TREE DATA STRUCTURES

- Binary search tree
- K-D tree
- Quadtree
- R-tree

Author: prof. Yevhenii Borodavka

BINARY SEARCH TREE

In computer science, a binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. The time complexity of operations on the binary search tree is linear with respect to the height of the tree.

A binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

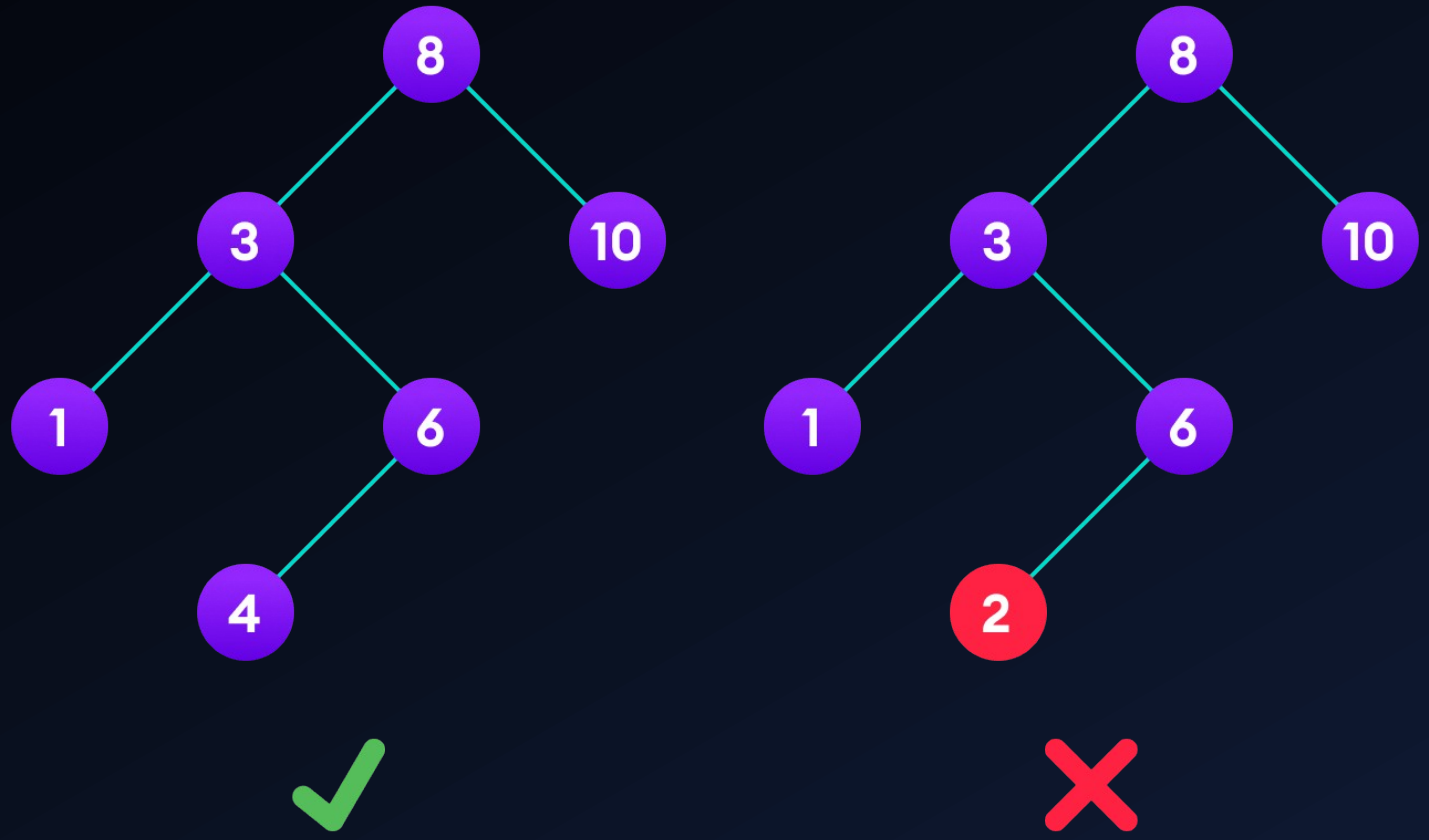
- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in **O(log(n))** time.

The properties that separate a binary search tree from a regular binary tree is:

1. All nodes of the left subtree are less than the root node.
2. All nodes of the right subtree are more than the root node.
3. Both subtrees of each node are also BSTs.

BINARY SEARCH TREE

A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree.



BINARY SEARCH TREE

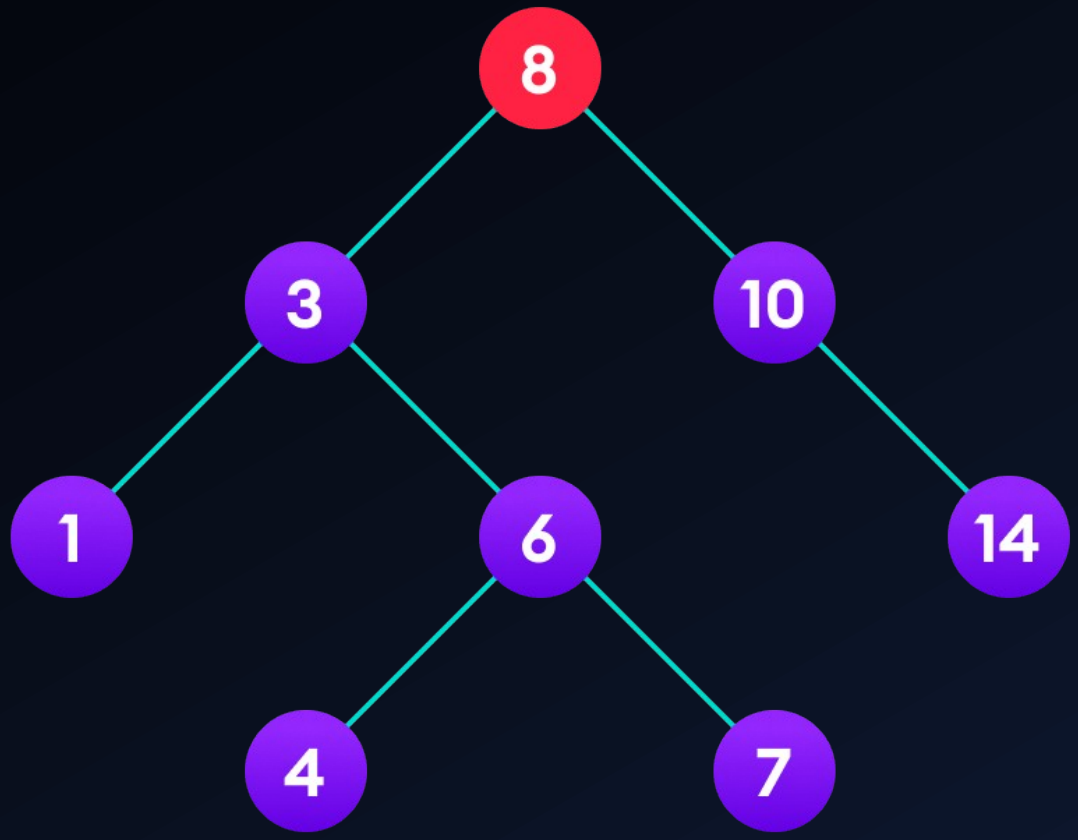
Search Operation depends on the property of BST that if each left subtree has values below the root and each right subtree has values above the root.
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

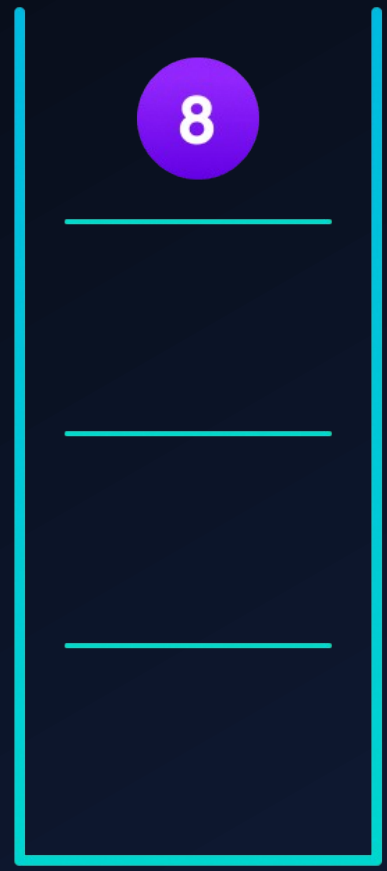
Algorithm:

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

BINARY SEARCH TREE

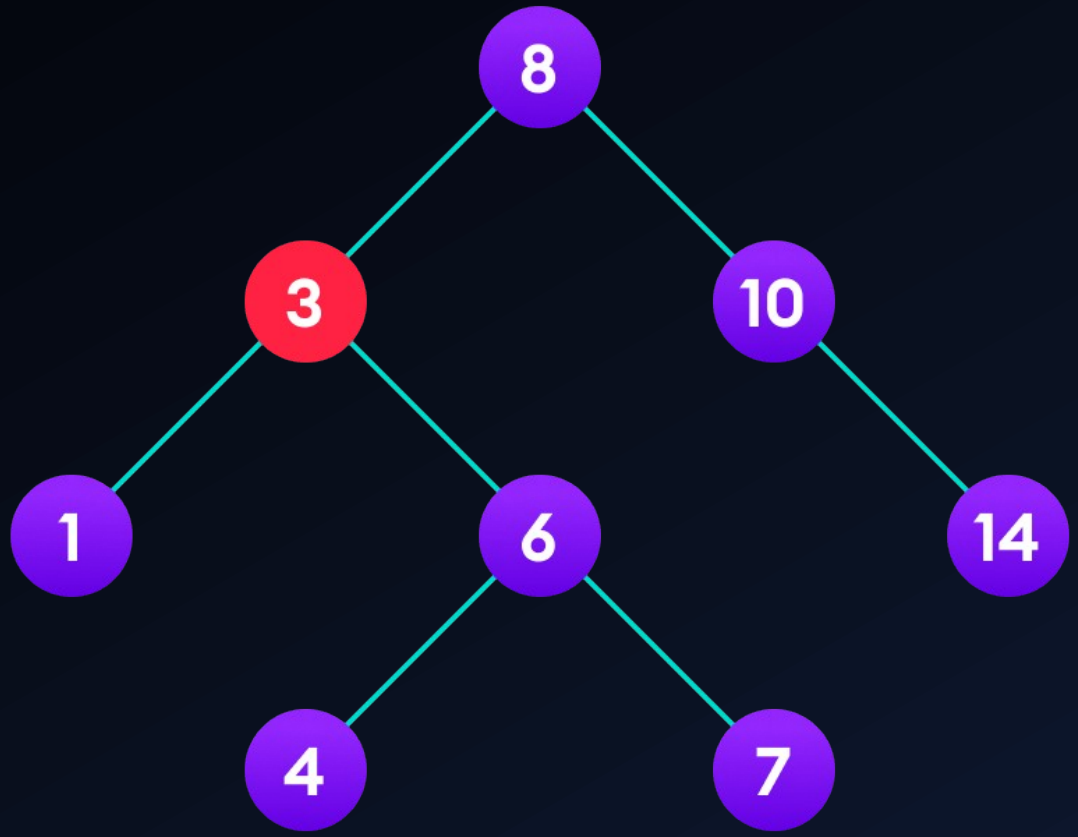
Let us try to visualize this with a diagram. We try to find value **4**. It is not found so, traverse through the left subtree of 8.

