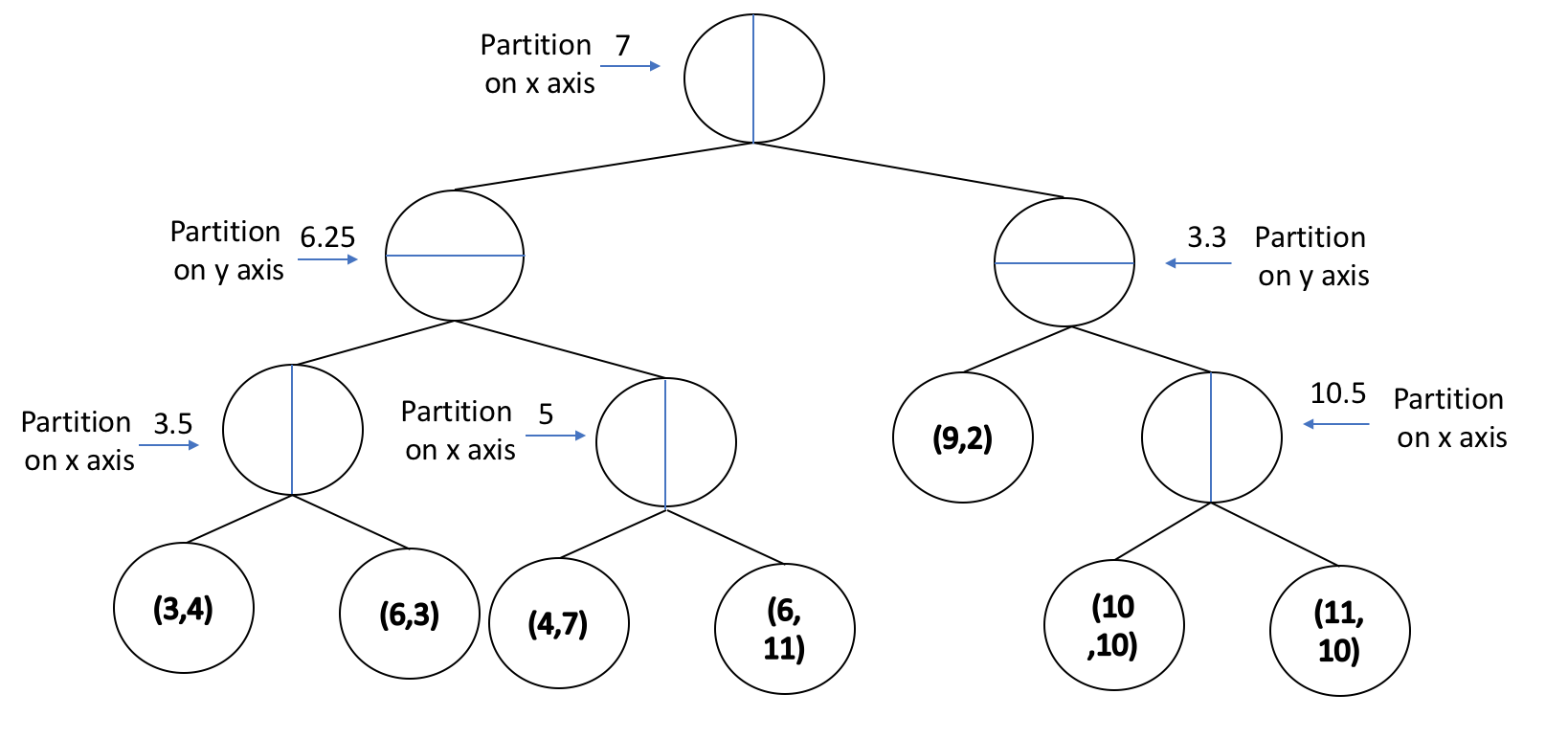
* + - Continue partitioning, and iterating through dimensions. So next we will partition each part depending on the **x** value (for 3-D spaces, we divide on **x, y, z** dimension)



* + - Stop until every part only has one point. That point will be the leaf data.



