Capítulo 8. Árboles

Mariana Ortíz García[1]
José Gabriel Navarro Favela[2]

DOI: https://doi.org/10.52501/cc.440.08

Resumen

Los árboles son estructuras fundamentales dentro de la teoría de grafos, caracterizadas por ser grafos no dirigidos, conexos y sin ciclos, donde existe un único camino entre cada par de vértices. Estas estructuras permiten representar relaciones jerárquicas de manera eficiente, siendo ampliamente utilizadas en informática para organizar datos, optimizar búsquedas y estructurar sistemas como bases de datos y archivos. Un árbol puede tener una raíz a partir de la cual se derivan nodos en relaciones padre-hijo, definiéndose conceptos clave como nivel, altura y grado, los cuales influyen en la eficiencia de los algoritmos que operan sobre ellos. Además, los árboles tienen aplicaciones en diversos campos como la biología, la lingüística y contextos cotidianos como los torneos deportivos o árboles genealógicos. En términos estructurales, un árbol con n vértices contiene n−1 aristas, lo que garantiza una conexión mínima sin redundancias. También destacan los árboles de expansión, que son subgrafos que conectan todos los vértices de una gráfica, y los árboles de expansión mínima, cuyo objetivo es minimizar el peso total de las conexiones. Para construir estos árboles se utilizan algoritmos como la búsqueda en anchura (BFS), la búsqueda en profundidad (DFS), y métodos como Prim y Kruskal, que siguen estrategias para optimizar recorridos y conexiones. Estas técnicas son esenciales en el análisis de redes, rutas y estructuras complejas. En conjunto, los árboles representan una herramienta clave para modelar, organizar y procesar información en múltiples áreas, desde la informática hasta la inteligencia artificial y la toma de decisiones.

[1] Maestra en Sistemas Computacionales. Docente en Tecnológico Nacional de México. ORCID: https://orcid.org/0009-0003-5415 ; Scopus: 57982741400 ; correo electrónico: mariana.ortiz@itz.edu.mx
[2] Maestro en Sistemas Computacionales. Encargado del Laboratorio de Desarrollo de Software en Tecnológico Nacional de México.

Palabras clave: *árboles, algoritmos de búsqueda, análisis de rutas, árboles de expansión.*

Conceptos básicos y representaciones de árboles

Los árboles constituyen una de las estructuras más fundamentales en la teoría de grafos y tienen múltiples aplicaciones en diversas áreas del conocimiento, desde las ciencias de la computación hasta la biología evolutiva. En términos formales, un árbol es un tipo de grafo no dirigido, conexo y sin ciclos, lo que significa que todos sus vértices están conectados por caminos únicos y que no existen trayectorias cerradas en su estructura (Johnsonbaugh, 1999).

Desde un punto de vista práctico, los árboles ofrecen una forma eficaz de organizar datos. En ciencias de la computación, por ejemplo, son esenciales para la búsqueda eficiente de información, el almacenamiento ordenado de datos, la composición de imágenes digitales y la estructuración de sistemas jerárquicos, como bases de datos o sistemas de archivos. Conceptualmente, un árbol con raíz es aquel en el que uno de los vértices es designado como punto inicial o raíz, y a partir de él se derivan subestructuras llamadas subárboles, con relaciones de padre e hijo entre nodos (Johnsonbaugh, 1999).

El nivel de un vértice se define como la distancia desde la raíz hasta dicho vértice, mientras que la altura del árbol es el mayor nivel presente en la estructura. Estas nociones son clave en el análisis del rendimiento de algoritmos basados en árboles, ya que inciden directamente en la eficiencia de operaciones y en la búsqueda o la inserción.

Más allá de la informática, los árboles tienen aplicaciones en campos como la filogenia, donde se utilizan para representar relaciones de parentesco evolutivo entre especies, o en la lingüística histórica, para reconstruir el origen común de las lenguas. Incluso en contextos cotidianos, como los torneos deportivos de eliminación directa, las estructuras arbóreas modelan el avance de los competidores, donde el equipo ganador aparece en la raíz del árbol.

En suma, los árboles no solo constituyen una herramienta formal de gran elegancia matemática, sino que también poseen una enorme potencia práctica en la modelización de relaciones jerárquicas, optimización de estructuras y análisis de procesos evolutivos.