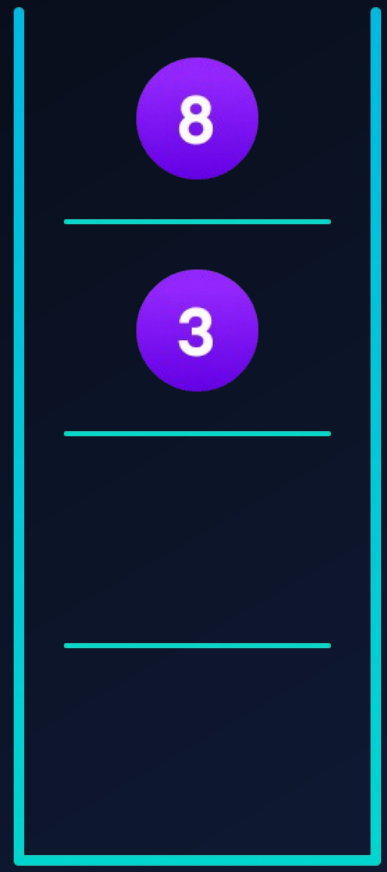




BINARY SEARCH TREE

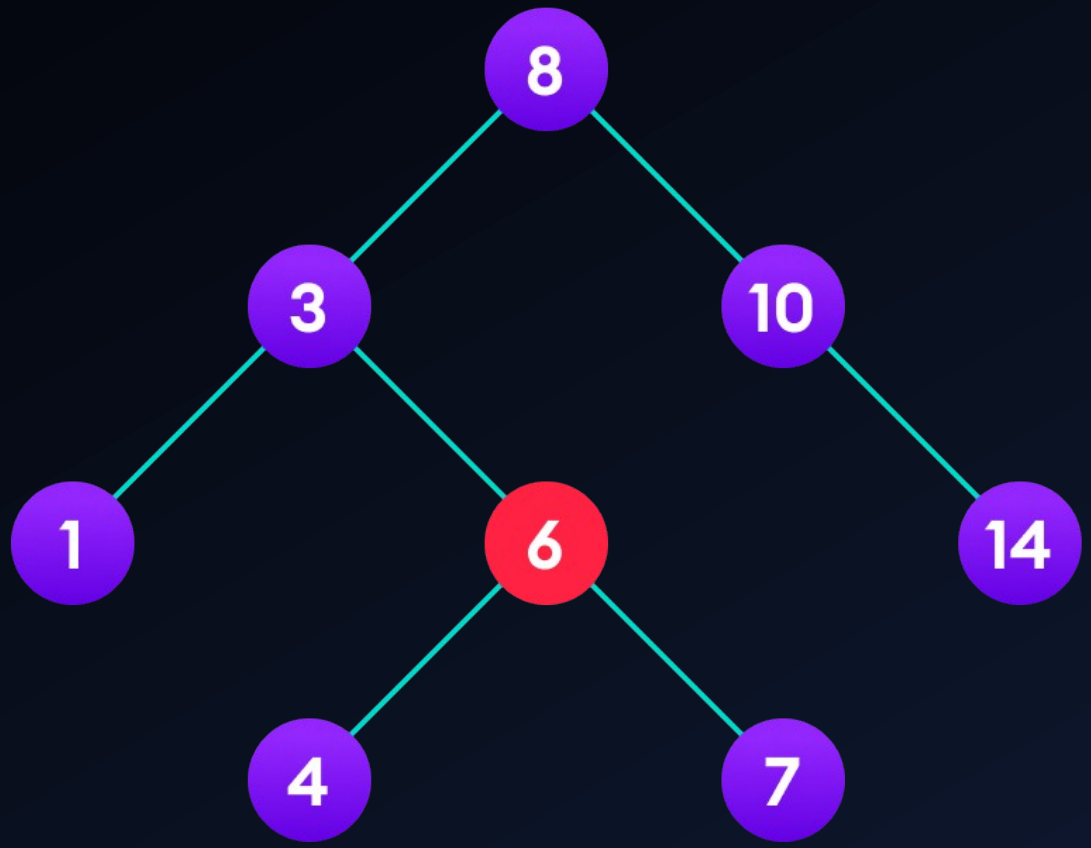
4 is not found so, traverse through the right subtree of 3.

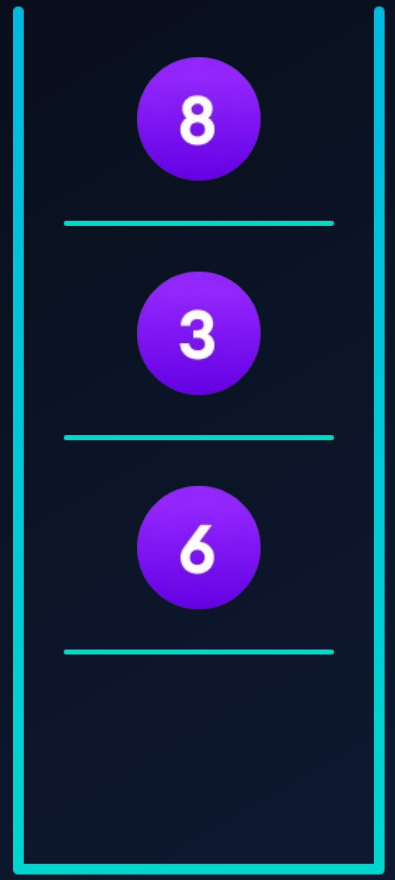




BINARY SEARCH TREE

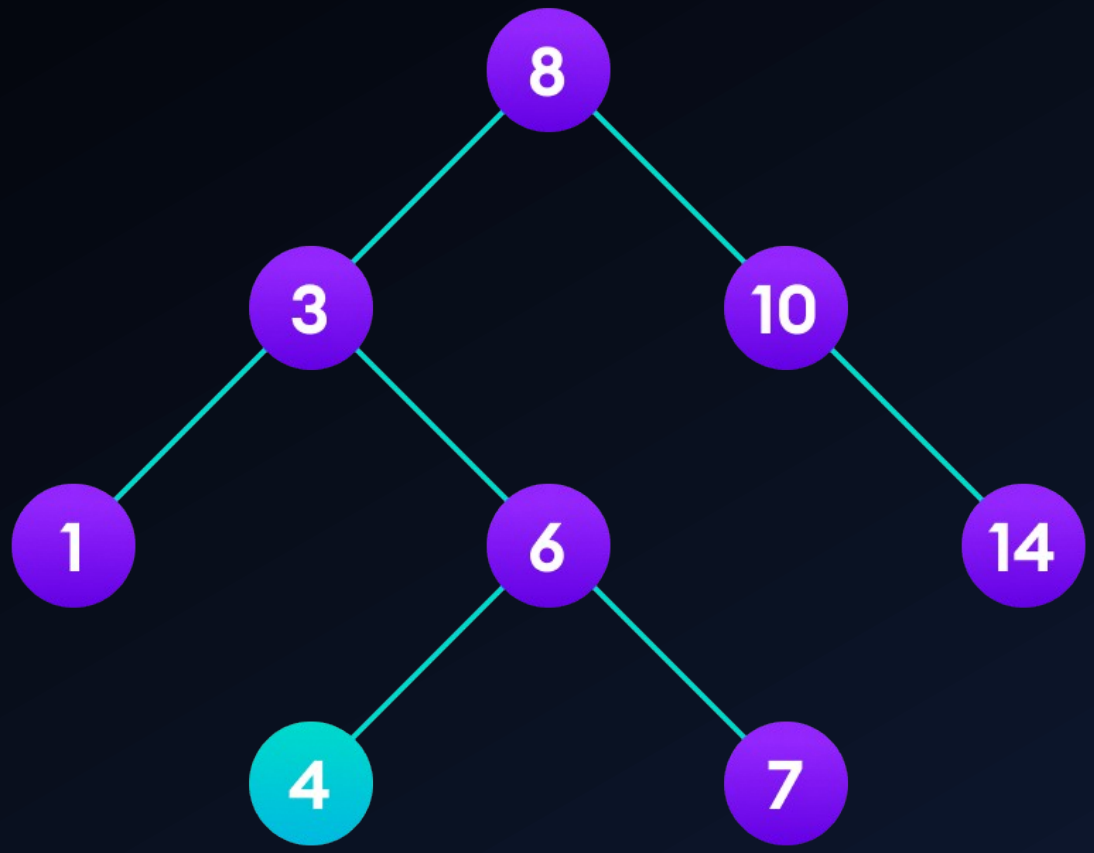
4 is not found so, traverse through the left subtree of 6.

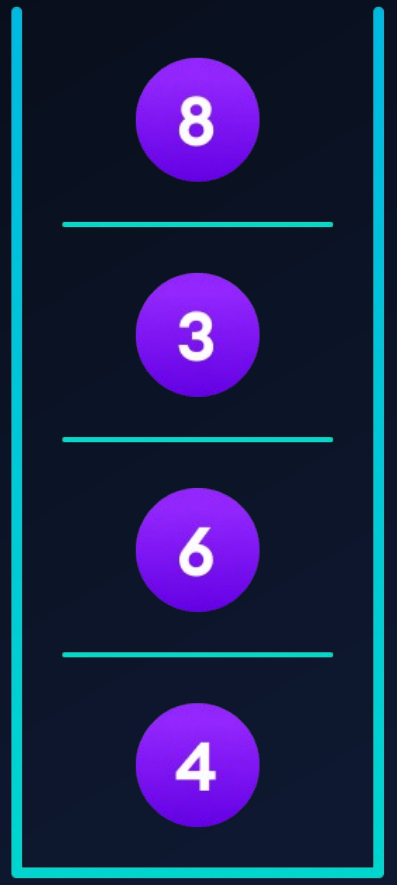




BINARY SEARCH TREE

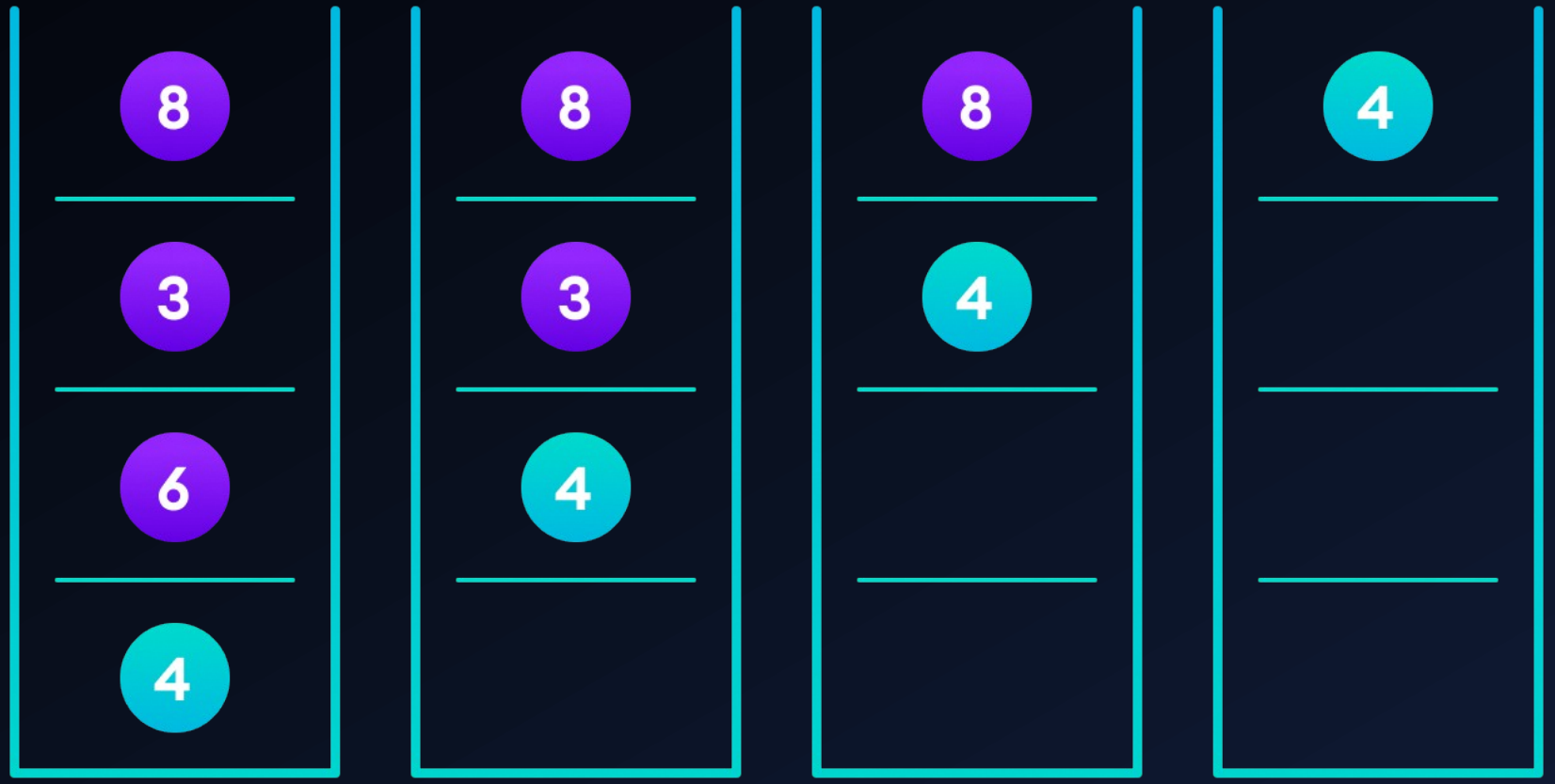
4 is found





BINARY SEARCH TREE

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.



BINARY SEARCH TREE

The performance of a binary search tree is dependent on the order of insertion of the nodes into the tree since arbitrary insertions may lead to degeneracy; several variations of the binary search tree can be built with guaranteed worst-case performance. The basic operations include search, traversal, insert and delete. BSTs with guaranteed worst-case complexities perform better than an unsorted array, which would require linear search time.

The complexity analysis of BST shows that, on average, the insert, delete, and search takes **O(log (n))** for **n** nodes. In the worst case, they degrade to that of a singly linked list **O(n)**.

The binary search tree algorithm was discovered independently by several researchers, including P.F. Windley, Andrew Donald Booth, Andrew Colin, and Thomas N. Hibbard in 1960.

The BST is allowed to use in one dimensional space only.

K-D TREE

A K-Dimensional Tree (also known as K-D Tree) is a space-partitioning data structure for organizing points in a K-Dimensional space. This data structure acts similarly to a binary search tree with each node representing data in the multi-dimensional space.

