# Estructuras de datos multidimensionales.

P(N): costo de preprocesamiento.
S(N): costo de almacenamiento.
Q(N): costo de acceso.
Static - Dynamic

# Maldicion de la dimensionalidad.

La maldicion de la dimensionalidad es un termino introducido por Bellman para describir el problema causado por el aumento EXPONENCIAL en volumen asociado con añadir dimensiones extra en un espacio euclidiano.

# Análisis comparativo de las técnicas de procesamiento de consultas del vecino más cercano.

Table 2. Computational Complexity of Data Structures

| Data Structure Name | Idea | Search Complexity (Query Time) | Search Complexity (Space) |
|---|---|---|---|
| LSH | Mapping data point by using hash function-search based on hash function. | O(n) | O(1) |
| KD-Tree | Making binary tree recursively based on the mean of points- search based on nearest query region | O(n + k) | O(1) |
| Quad-Tree | Making Quad-tree recursively based on same zones (or based on the place of points)-search based on nearest query region | O(n) | O(1) |
| Oct-Tree | Making Oct-tree recursively based on axis-search based on nearest query region | O(n) | O(1) |
| Ball-Tree | Making binary tree recursively based on axis-search based on data pruning and the distance from the axis. | O(n) | O(1) |
| R-Tree | Making R-tree into down to up based on MBR-search based on data pruning and MBR | O(n) | O(log n) |
| M-Tree | Making M-tree into down to up based on the radius of adjacent data. Search based on data pruning and the radius of nodes. | O(log n) | O(log n) |

# Estructuras de datos para busquedas de rango.

Scan secuencial.

$$P_{\mathrm{SS}}(N,k) = O(Nk),$$
$$S_{\mathrm{SS}}(N,k) = O(Nk),$$
$$Q_{\mathrm{SS}}(N,k) = O(Nk).$$

Proyección.

$$P_{\mathrm{p}}(N,k) = O(kN \log N),$$
$$S_{\mathrm{p}}(N,k) = O(kN).$$

$$Q_{\mathrm{p}}(N,k) = O(N^{1-1/k}) \quad \textbf{(average case)}$$

Cells / Grid.

$$P_c(N, k) = O(Nk),$$
$$S_c(N, k) = O(Nk),$$
$$Q_c(N, k) = O(2^k F) \quad \text{(average)},$$

K-d trees.

$$P_k(N, k) = O(N \log N),$$
$$S_k(N, k) = O(Nk).$$

$$Q_k(N, k) \leq O(N^{1-1/k} + F)$$

$$Q_k(N, k) = O(\log N + F)$$

Range trees.

$$P_r(N, 1) = O(N \log N),$$
$$S_r(N, 1) = O(N),$$
$$Q_r(N, 1) = O(\log N + F).$$

$$P_r(N, 2) = O(N \log N),$$
$$S_r(N, 2) = O(N \log N),$$
$$Q_r(N, 2) = O(\log^2 N + F).$$

$$P_r(N, k) = O(N \log^{k-1} N),$$
$$S_r(N, k) = O(N \log^{k-1} N),$$
$$Q_r(N, k) = O(\log^k N + F).$$

K-range.

$$P_o(N, k) = O(N^{1+\epsilon}),$$
$$S_o(N, k) = O(N^{1+\epsilon}),$$
$$Q_o(N, k) = O(\log N + F)$$

$$P_n(N, k) = O(N \log N),$$
$$S_n(N, k) = O(N),$$
$$Q_n(N, k) = O(N),$$

# Estructuras de datos multidimensionales.

# KD-tree

```
Node nearestNeighbor(Node root, Point target, int depth) {

    if (root == null) return null

    if (target[depth % K] < root.point[depth % K]) {
        nextBranch = root.left
        otherBranch = root.right
    } else {
        nextBranch = root.right
        otherBranch = root.left
    }

    Node temp = nearestNeighbor(nextBranch, target, depth + 1)
    Node best = closest(target, temp, root)

    long radiusSquared = distSquared(target, best.point)
    long dist = target[depth % K] - root.point[depth % K]

    if (radiusSquared >= dist * dist) {
        temp = nearestNeighbor(otherBranch, target, depth + 1)
        best = closest(target, temp, best)
    }

    return best
}
```