Un **gráfico de árbol** es una estructura especial dentro de la teoría de grafos que se caracteriza por ser **acíclica y conexa**, es decir, no contiene ciclos y permite que todos sus vértices estén enlazados entre sí mediante caminos únicos. En este sentido, el árbol es la forma más eficiente de conectar todos los puntos de un grafo sin redundancias. Como señala

Johnsonbaugh (1999), los árboles son fundamentales en matemáticas discretas porque establecen la conexión más simple y directa entre nodos dentro de una red.

En un grafo compuesto por **n vértices**, un árbol contará con **exactamente n−1 aristas**, lo cual refleja su estructura mínima de conexión. Además, la cantidad total de árboles distintos que se pueden formar con esos vértices es de $\mathbf{n^{n-2n}}$, lo que evidencia la enorme variedad estructural posible dentro de esta categoría de grafos. Esta propiedad fue formalizada en la fórmula de Cayley, y es ampliamente utilizada en áreas como la computación y la combinatoria (Rosen, 2012).

Desde el punto de vista formal, un **árbol T** se define como un grafo simple que cumple con la condición de que para cada par de vértices v y w, existe un **único camino simple** que los conecta. Además, si se selecciona un vértice específico como punto de inicio, se obtiene un **árbol con raíz**, donde ese nodo central sirve como referencia para establecer relaciones jerárquicas dentro de la estructura (Johnsonbaugh, 1999).

Una de las aplicaciones más prácticas de los árboles es en la **organización de datos dentro de bases de datos**, ya que permiten establecer relaciones jerárquicas entre los elementos, facilitando su clasificación y recuperación. En este contexto, los árboles sirven como modelos para representar estructuras que requieren eficiencia y orden, como los índices en bases de datos relacionales (Johnsonbaugh, 1999).

Más allá de la informática, los árboles también tienen gran utilidad en áreas como la **biología evolutiva**, donde se emplean para representar procesos de **filogenia**. En este caso, los árboles muestran cómo es que diferentes entidades están vinculadas por líneas de descendencia, lo que permite trazar relaciones de parentesco entre especies u otros sistemas como las lenguas humanas. Como afirman Semple y Steel (2003), los árboles filogenéticos constituyen herramientas clave para el estudio de la evolución.

En el campo de la **informática**, el concepto de árbol también se utiliza como una **estructura de datos abstracta** que refleja una jerarquía natural. Cada árbol contiene una **raíz**, que es el nodo principal, y a partir de esta se ramifican **subárboles**, cuyos nodos están conectados a través de relaciones padre-hijo. Esta representación se ha vuelto esencial en algoritmos de búsqueda, estructuras organizativas y almacenamiento de datos eficientes (Knuth, 1998).

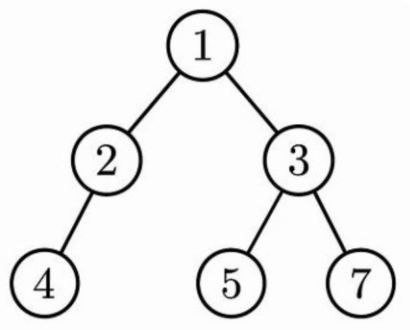
Al representar con una gráfica cualquier torneo deportivo de eliminación se obtiene un árbol, y el equipo ganador será la raíz del árbol.

Dentro de los usos más comunes de los árboles se encuentra el de almacenar un dato de tal modo que su búsqueda sea eficiente, que tenga listas ordenadas de datos y que haya una composición de imágenes digitales.

El nivel de un vértice v es la longitud del camino simple de la raíz a v. La altura de un árbol es el número máximo de nivel que aparece en el árbol. Los hijos de la raíz son los vértices subsecuentes unidos a la raíz. Un vértice padre es un vértice que está por encima de un vértice hijo y más cerca de la raíz.

Partes de un árbol

Figura 1. *Partes de un árbol*



Fuente: elaboración propia.

Si 1 es la raíz,
entonces 2 y 3 son hijos de 1,
3 es padre de 5 y de 7; y 2 es padre de 4.
4, 5 y 7 solamente son hojas.
El grado de un vértice mide lo retirado que esté de la raíz. Por ejemplo:

Vértice	1	2	3	4	5	7
Grado	0	1	1	2	2	2

El grado del árbol es el grado máximo de un vértice, en este caso es 2.