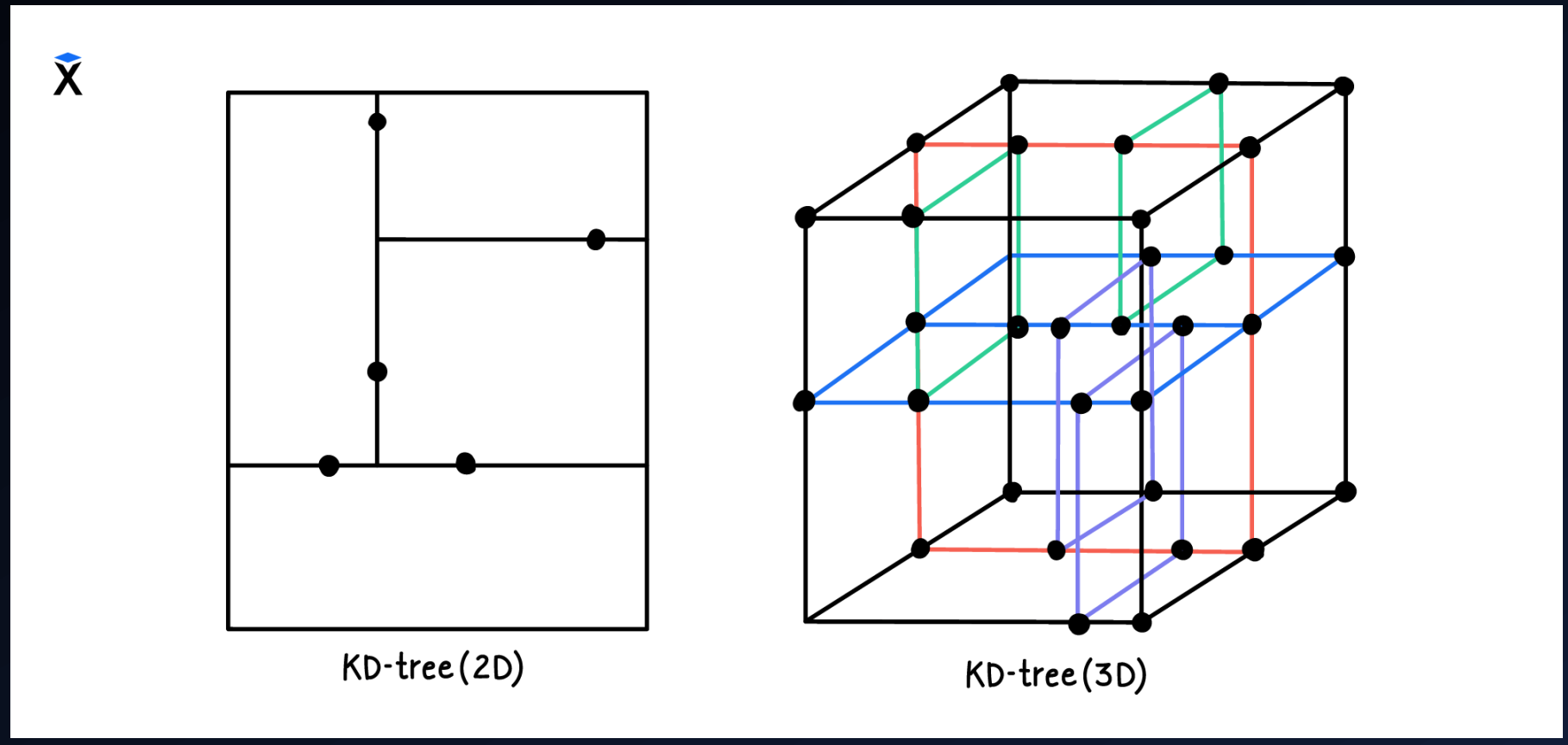
The K-Dimensional Tree was first developed in 1975 by Jon Bentley. The purpose of the tree was to store spatial data with the goal of accomplishing:

1. Nearest neighbor search.
2. Range queries.
3. Fast look-up.

K-D Trees are capable of guaranteeing a **log2(n)** depth, where **n** is the number of points in the set. Since this data structure takes place in a multi-dimensional space, this data structure is incredibly useful right now. Some modern applications of a K-D Tree could range from astrophysical simulation to computer graphics to even data compression. Thanks to being similar in performance to a Binary Search Tree, this data structure also works exceedingly fast.

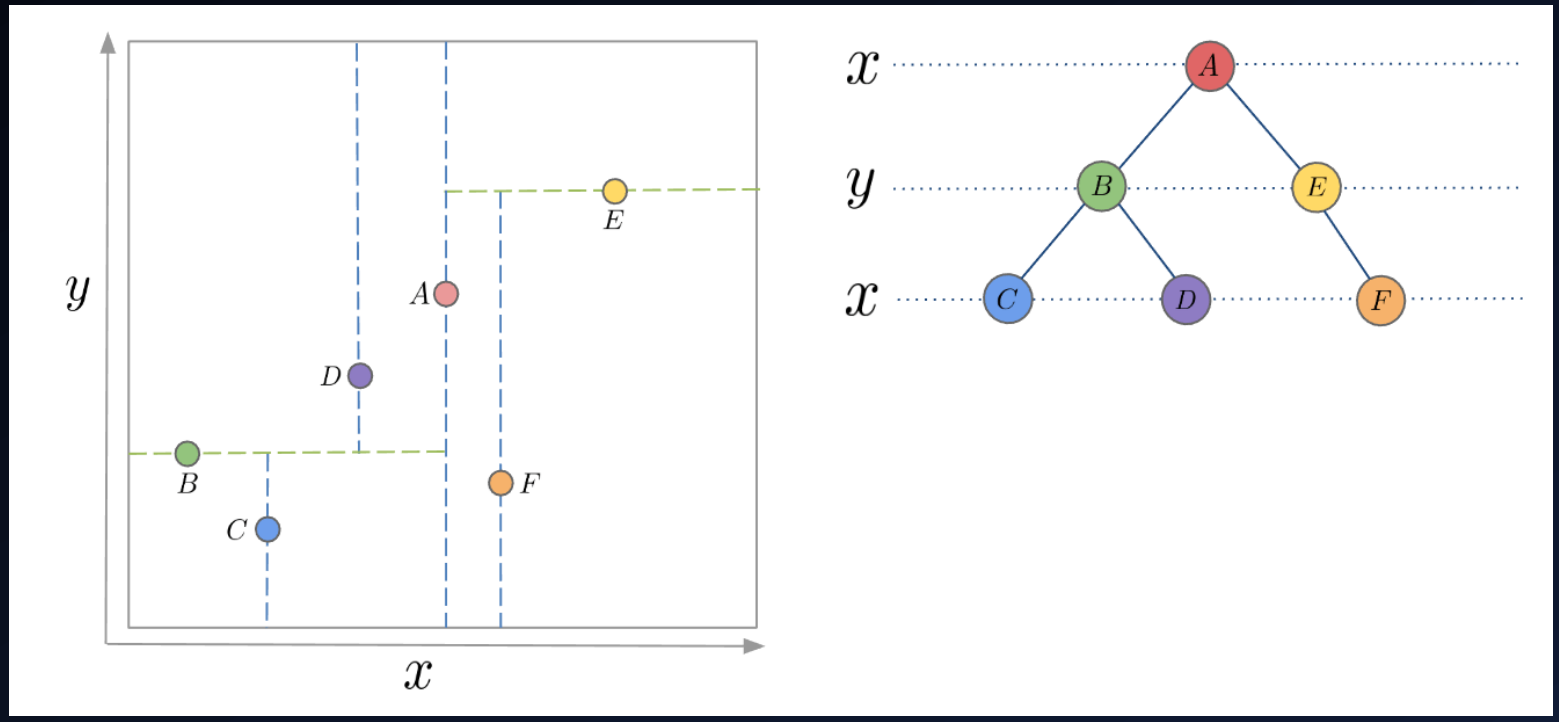
K-D TREE

A K-D Tree is a binary tree in which each node represents a K-dimensional point. Every non-leaf node in the tree acts as a hyperplane, dividing the space into two partitions. This hyperplane is perpendicular to the chosen axis, which is associated with one of the K dimensions.



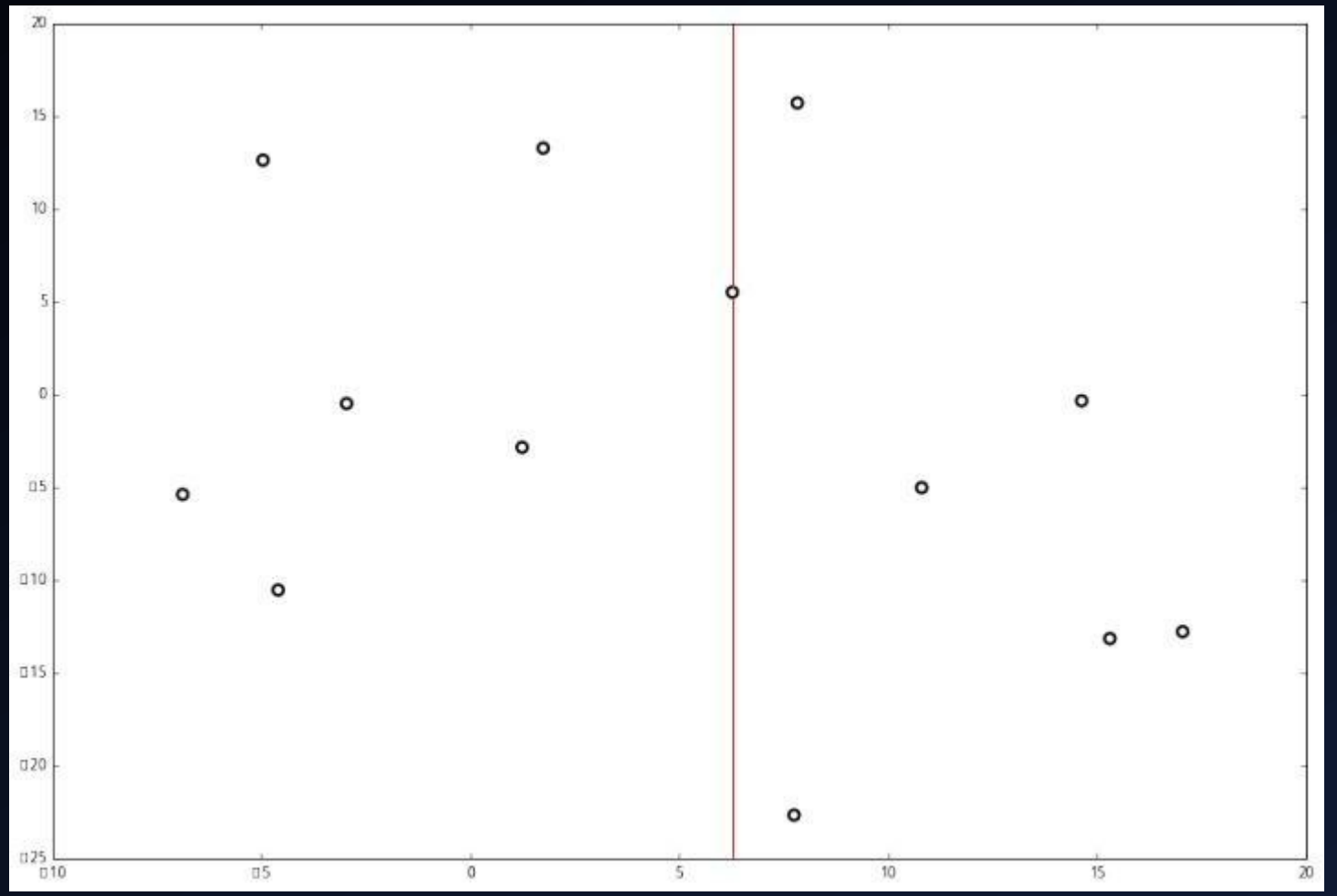
K-D TREE

There are different strategies for choosing an axis when dividing, but the most common one would be to cycle through each of the K dimensions repeatedly and select a midpoint along it to divide the space. For instance, in the case of 2-dimensional points with x and y axes, we first split along the x-axis, then the y-axis, and then the x-axis again, continuing in this manner until all points are accounted for:



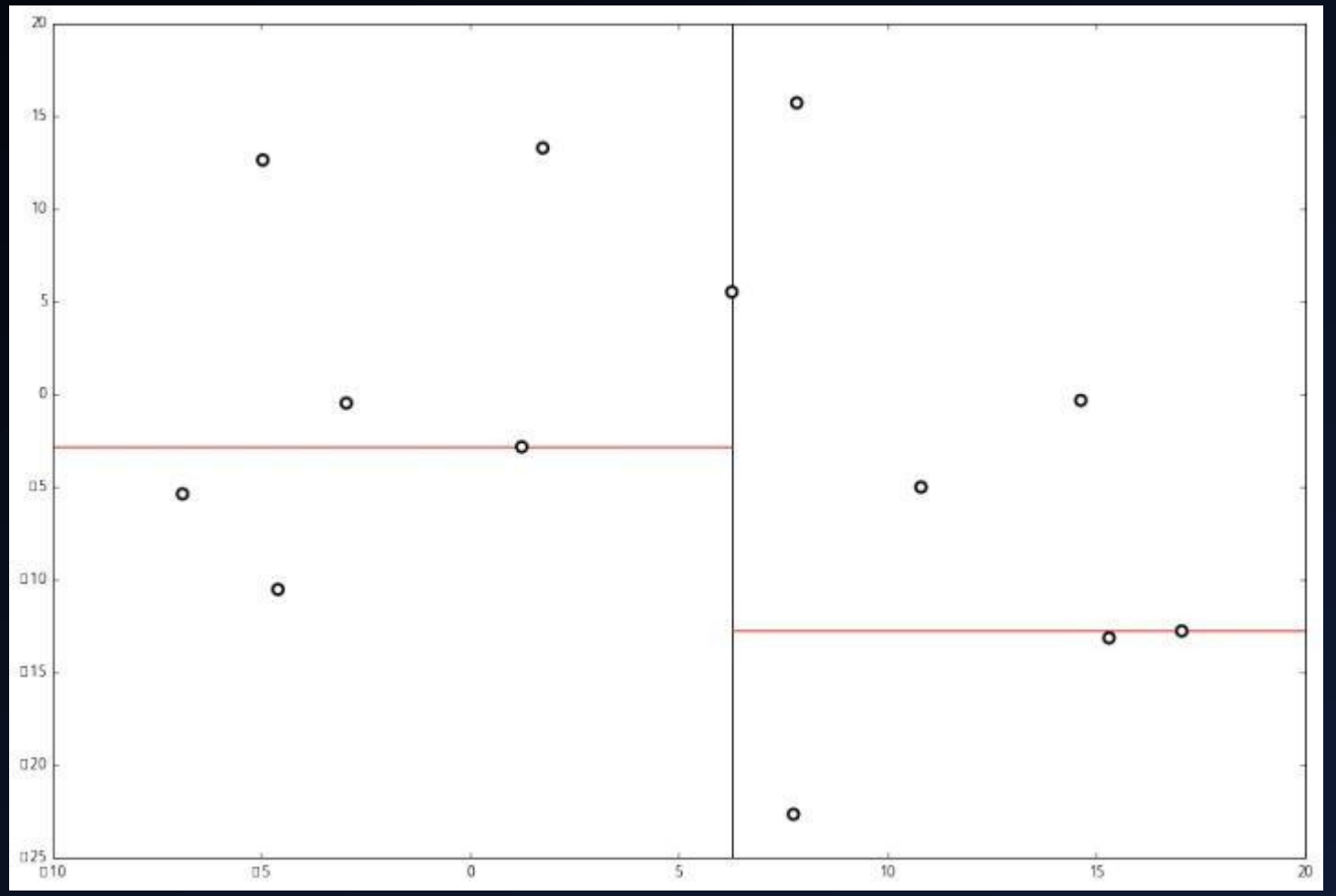
K-D TREE

The construction of a K-D Tree involves recursively partitioning the points in the space, forming a binary tree. We start the process by selecting the X axis.