**1. KD-tree (k-dimensional tree):**

- **Functionality:** Used for efficient spatial indexing and searching in k dimensions, typically 2D (e.g., image data) or 3D (e.g., point clouds).
- **Structure:** Organizes data points by recursively splitting the space along alternating axes (e.g., x-axis then y-axis in 2D). Each node represents a hyperrectangle containing data points and a split plane along a chosen axis.
- **Strengths:**
    - Efficient for k nearest neighbor searches: quickly finds the k data points closest to a query point.
    - Fast for range queries: efficiently retrieves points within a specific region defined by a bounding box.
- **Weaknesses:**
    - "Curse of dimensionality": Performance degrades significantly in high dimensions due to increased hyperrectangle overlap.
    - Complex splits: Choosing the optimal splitting axis in higher dimensions can be challenging.

**2. B-tree:**

- **Functionality:** A self-balancing search tree for sorted data. Commonly used in databases and file systems.
- **Structure:** Each node can have a variable number of children (more than binary search trees) and stores ordered keys and data pointers (or actual data depending on implementation). Internal nodes guide search by comparing keys.
- **Strengths:**
    - Efficient search, insertion, and deletion: All operations are performed in logarithmic time (avg) due to balanced structure.
    - Optimized for disk access: Data is stored in contiguous blocks on disk for faster reads and writes.
- **Weaknesses:**
    - Limited data type: Primarily designed for numeric or string data. Not ideal for complex objects.

*3. B-tree (B-star tree):**

- **Functionality:** A variation of the B-tree with stricter balancing conditions.
- **Structure:** Similar to B-tree, but enforces a stricter fill ratio for nodes (typically at least 2/3 full). This ensures better performance during insertions and deletions.
- **Strengths:**
    - Improved insertion and deletion performance compared to B-tree.
    - Maintains good search efficiency.
- **Weaknesses:**
    - More complex to implement due to stricter balancing requirements.

**4. Quad-tree:**

- **Functionality:** Designed for efficient spatial indexing and searching in 2D space.
- **Structure:** Each internal node has four children, recursively subdividing the space into quadrants (north-west, north-east, south-west, south-east). Data points are associated with the leaf nodes representing the smallest quadrants containing them.
- **Strengths:**
    - Efficient for spatial queries: Well-suited for point location (finding a specific point) and region queries (finding points within a specific area).
    - Simple to implement due to its well-defined structure.
- **Weaknesses:**
    - Limited to 2D data.
    - Not ideal for nearest neighbor searches: Performance degrades as data points become clustered within quadrants.

**5. Oct-tree:**

- **Functionality:** An extension of the quad-tree for 3D space.

- **Structure:** Similar to a quad-tree, but each internal node has eight children, recursively subdividing the space into octants (analogous to quadrants in 3D).
- **Strengths:**
    - Efficient for spatial indexing and searching in 3D data.
    - Shares similar advantages and limitations as quad-trees, but for the third dimension.
- **Weaknesses:**
    - Limited to 3D data.
    - Performance can suffer for nearest neighbor searches with clustered data.

**6. R-tree:**

- **Functionality:** A spatial access method for indexing multi-dimensional data like points, rectangles, or polygons.
- **Structure:** Uses minimum bounding rectangles (MBRs) to represent data objects in the tree. Each node groups data objects spatially and stores their corresponding MBRs.
- **Strengths:**
    - Efficient for spatial search queries: Finding objects within a specific region, finding objects intersecting a given shape, etc.
    - Can also handle nearest neighbor searches in some cases.
- **Weaknesses:**
    - Overlapping MBRs: Overlapping bounding rectangles in the tree can lead to redundant search paths during some queries.
    - Not ideal for high-dimensional data: Performance can degrade due to increased MBR overlap.

*7. R-tree (R-star tree):**

- **Functionality:** An improved version of the R-tree that addresses the issue of MBR overlap.
- **Structure:** Similar to R-tree, but uses a different splitting strategy that aims to minimize area overlap between MBRs in the tree during insertions.