Los organigramas son ejemplos de árboles. A este tipo de árboles se les denomina árboles jerárquicos, se emplean para representar cómo se relacionan lógicamente los distintos registros dentro de una base de datos, donde se muestra el control o administración de una determinada empresa, institución u organización.

Los árboles familiares también son un tipo especial de árbol jerárquico. Se recomienda realizar un árbol genealógico de la familia donde se incluya por lo menos 25 integrantes.

Árboles de expansión

El desafío consiste en encontrar una subgráfica T dentro de una gráfica G, tal que T sea un árbol que incluya todos los vértices de G; a esta estructura se le conoce como árbol de expansión. Una gráfica tiene árbol de expansión siempre y cuando sea conexa, es decir que todos sus vértices estén interconectados (Johnsonbaugh, 1999).

¿Qué es un árbol de expansión mínima? Es un **subconjunto de las aristas de un grafo** que:

1. **Conecta todos los vértices** del grafo (sin ciclos).
2. **Tiene el menor peso total posible**.

Un algoritmo de búsqueda encuentra la única ruta más corta entre dos puntos, utilizando una heurística que ayuda a agilizar el proceso de búsqueda.

Existen varios métodos para encontrar árboles en expansión.

A) **Búsqueda a lo ancho**. Consiste en explorar todos los vértices que se encuentran en un mismo nivel antes de avanzar al siguiente. La búsqueda en anchura es una técnica que permite recorrer de forma sistemática todos los nodos de un grafo. Es especialmente útil en la resolución de problemas de optimización, donde se requiere identificar la mejor alternativa entre varias. El proceso comienza en un vértice inicial v, considerado la raíz, el cual se activa primero. Luego se marcan como visitados todos los vecinos de ese vértice que aún no hayan sido explorados. A continuación, se repite el proceso con los vecinos de los hijos de v, asegurándose de no visitar ningún nodo más de una vez. Como resultado, se genera una estructura sin ciclos, es decir, un árbol.

B) **Búsqueda en profundidad (DFS, por sus siglas en inglés)** es un algoritmo diseñado para recorrer todos los vértices de un grafo finito, siguiendo una estrategia que prioriza explorar lo más profundo posible un camino antes de retroceder. El proceso comienza seleccionando un vértice inicial o raíz, desde el cual se visitan primero sus descendientes directos, luego los hijos de estos y así sucesivamente, hasta llegar a un nodo que no tenga vecinos no visitados.

A diferencia de la búsqueda en anchura, DFS se caracteriza por investigar completamente una rama antes de considerar otras, lo que implica que atraviesa los niveles inferiores del grafo antes de regresar a los superiores. Para gestionar esta secuencia de pasos, generalmente se utiliza una **pila**, ya sea de forma explícita o mediante **recursión** que emplea la pila del sistema, lo que permite regresar a vértices anteriores cuando no se encuentran más caminos disponibles desde el nodo actual.

Cuando ya no es posible avanzar desde un vértice activo, el algoritmo **retrocede** a través de los vértices previamente visitados hasta localizar uno con una conexión pendiente por explorar. Desde allí se reinicia el proceso de avance, repitiendo este patrón de descenso y retroceso hasta que se hayan examinado todos los nodos alcanzables. Este enfoque hace de DFS una herramienta fundamental en diversos algoritmos aplicados sobre grafos, como los de ordenamiento topológico o los que evalúan propiedades como la **conectividad** o la **planitud** de una estructura gráfica.

C) **Búsqueda por menor costo (árbol de expansión mínima).** Es una subgráfica que se construye a partir de una gráfica conexa, que tiene un único camino simple de un vértice a otro y cuya suma de pesos sea mínima.

En el ámbito de la informática, estos algoritmos se emplean ampliamente para **recorrer grafos** y **buscar rutas** entre puntos conocidos como nodos. Su objetivo principal es **encontrar trayectos viables de manera eficiente**, lo que los convierte en herramientas clave para el análisis de redes y estructuras conectadas. Se destacan por su **precisión y buen desempeño**, lo que ha llevado a su uso extendido en múltiples aplicaciones computacionales.

No obstante, en contextos reales como los **sistemas de navegación o enrutamiento de viajes**, suelen ser superados por otros algoritmos más avanzados que permiten **preprocesar el grafo**, optimizando así la velocidad y eficiencia en la resolución del problema (Kurose y Ross, 2020).