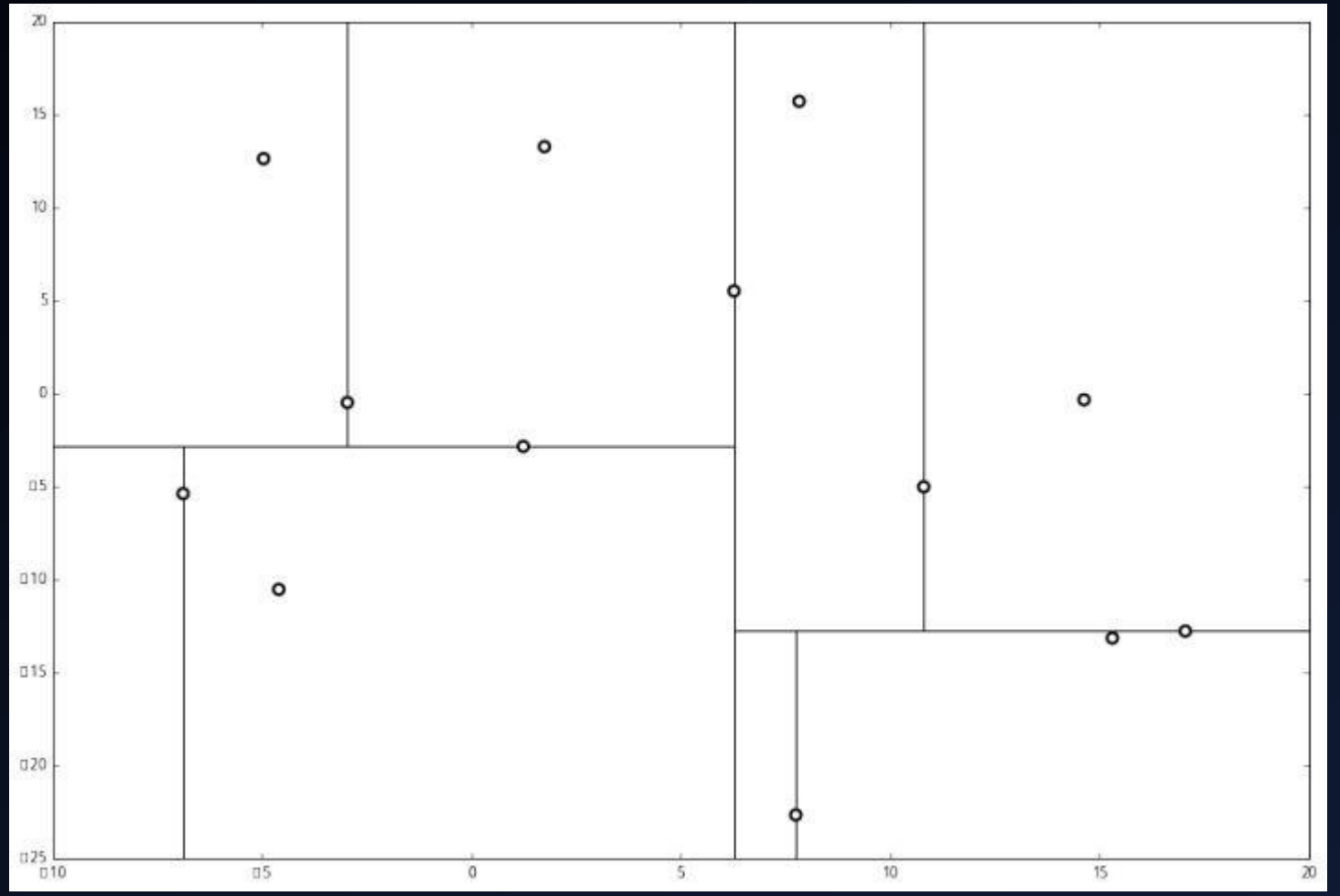
K-D TREE

The next split we perform along the Y axis.



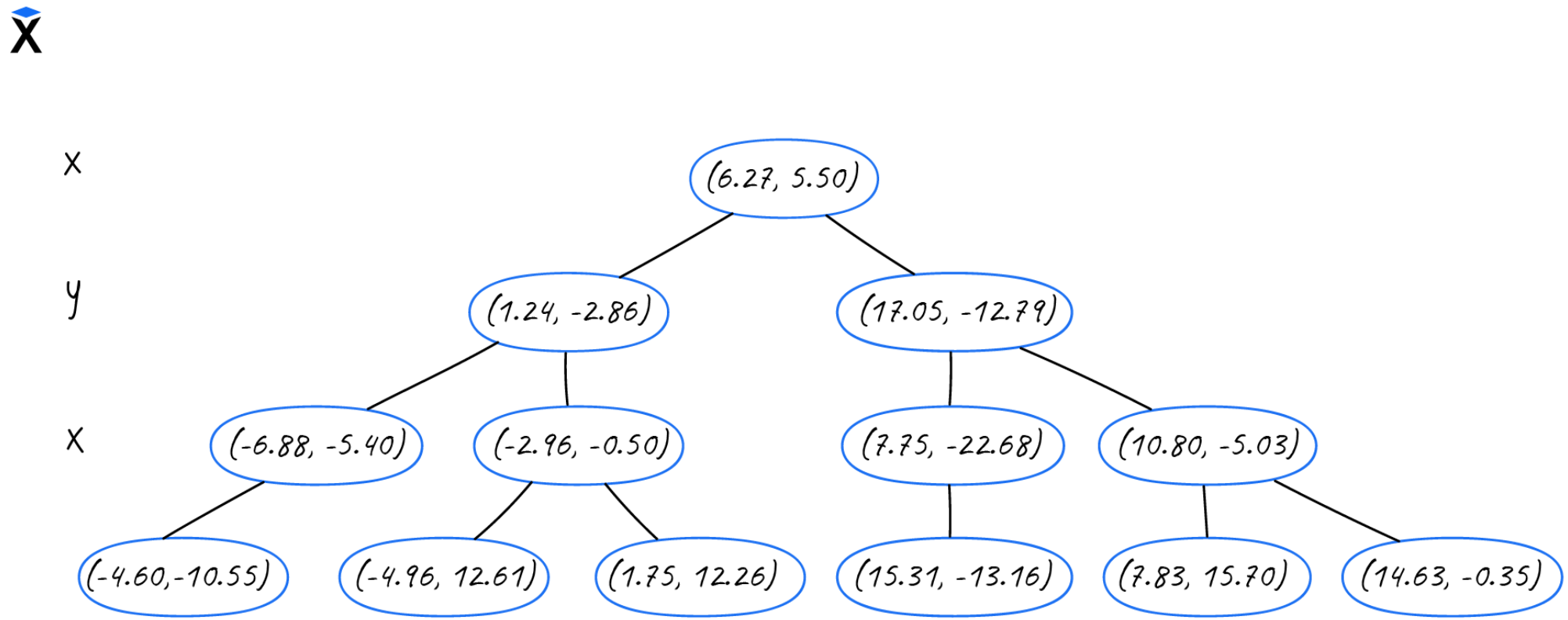
K-D TREE

Continue splitting till it is possible.



K-D TREE

If the algorithm is executed correctly, the resulting tree will be balanced, with each leaf node being approximately equidistant from the root.



K-D TREE

K-D trees are widely used for nearest-neighbor searches, where the objective is to find the point in the tree that is closest to a given query point.

To accomplish this, we traverse the tree and compare the distance between the query point and the points in each leaf node. Starting at the root node, we recursively move down the tree until we reach a leaf node, following a similar process as when inserting a node. At each level, we decide whether to go down the left or right subtree based on which side of the splitting hyperplane the query point lies.

Once we reach a leaf node, we compute the distance between the query point and that leaf node, and save it as the “current best”. **During each unwinding of the recursion, we keep track of the distance and update the current best.** Additionally, at each step, we check whether there could be a point on the other side of the splitting plane that is closer to the search point than the current best.

K-D TREE

Conceptually, we are checking if a hypersphere around the query point with a radius equal to the current nearest distance intersects the splitting hyperplane of the node.

Since the hyperplanes are all axis-aligned this is implemented as a simple comparison to see whether the absolute distance $\mathbf{|cur\_best_{axis} - cur\_node_{axis}|}$ between the splitting coordinate of the search point and the current node is lesser than the overall distance **d** from the search point to the current best.

If the hypersphere intersects the plane, there may be nearer points on the other side of the plane. Therefore, we move down the other branch of the tree from the current node, looking for closer points and following the same recursive process as the entire search.

If the hypersphere doesn't intersect the splitting plane, we continue unwinding and walking up the tree and eliminate the entire branch on the other side of that node. After we finish this process for the root node, the search is complete.

K-D TREE

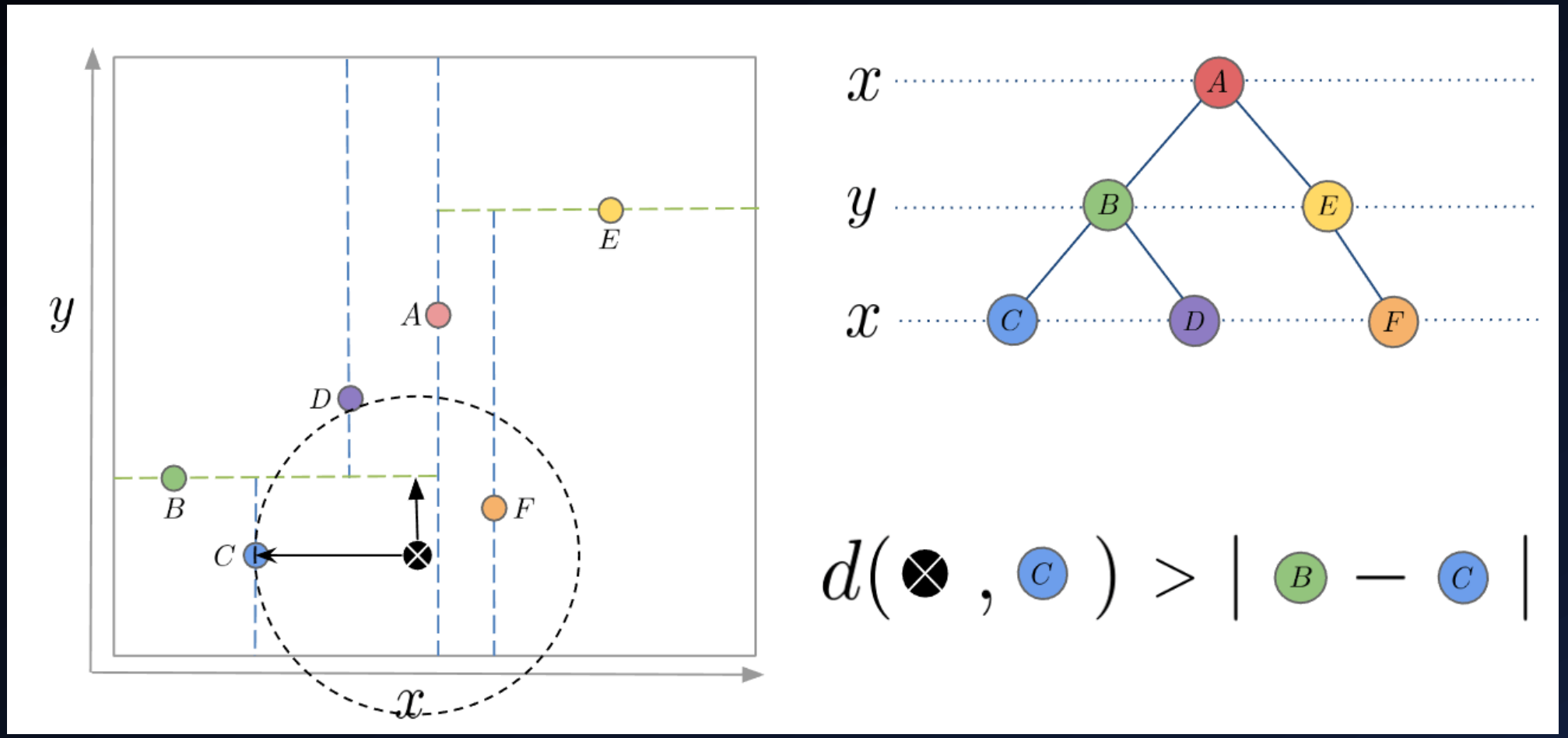
One thing to note is that the distance metric in our example is the classical Euclidian distance defined as:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

The search procedure can certainly be generalized to other distance metrics, however, the “hypersphere” which we test for intersection with the hyperplane will be replaced with the equivalent geometrical object for the given metric.
In the next illustration, we are querying the black crossed dot’s nearest neighbor in the tree. We have already traversed down the tree reached the leaf node **C** and saved it as the current best. So, we are unwinding and comparing the Euclidian distance from it to the query point with the absolute distance from it to the next current node **B**.