* + - The actual tree will be like this



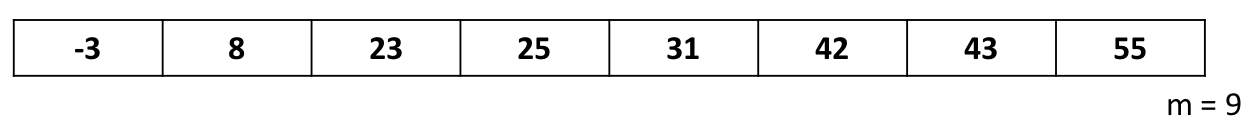
* + Search: similar to a binary search
    - Compare on the **x** value, then the **y** value, then the **x** value....
    - Alternating until we find the point we want
    - Each comparison eliminates half of the points in the search

#### **BTrees**

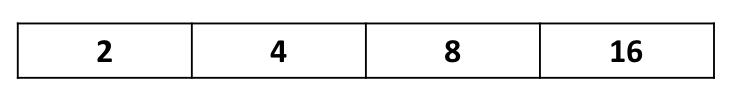
* + Motivation
    - We cannot always keep data in the memory
    - Where do we put data?
      * Disk
      * “Cloud” ¯\\_(ツ)\_/¯
      * Quantum Protons ( ͠° ͟ʖ ͡°)
    - Runtime
      * **3GHz** CPU performs **3 million** operations in **1ms**
      * Hard Drives
        + Bleeding-edge Disks can be very fast, ~32MB/ms
        + But most large disks are very slow, ~30 MB/s
      * The Cloud is slow
        + good ping time is 20-40 ms
        + takes ~40ms to going through an edge in a tree: very slow
        + want to lower the height even more!

#### **BTree (of order m)**

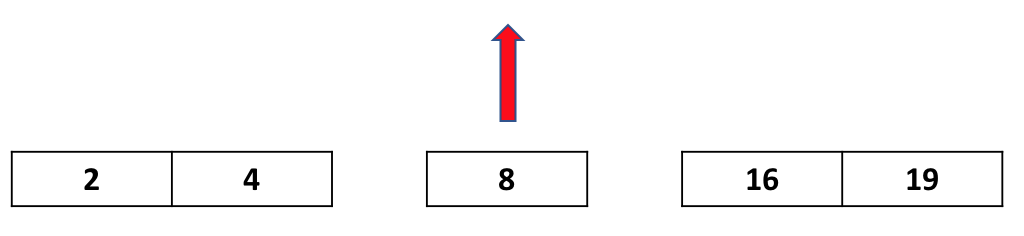
* + Please note that BTree is not a binary tree at all.
  + Goal: to minimize the number of reads.
  + In practice, every node will store exactly (1 network packet/1 disk block/...)
  + BTree node:
    - Every node is a sorted array
    - Contains up to m - 1 keys
    - For example: Btree of m = 9



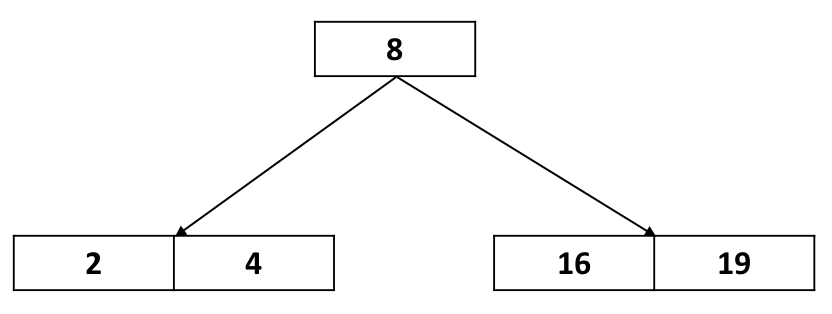
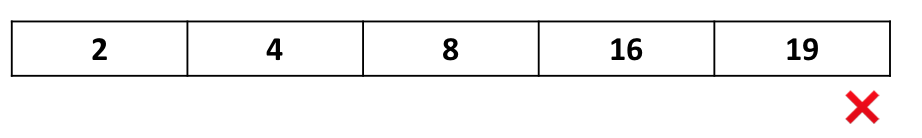
* + **Insertion (m=5)**
    - We add to the tree until we reach the maximum number of keys in a node.



* + - If we insert for example 19, we will exceed the allowed number of keys in this node. This is an overfilled array
    - we split the data and throw up the middle element.



* + - Now we have a BTree with order m=5



* + **Recursive call of split**
    - m = 3