Métodos para sacar árboles de expansión mínima

Algoritmo de PRIM

El algoritmo en cuestión construye un **árbol de expansión mínima** incorporando aristas de forma progresiva. En cada paso, selecciona la arista con **menor peso** que no genere un ciclo dentro del subgrafo en desarrollo. Este procedimiento pertenece a la categoría de **algoritmos codiciosos**, ya que en cada iteración toma la mejor decisión disponible en ese momento, sin considerar el impacto de decisiones anteriores (Cormen et al., 2009).

El principio que guía este método puede resumirse como una estrategia de **optimización local**, en la que se elige la mejor opción inmediata —una sola arista con el menor peso posible— con la intención de construir una solución eficiente. Sin embargo, este tipo de optimización **no siempre garantiza** que la solución final sea óptima en términos globales. Si durante el proceso no se encuentra una conexión válida hacia los vértices restantes, el algoritmo retrocede al vértice previo para verificar si existen otras conexiones posibles que aún no se han evaluado (Cormen et al., 2009).

Algoritmo de Kruskal

El algoritmo de Kruskal es un método *greedy* (codicioso) que se utiliza para encontrar un árbol de expansión mínima (MST, por sus siglas en inglés) en un grafo no dirigido, conectado y ponderado.

Consideremos un grafo dirigido y ponderado con N nodos, ninguno de ellos aislado. Sea x el nodo de inicio y un vector D de tamaño N que almacenará, al finalizar el algoritmo, las distancias mínimas desde x a cada uno de los demás nodos. El procedimiento se desarrolla de la siguiente manera:

1. Inicialmente, todas las entradas del vector D se asignan un valor infinitamente grande para representar distancias desconocidas, excepto la distancia desde x a sí mismo, que se establece en cero, dado que no existe distancia alguna para llegar a sí mismo.
2. Se define a = x, donde a es el nodo actual que se está evaluando.
3. Se examinan todos los nodos adyacentes a a que no hayan sido marcados como visitados; a estos los denominamos v_i.

4. Si la distancia almacenada en D para llegar a v_i desde x es mayor que la suma de la distancia desde x a a más la distancia desde a a v_i, entonces el valor en D para v_i se actualiza con esta suma menor.

5. El nodo a se marca como completo o visitado.

6. Se selecciona como nuevo nodo actual aquel con el valor más pequeño en D entre los nodos no visitados (esto se puede gestionar eficazmente utilizando una cola de prioridad). Se repite el proceso desde el paso 3 hasta que todos los nodos estén marcados.

Este método es una implementación típica del algoritmo de Dijkstra, ampliamente utilizado para encontrar rutas óptimas en grafos ponderados (Cormen et al., 2009).

Pasos del algoritmo de Kruskal

1. Ordena todas las aristas del grafo de menor a mayor peso.
2. Inicializa el MST como vacío (sin aristas).
3. Recorre la lista de aristas en orden creciente de peso:
4. Si agregar la arista NO forma un ciclo, agrégala al MST.
5. Si forma un ciclo, descártala.
6. Repite hasta que el MST tenga exactamente (n – 1) aristas, donde n es el número de vértices.

En la tabla 1 se observa la comparación de los diferentes algoritmos en cuanto a rutas cortas y árboles en expansión.

Tabla 1. *Comparación de algoritmos*

Algoritmo	¿Sirve para rutas más cortas?	¿Sirve para árbol de expansión?	Acepta pesos negativos
Bellman-Ford	☑ Sí	✗ No	☑ Sí
Dijkstra	☑ Sí	✗ No	✗ No (sin modificar)
Kruskal/Prim	✗ No	☑ Sí	⚠ Solo pesos positivos