K-D TREE

Here the Euclidian distance is bigger, meaning the hypersphere intersects the cutting plane of **B** and therefore we switch to traversing the **E** branch.



K-D TREE

The time complexity of building a K-D tree from a set of n points in K-dimensional space depends on the median finding algorithm used.
If a more efficient median-of-medians algorithm is used the total complexity is **O(n*log(n))** on average. This is because, at each level of the tree, a new hyperplane is chosen to split the set of points, which takes **O(n)** time. Since the tree is binary, the number of levels in the tree is **O(log(n))**. Therefore, the total time complexity of building the *K*-D tree is **O(n*log(n))**. However, in the worst case, the construction time can be $\mathbf{O(n^2)}$ which can occur if the tree is very unbalanced. if instead, a classical sort algorithm with **O(n*log(n))** complexity is used to find the median, the total complexity becomes $\mathbf{O(n*\log^2(n))}$.
The time complexity of the nearest neighbor search in a K-D tree is **O(n*log(n))** on average, where n is the number of points in the tree. This is because the search process involves traversing the tree in a binary search-like manner, where the number of nodes visited is proportional to the height of the tree, which is **O(log(n))** on average.

QUADTREE

A quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are the two-dimensional analog of octrees and are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The data associated with a leaf cell varies by application, but the leaf cell represents a "unit of interesting spatial information".

The subdivided regions may be square or rectangular or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a Q-tree.

All forms of quadtrees share some common features:

- They decompose space into adaptable cells.
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits.
- The tree directory follows the spatial decomposition of the quadtree.

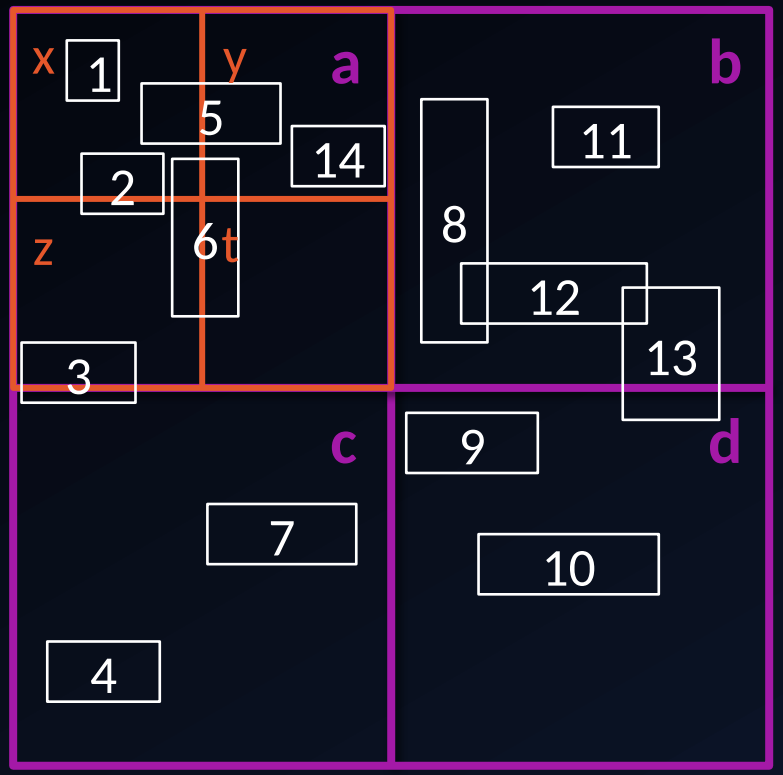
QUADTREE

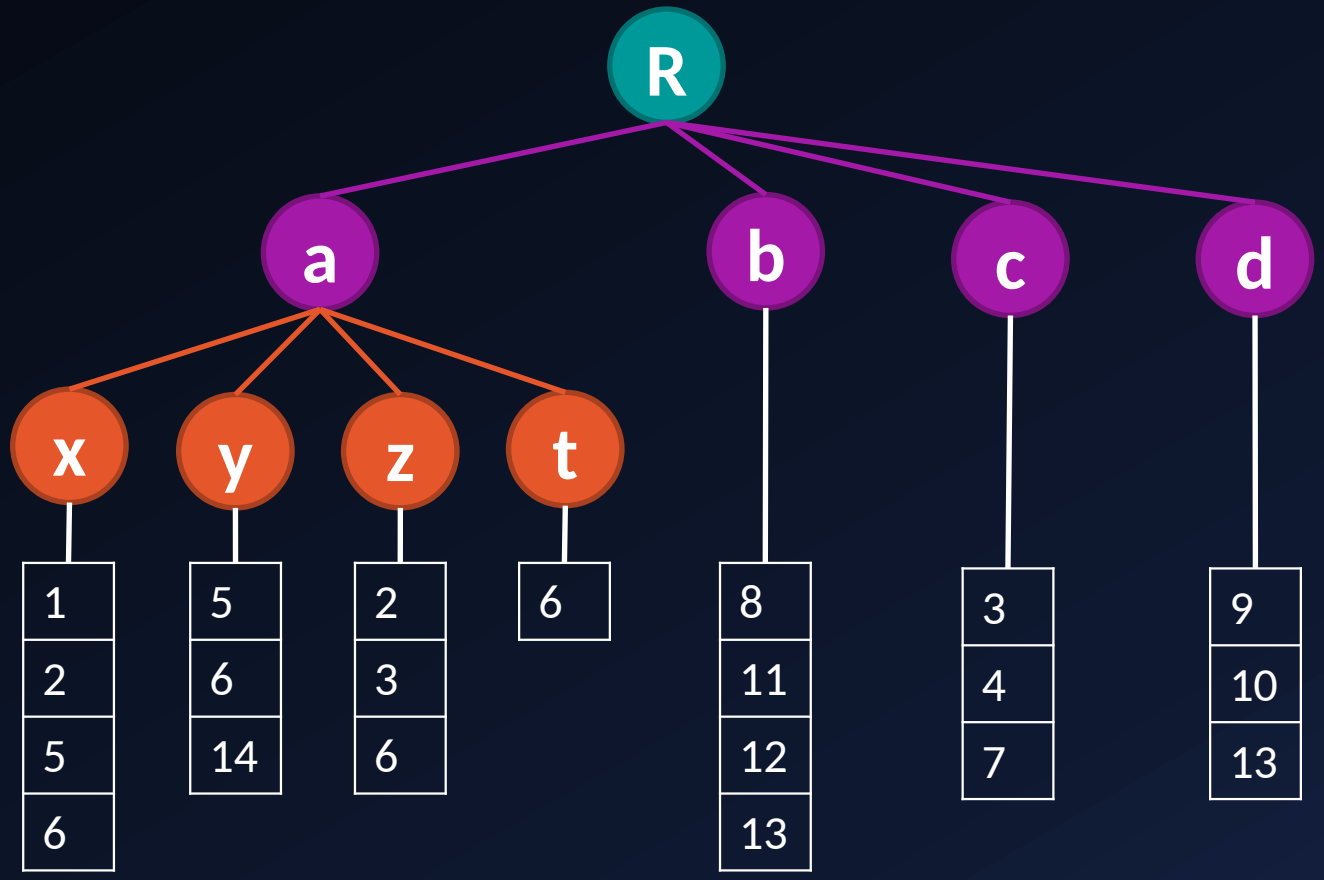
Quadtrees may be classified according to the type of data they represent, including areas, points, lines, and curves. Quadtrees may also be classified by whether the shape of the tree is independent of the order in which data is processed. The following are common types of quadtrees:

- Region quadtree
- Point quadtree
- Node structure for a point quadtree
- Point-region (PR) quadtree
- Edge quadtree
- Polygonal map (PM) quadtree
- Compressed quadtrees

QUADTREE

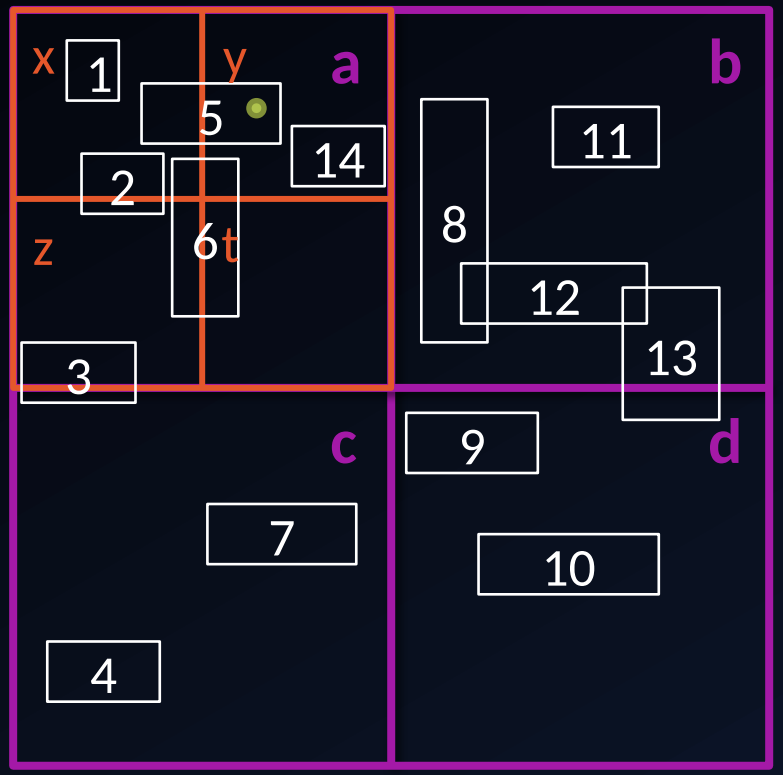
The region quadtree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on with each leaf node containing data corresponding to a specific subregion. Each node in the tree either has exactly four children or has no children (a leaf node).

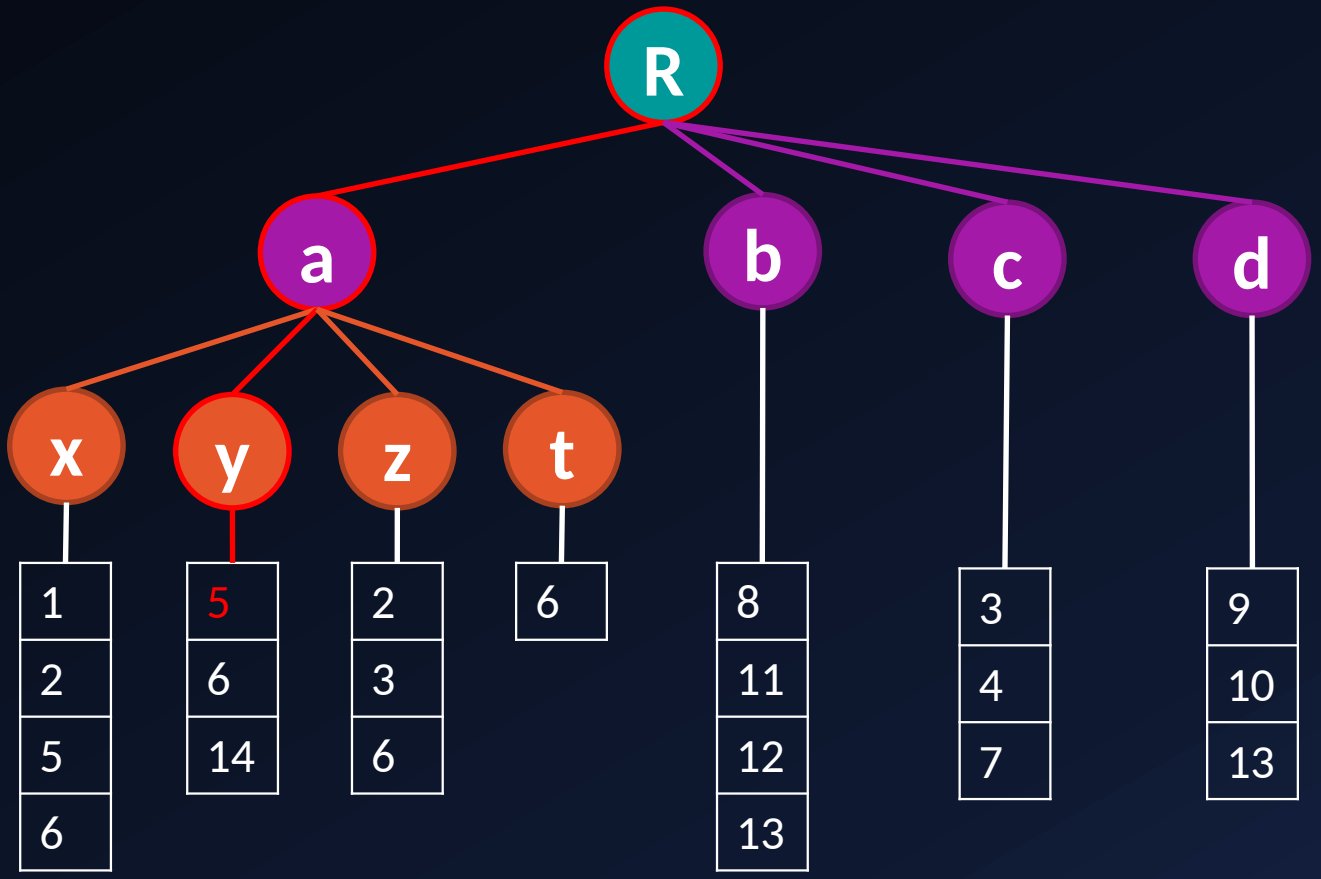




QUADTREE

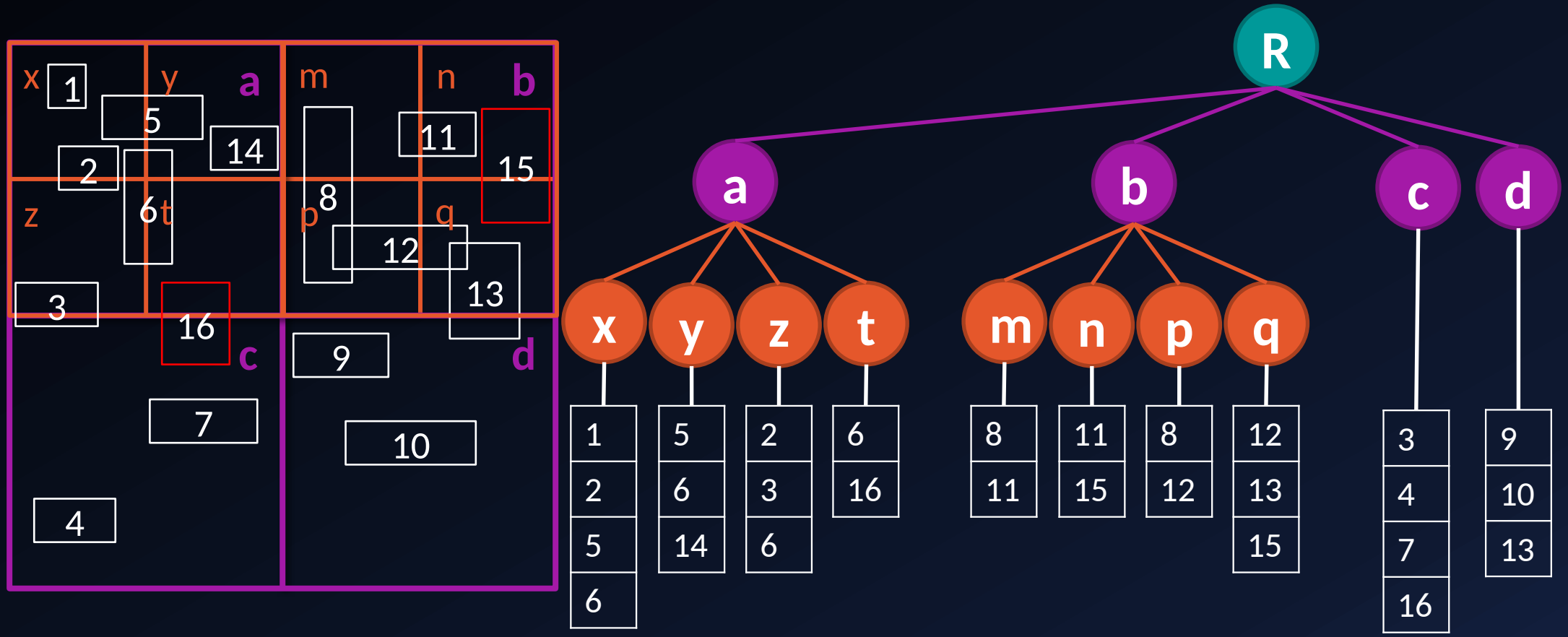
The point request is processed the next way. The tree is traversed from root to leaf and at each level the node which contains the point **P** is chosen.





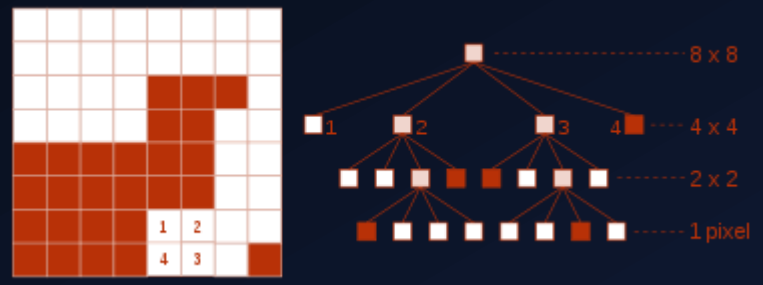
QUADTREE

Let’s consider dynamic object insertion. The object must be added to each quad it lies on. The leaf nodes of the current quad are checked and there are two possible options: quad is overloaded (15) and quad is not overloaded (16). When the first case occurs we have to split the quad into 4 quads and rearrange objects.



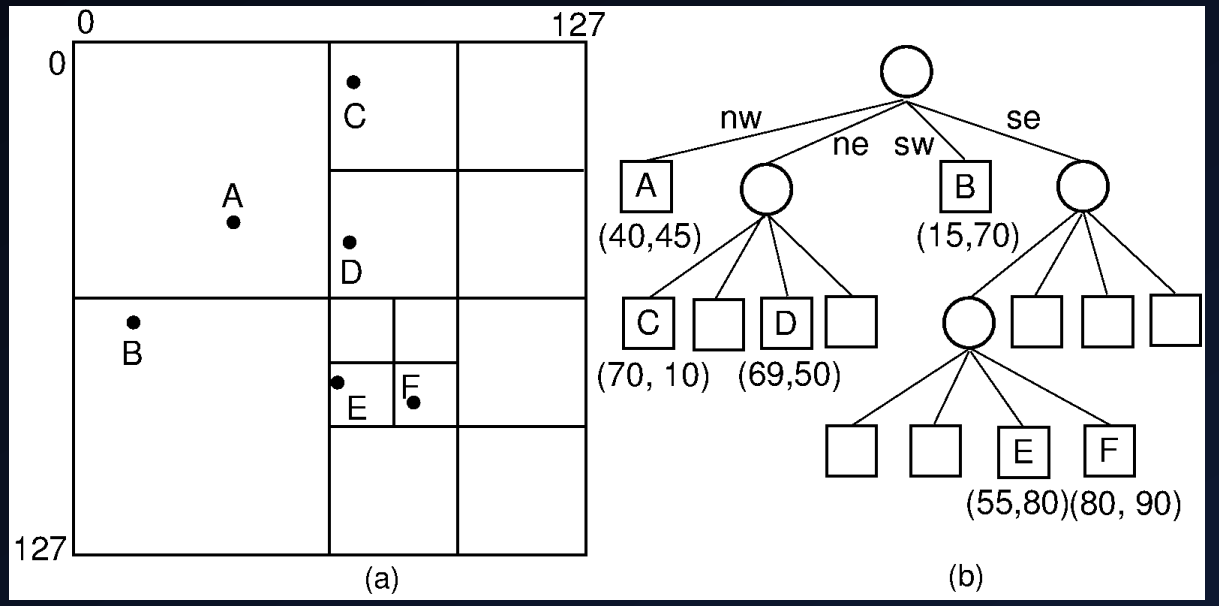
QUADTREE

A region quadtree with a depth of **n** may be used to represent an image consisting of $2^n \times 2^n$ pixels, where each pixel value is **0** or **1**. The root node represents the entire image region. If the pixels in any region are not entirely **0s** or **1s**, it is subdivided. In this application, each leaf node represents a block of pixels that are all **0s** or all **1s**. Note the potential savings in terms of space when these trees are used for storing images; images often have many regions of considerable size that have the same color value throughout. Rather than store a big 2-D array of every pixel in the image, a quadtree can capture the same information potentially many divisive levels higher than the pixel-resolution-sized cells that we would otherwise require. The tree resolution and overall size are bounded by the pixel and image sizes.



QUADTREE

The Point quadtree is an adaptation of a binary tree used to represent two-dimensional point data. It shares the features of all quadtrees but is a true tree as the center of a subdivision is always on a point. It is often very efficient in comparing two-dimensional, ordered data points, usually operating in **O(log(n))** time. Point quadtrees are worth mentioning for completeness, but they have been surpassed by **K-D trees** as tools for generalized binary search.
An example of the interactive Point quadtree is here.