El siguiente es un código de árbol en expansión a lo **ancho** en Java.

```
import java.util.*;
public class BFSExpansionTree {
   // Número de vértices
   private int V;
   // Lista de adyacencia
   private LinkedList<Integer>[] adj;
   // Constructor
   public BFSExpansionTree(int vértices) {
      this.V = vertices;
      adj = new LinkedList[vértices];
      for (int i = 0; i < vértices; ++i)
         adj[i] = new LinkedList<>();
   }

   // Método para agregar aristas (no dirigido)
   void addEdge(int v, int w) {
      adj[v].add(w);
      adj[w].add(v); // Porque es grafo no dirigido
   }

   // Método para construir e imprimir el árbol de expansión por BFS
   void bfsExpansionTree(int start) {
      boolean[] visited = new boolean[V];
      int[] parent = new int[V]; // Para registrar el "padre" de cada nodo en el árbol
      Arrays.fill(parent, -1);
      Queue<Integer> queue = new LinkedList<>();
      visited[start] = true;
      queue.add(start);
      System.out.println("Aristas del árbol de expansión (BFS):");
      while (!queue.isEmpty()) {
```

```
            int current = queue.poll();
            for (int neighbor : adj[current]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    parent[neighbor] = current;
                    queue.add(neighbor);
                    System.out.println(current + " - " + neighbor);
                }
            }
        }
    }

    // Ejemplo de uso
    public static void main(String[] args) {
        BFSExpansionTree g = new BFSExpansionTree(6);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 3);
        g.addEdge(1, 4);
        g.addEdge(2, 5);
        g.bfsExpansionTree(0); // Comienza desde el nodo 0
    }
}
```

Ahora el código de expansión mínima con **Kruskal**

```
import java.útil.*;
// Clase para representar una arista
class Edge implements Comparable<Edge> {
    int src, dest, weight;
    public Edge(int s, int d, int w) {
        src = s;
```

```
        dest = d;
        weight = w; }

    // Para ordenar por peso
    public int compareTo(Edge other) {
        return this.weight - other.weight;
    }
}

// Clase para representar un subconjunto (para Union-Find)
class Subset {
    int parent, rank;
}

public class KruskalMST {
    int vertices;
    List<Edge> edges = new ArrayList<>();

    public KruskalMST(int v) {
        this.vértices = v;
    }

    // Añadir arista
    void addEdge(int src, int dest, int weight) {
        edges.add(new Edge(src, dest, weight));
    }

    // Encontrar conjunto con compresión de caminos
    int find(Subset[] subsets, int i) {
        if (subsets[i].parent != i)
            subsets[i].parent = find(subsets, subsets[i].parent);
```

```
        return subsets[i].parent;
    }

    // Unir dos conjuntos por rango
    void union(Subset[] subsets, int x, int y) {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);
        if (subsets[xroot].rank < subsets[yroot].rank)
            subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
            subsets[yroot].parent = xroot;
        else {
            subsets[yroot].parent = xroot;
            subsets[xroot].rank++;
        }
    }

    // Método principal para construir el MST
    void kruskal() {
        List<Edge> result = new ArrayList<>();
        Collections.sort(edges); // Ordenar aristas por peso
        Subset[] subsets = new Subset[vertices];
        for (int i = 0; i < vertices; i++) {
            subsets[i] = new Subset();
            subsets[i].parent = i;
            subsets[i].rank = 0;
        }
        for (Edge edge : edges) {
            int x = find(subsets, edge.src);
            int y = find(subsets, edge.dest);
            if (x != y) {
```

```
                result.add(edge);
                unión(subsets, x, y);
            }
        }
        System.out.println("Aristas del Árbol de Expansión Mínima:");
        int totalWeight = 0;
        for (Edge e : result) {
            System.out.println(e.src + " - " + e.dest + " : " + e.weight);
            totalWeight += e.weight;
        }
        System.out.println("Peso total del MST: " + totalWeight);
    }
    // Ejemplo de uso
    public static void main(String[] args) {
        KruskalMST g = new KruskalMST(4);
        g.addEdge(0, 1, 10);
        g.addEdge(0, 2, 6);
        g.addEdge(0, 3, 5);
        g.addEdge(1, 3, 15);
        g.addEdge(2, 3, 4);
        g.kruskal();
    }
}
```

Aplicaciones de los árboles

Los árboles, como estructuras de datos jerárquicas, poseen una amplia aplicación en contextos reales debido a su capacidad para representar relaciones de dependencia, clasificación y organización escalonada de información. Su utilidad no es meramente teórica; por el contrario, constituyen uno de los fundamentos operativos de múltiples sistemas tecnológicos contemporáneos (Cormen et al., 2022; Goodrich et al., 2014).