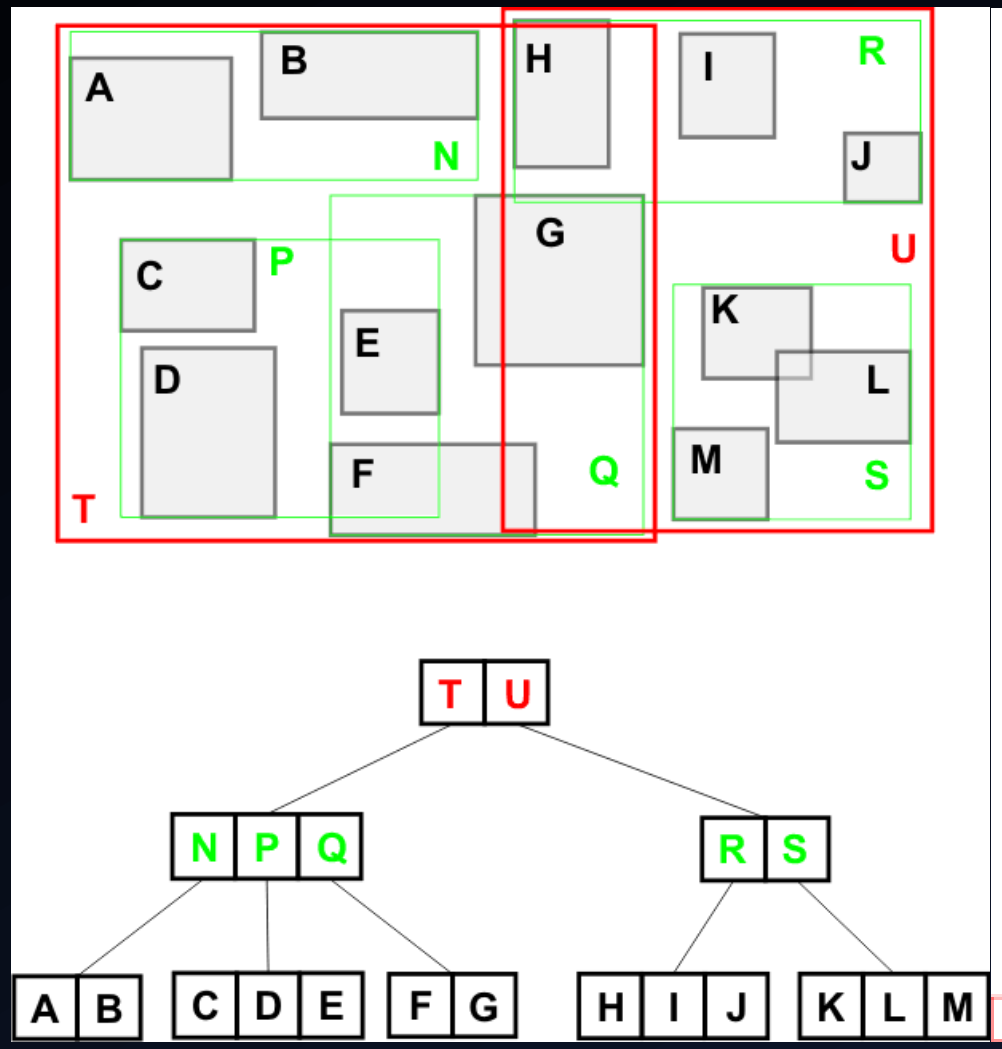
R-TREE

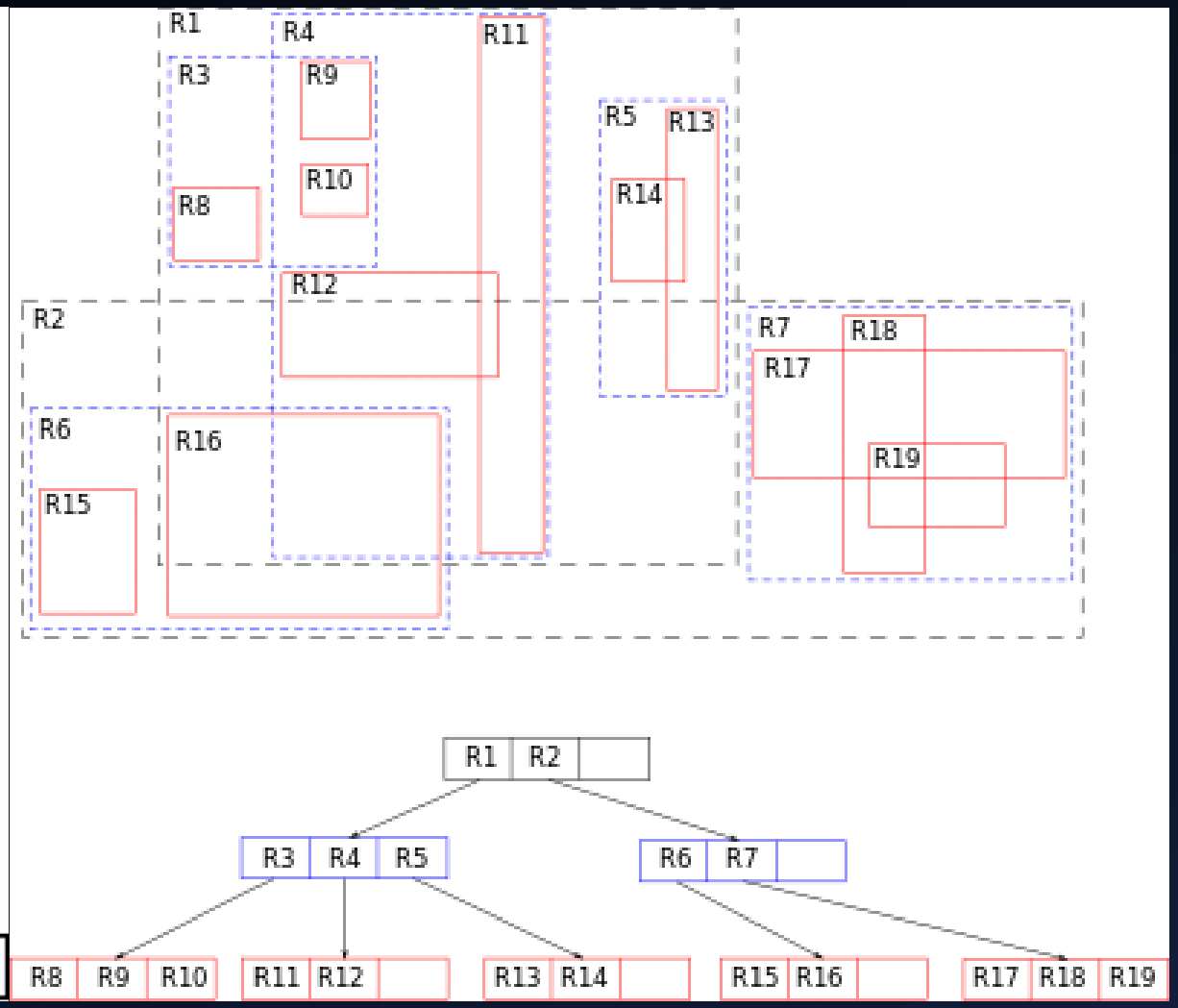
R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles, or polygons. The R-tree was proposed by Antonin Guttman in 1984 and has found significant use in both theoretical and applied contexts.

The key idea of the data structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree; the "R" in the R-tree is for the rectangle. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object (consists of object ID and MBR); at higher levels, the aggregation includes an increasing number of objects (consists of node ID and directory rectangle). This can also be seen as an increasingly coarse approximation of the data set.

R-TREE

Set of rectangles indexed by an R-Tree and the corresponding R-Tree structure.





R-TREE

There are next variations of the R-tree:

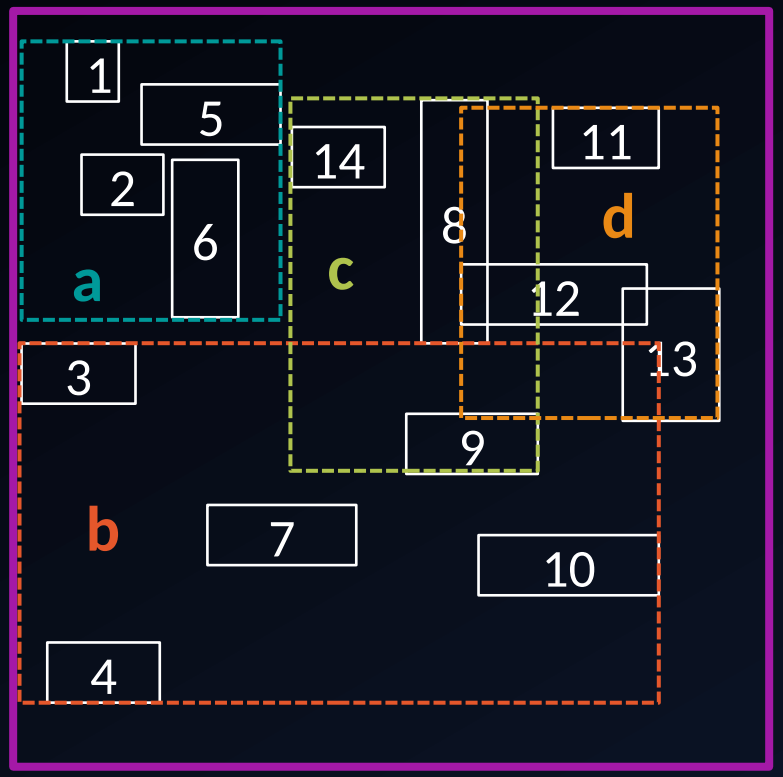
- Priority R-tree
- R*-tree
- R+ tree
- Hilbert R-tree
- X-tree

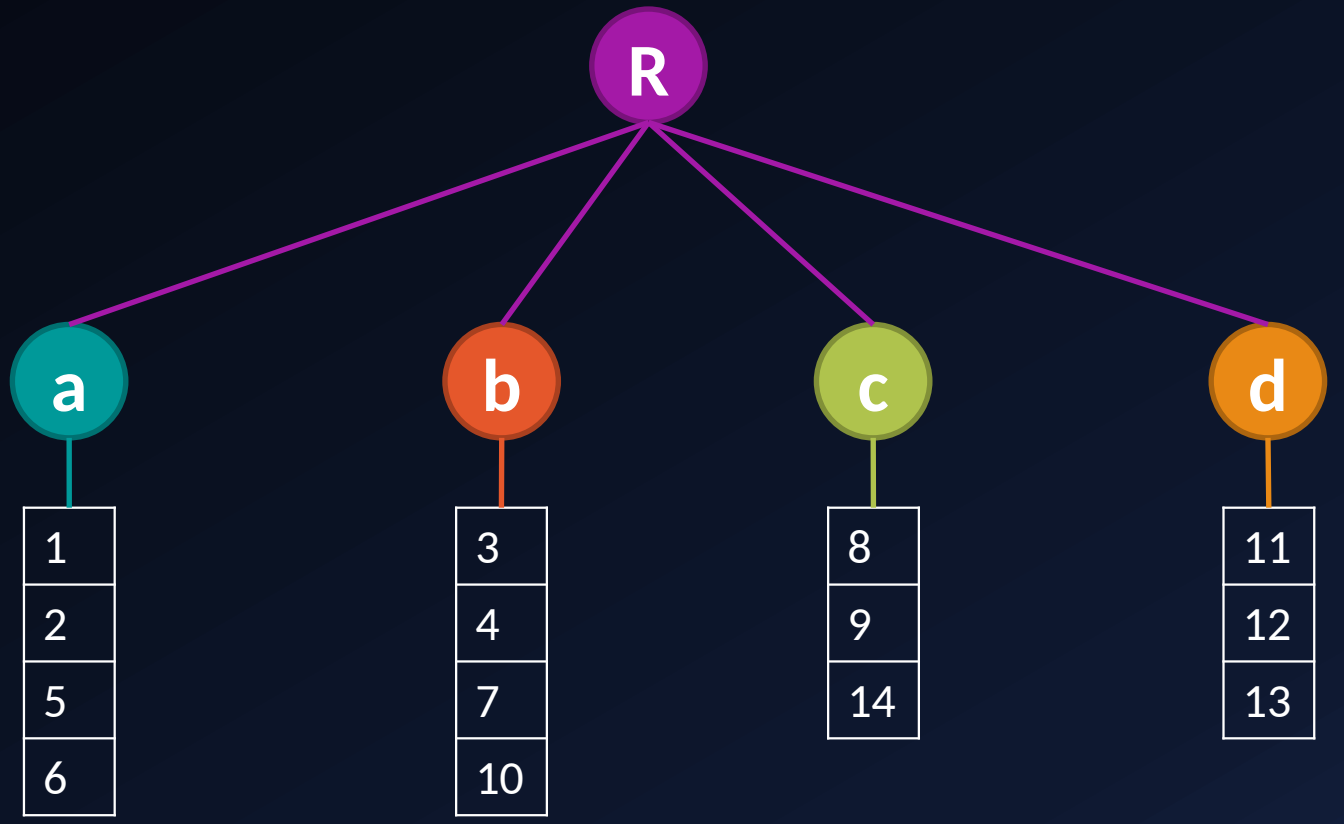
Properties of R-tree:

- Consists of a single root, internals nodes, and leaf nodes.
- The root contains the pointer to the largest region in the spatial domain.
- Parent nodes contain pointers to their child nodes where the region of child nodes completely overlaps the regions of parent nodes.
- Leaf nodes contain data about the MBR to the current objects.
- MBR-Minimum bounding region refers to the minimal bounding box parameter surrounding the region/object under consideration.

R-TREE

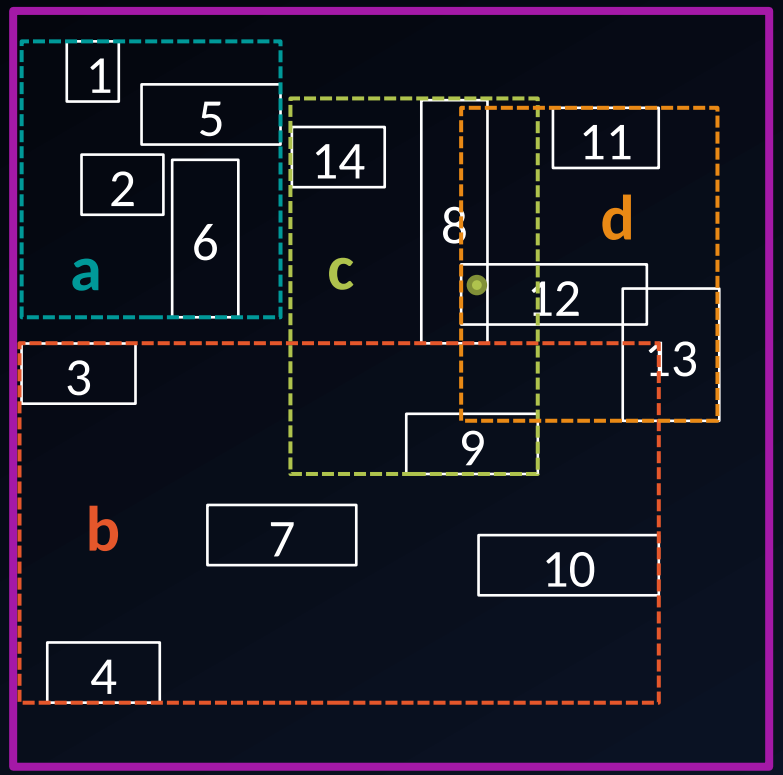
The number of entries of a node (except for the root) in the tree is between **m** and **M** where **m**∈[0, **M**/2]. Below is the example with **m=2** and **M=4**.

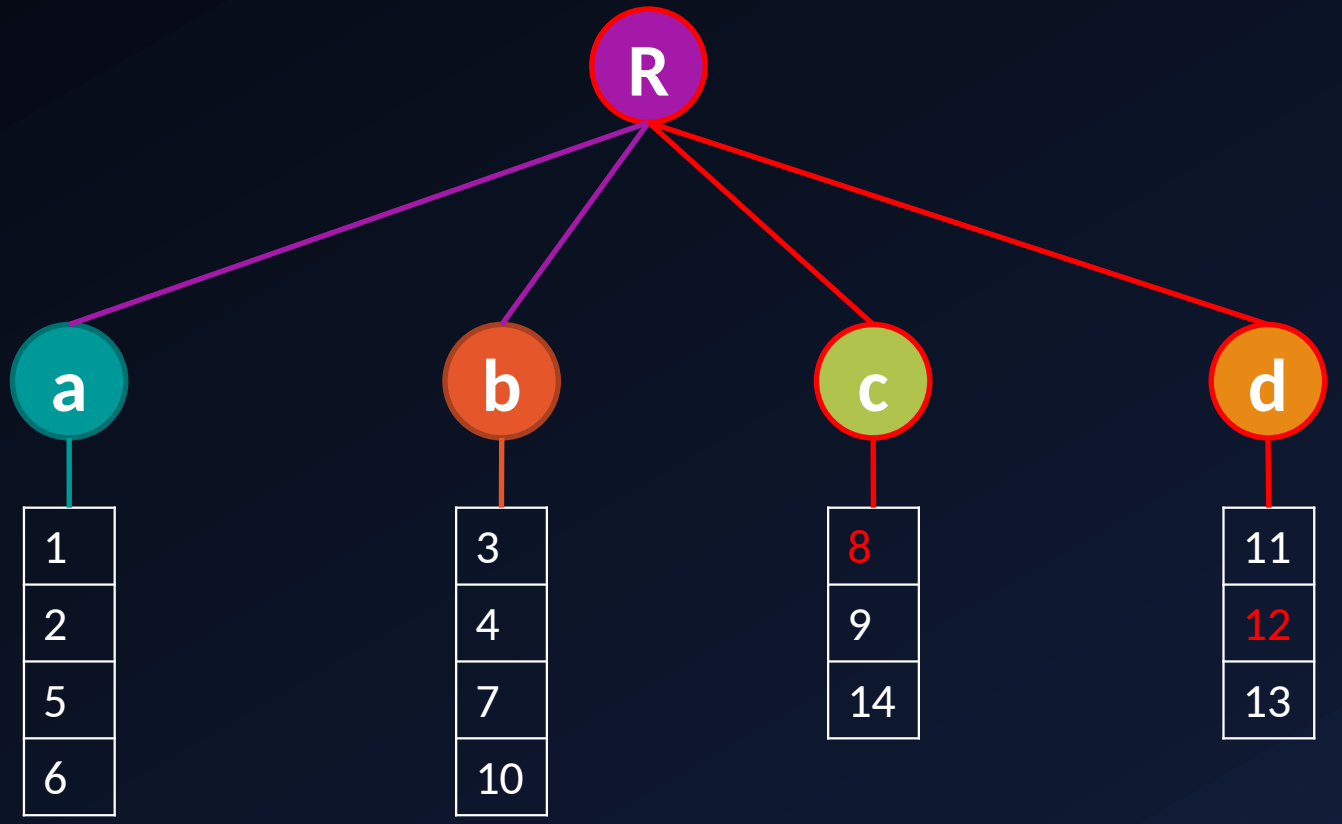




R-TREE

The point search function is performed in two stages. First, a search is made for all nodes whose directory rectangle contains the point **P**. All subtrees are considered, since the point may belong to the intersection of several rectangles.