En el ámbito de los sistemas de archivos, por ejemplo, la organización de carpetas y subcarpetas en sistemas operativos como Windows, macOS o Linux responde a una estructura arbórea. Cada directorio funciona como un nodo que puede contener otros nodos (subdirectorios o archivos), formando una jerarquía que facilita la localización eficiente de información. Esta estructura permite búsquedas organizadas y rápidas, especialmente cuando se implementan variantes como árboles balanceados para optimizar tiempos de acceso.

En el campo de las bases de datos, los árboles son esenciales para indexar información. Estructuras como el *B-tree* y sus variantes permiten que sistemas gestores de bases de datos encuentren registros en grandes volúmenes de información en tiempos logarítmicos. Sin estos mecanismos, consultas en bancos, hospitales o plataformas de comercio electrónico serían considerablemente más lentas. La eficiencia en la recuperación de datos se traduce directamente en mejor desempeño organizacional y experiencia del usuario (Cormen et al., 2022).

Otra aplicación significativa se encuentra en los motores de búsqueda en internet. Algoritmos como PageRank, desarrollado por Larry Page, utilizan estructuras relacionadas con grafos y árboles para jerarquizar páginas web según su relevancia. Aunque conceptualmente se apoyan en grafos, la organización jerárquica de resultados y la estructuración del contenido digital mantienen principios análogos a los árboles, particularmente en la clasificación y recuperación de información.

En el ámbito de la inteligencia artificial, los árboles de decisión constituyen herramientas fundamentales para el aprendizaje automático y la toma de decisiones automatizada. Modelos como el C4.5 permiten clasificar datos y predecir comportamientos a partir de reglas jerarquizadas. Estos modelos se emplean en diagnóstico médico, evaluación crediticia, detección de fraudes y sistemas de recomendación. La estructura arbórea facilita la interpretación de resultados, lo que aporta transparencia en entornos donde la explicabilidad es crucial.

Asimismo, en el desarrollo de compiladores y lenguajes de programación, los árboles sintácticos abstractos representan la estructura gramatical del código fuente. Cada instrucción se descompone en nodos que reflejan operaciones y operandos, permitiendo que el sistema interprete y ejecute correctamente los programas (Goodrich et al., 2014).

Desde una perspectiva más amplia, los árboles modelan fenómenos sociales y naturales: genealogías familiares, estructuras organizacionales, taxonomías biológicas y

esquemas curriculares pueden representarse mediante estructuras arbóreas. Su fortaleza radica en representar relaciones jerárquicas sin ambigüedad y con eficiencia computacional.

En síntesis, los árboles no son únicamente una abstracción matemática, sino una herramienta estructural indispensable para organizar, buscar, clasificar y procesar información en múltiples ámbitos de la vida cotidiana. Su aplicación transversal confirma que la teoría algorítmica se materializa en soluciones concretas que sostienen la infraestructura digital contemporánea.

Referencias

Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2009). *Introduction to algorithms* (3.ª ed.). MIT Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2022). *Introduction to algorithms* (4.ª ed.). MIT Press.

Goodrich, M. T., Tamassia, R., y Goldwasser, M. H. (2014). *Data structures and algorithms in Python*. Wiley.

Johnsonbaugh, R. (1999). *Matemáticas discretas* (4.ª ed.). Editorial Pearson.

Knuth, D. E. (1998). *The art of computer programming, volume 1: fundamental algorithms* (3.ª ed.). Addison-Wesley.

Kurose, J. F., y Ross, K. W. (2020). C*omputer networking: a top-down approach* (8th ed.). Pearson.

Rosen, K. H. (2012). *Discrete mathematics and its applications* (7.ª ed.). McGraw-Hill.

Semple, C., y Steel, M. (2003). *Phylogenetics*. Oxford University Press.

Stallings, W. (2017). *Computer organization and architecture: designing for performance*. Pearson.

Tanenbaum, A. S. (2016). *Structured computer organization* (6.ª ed.). Pearson.

Tanenbaum, A. S., y Austin, T. (2013). *Structured computer organization*. Pearson.

Tanenbaum, A. S., y Bos, H. (2014). *Modern operating systems*. Pearson.

UNAD. (s.f.). *Sistemas numéricos*. https://www.unadmexico.mx/LITE_36/_Un_151_OperacionesAritmeticasBasicas/escenas/2_Inicio_1.html#:~:text=B%C3%A1sicamente%20los%