R-TREE

The time complexity of the search.

- If MBRs do not overlap on **q**, the complexity is **$O(\log_m(N))$**.
- If MBRs overlap on **q**, it may not be logarithmic, in the worst case when all MBRs overlap on **q**, it is **O(N)**.

Comparison with Quadtrees:

- Tiling level optimization is required in Quadtrees whereas an R-tree doesn't require any such optimization.
- Quadtree can be implemented on top of existing B-tree whereas R-tree follows a different structure from a B-tree.
- Spatial index creation in Quadtrees is faster as compared to R-trees.
- R-trees are faster than Quadtrees for Nearest Neighbor queries while for window queries, Quadtrees are faster than R-trees.

R-TREE

Insertion – choose a leaf node

- Traverse the R-tree top-down, starting from the root, at each level
  - If there is a node whose directory rectangle contains the MBR to be inserted, then search the subtree
  - Else choose a node such that the enlargement of its directory rectangle is minimal, then search the subtree
  - If more than one node satisfies this, choose the one with the smallest area
- Repeat until a leaf node is reached

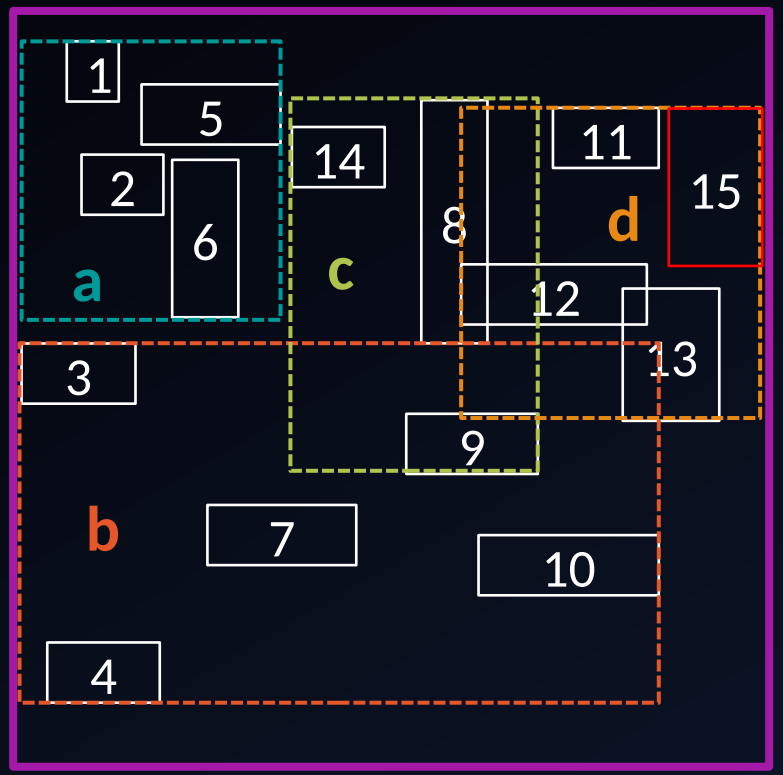
Insertion – insert into the leaf node

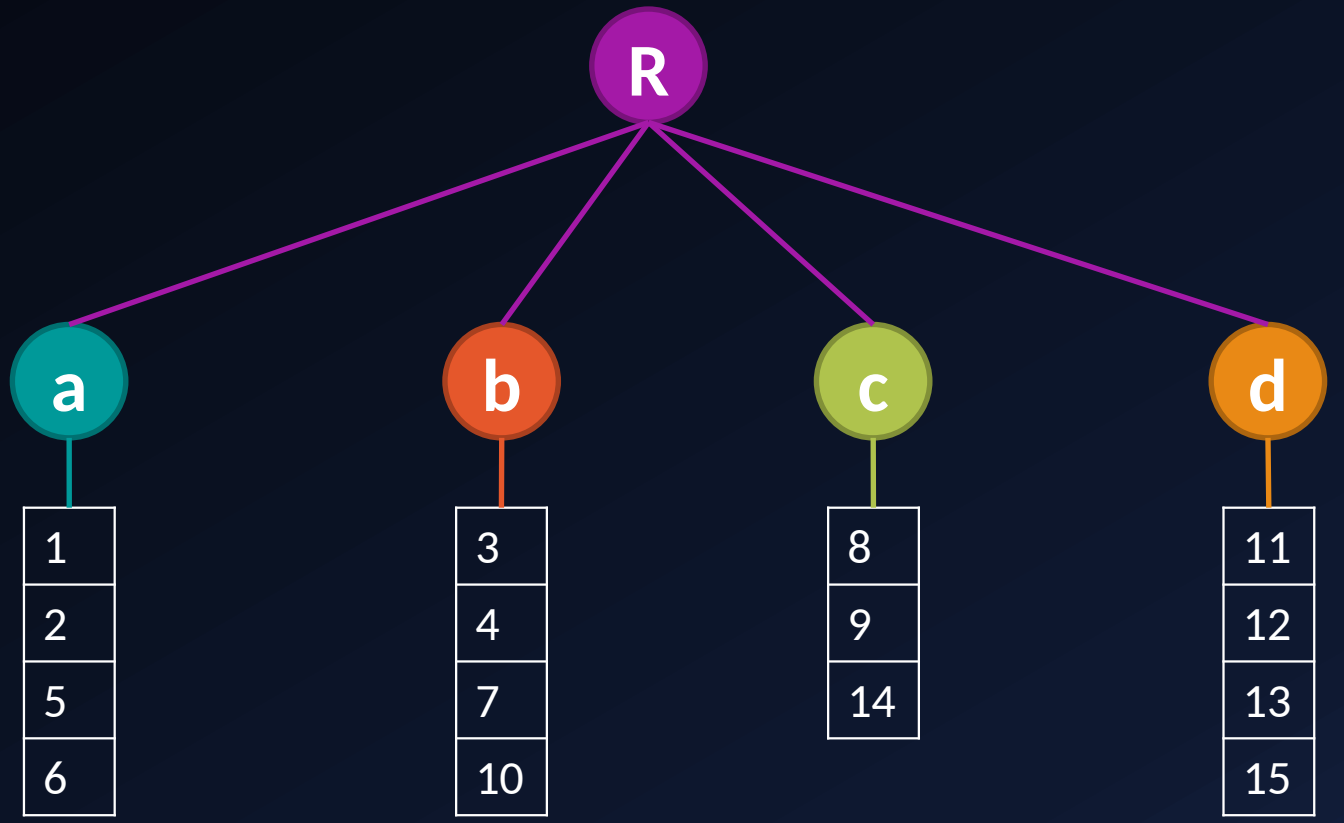
- If the leaf node is not full, an entry [MBR, OID] is inserted
- Else // the leaf node is full
  - Split the leaf node
  - Update the directory rectangles of the ancestor nodes if necessary

Let’s consider both examples.

R-TREE

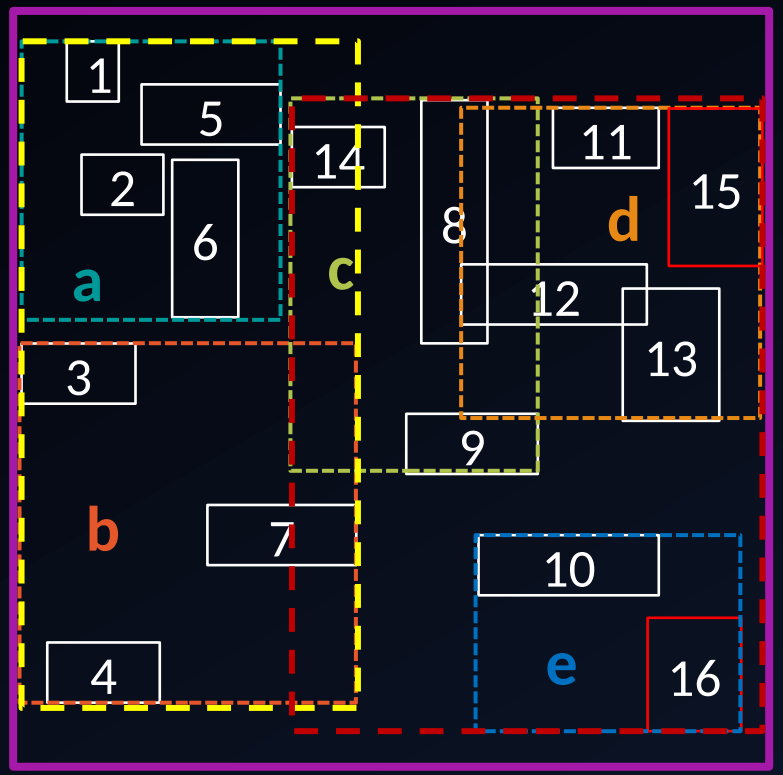
Insert object 15. The directive rectangle of the leaf node **d** is not full, so we just adding object 15 into this leaf.

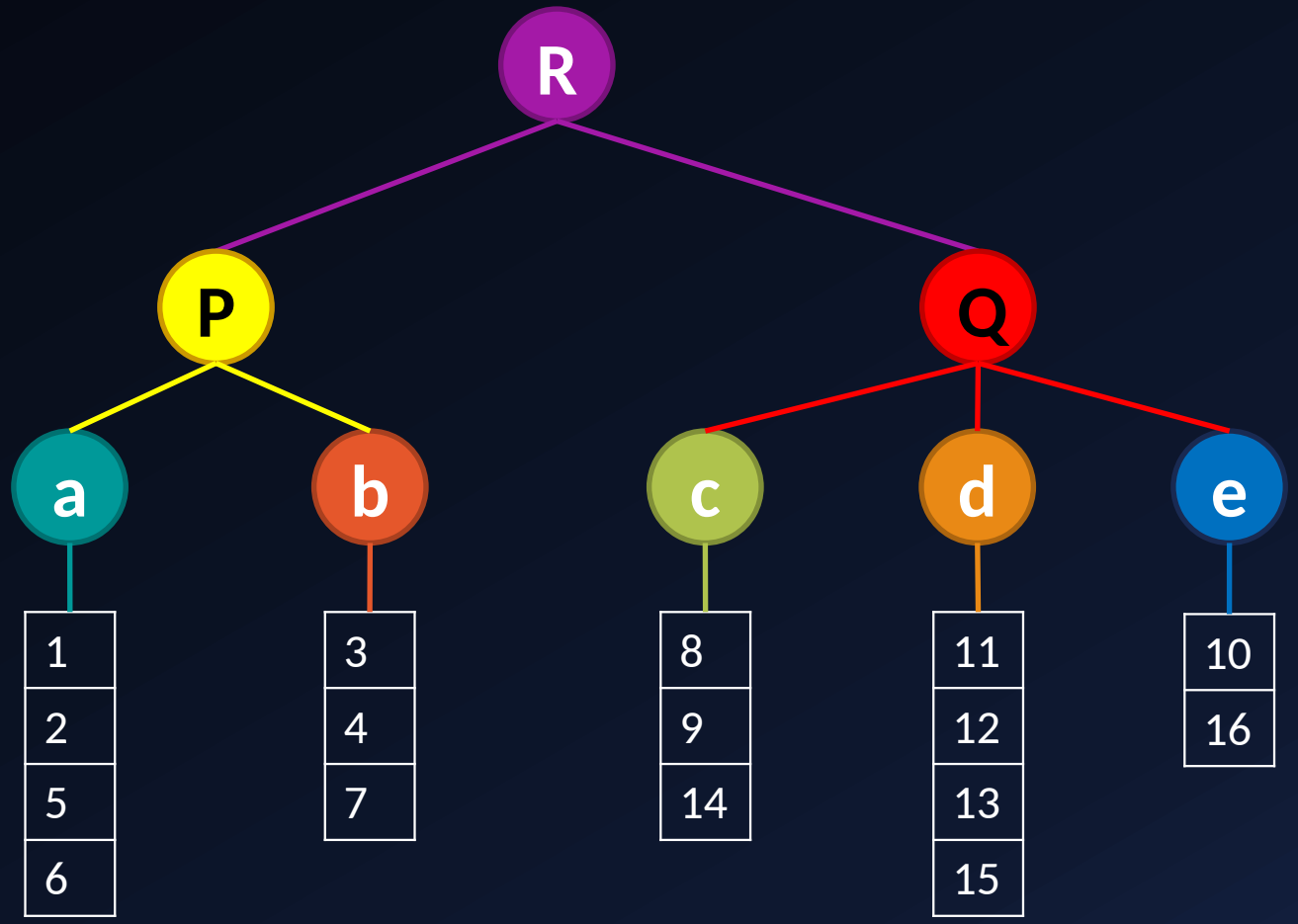




R-TREE

Insert object 16. The directive rectangle of the leaf node **b** is full, so we need to split node **b** and rearrange objects 3, 4, 7, 10, and 16. Then we have 5 leaf nodes, but our **M=4**, so we need to add one internal level to the R-tree.





Thank you!