**UNIVERSIDAD AMERICANA**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algoritmo y Estructura de Datos**

Investigación de arboles

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Estudiantes:**

Allan Raúl Valentín Acuña Otero **CIF**: 23020611

Fernando Gabriel Gómez Gazo **Cif:** 23012349

Guillermo José Vega Vega **Cif:** 24010412

Gabriela Michelle Guerrero Paiz **Cif:** 24010347

Steven Leonel Sequeira Reyes **Cif:** 24010446

**Docente:**

Jose Alejandro Durán

**Carreras:**

Ingeniería en Sistemas de la Información

**Fecha:**

11 de Junio del 2025

**Managua, Nicaragua**

Índice

[**1. Introducción general a árboles 3**](#_6tc7nqoy0z6s)

[**2. Árboles ternarios 4**](#_wvzeu3xd388m)

[Árbol de búsqueda ternario (Ternary Search Tree, TST) 4](#_5qipl8m8igks)

[Montículo ternario (ternary heap) 4](#_ihd9v6d8wle4)

[**3. Árboles n-arios (genéricos) 5**](#_4nhbsmuzq3nk)

[Árboles B (B-trees) y B+ trees 5](#_qtqjokmyecse)

[Tries (árboles de prefijos) 6](#_7q8gwqpcg7f4)

[**4. Montículos (Heaps) 7**](#_28o0nnbizw28)

[Insertar 7](#_y4088q2f17k3)

[Extraer raíz 8](#_erpa5t8grsk9)

[Construcción de montículo (build-heap) 8](#_h4q4hz4qnlz5)

[**5. Métodos de ordenación relacionados 9**](#_v64abe8x44ev)

[Heapsort 9](#_dyhba3jpy8et)

[Heapsort ternario 9](#_n7bebd6a6ksu)

[Smoothsort 10](#_tmt4p1z9ta9y)

[Ordenación con árboles binarios (Tree Sort) 10](#_irkny0egbom7)

[**6. Recorridos en árboles (tree traversal) 11**](#_mpe97itiy79w)

[Recorridos en profundidad (DFS) 11](#_4wvuo7upxnr7)

[Recorrido en anchura (BFS o level-order) 11](#_qcdyanr0iadv)

[**7. Conexiones entre temas 12**](#_h9e36rj8hsdg)

[**8. Conclusión y futuras líneas 13**](#_i9q3epfckn0v)

[**Referencias 15**](#_up4ibwawv72q)

Árboles

# 1. Introducción general a árboles

Un **árbol** es una estructura de datos jerárquica no lineal compuesta por *nodos* conectados por aristas. Cada nodo almacena un valor y puede tener cero o más **hijos** (descendientes), mientras que, salvo la *raíz*, cada nodo tiene un único **padre** (antecesor inmediato). El **nodo raíz** es el nodo superior del árbol (el primer elemento) y no tiene padre. Un **nodo hoja** o terminal es aquel que no tiene hijos (grado cero). El **nivel** (profundidad) de un nodo es la longitud del camino desde la raíz hasta ese nodo, y la **altura** del árbol es la longitud máxima de un camino de la raíz a cualquier hoja. El **grado** de un nodo es el número de hijos directos que tiene, por lo que las hojas tienen grado 0.

Los árboles son muy útiles para representar datos con relaciones jerárquicas o dependientes, como genealogías, estructuras de archivos o jerarquías organizacionales. Además se emplean en indexación de bases de datos (por ejemplo con **B-trees** y **B+ trees**), en **colas de prioridad** (montículos) y en algoritmos de búsqueda eficientes (como árboles balanceados). Así, estructuras tipo árbol facilitan operaciones rápidas de inserción, eliminación y búsqueda: por ejemplo, un árbol de búsqueda balanceado permite localizar un elemento en tiempo *O*(log *n*) en el peor caso. También se usan en compiladores (árboles de expresión), sistemas de archivos, procesamiento de cadenas (tries para prefijos) y en muchos algoritmos de ordenación y estructura de datos avanzados.

**Ejemplo sencillo: clase para representar un árbol genérico**

*# Ejemplo sencillo: clase para representar un árbol genérico*

class TreeNode:

def \_\_init\_\_(self, value, children=None):

self.value = value

self.children = children if children is not None else []

*# Creación de un árbol con un nodo raíz y dos hijos*

root = TreeNode(1, [TreeNode(2), TreeNode(3, [TreeNode(4)])])

*# Función para imprimir el árbol de forma recursiva*

def print\_tree(node, level=0):

print(' ' \* level + *str*(node.value))

for child in node.children:

print\_tree(child, level + 2)

*# Imprimir el árbol*

print\_tree(root)

# 2. Árboles ternarios

Un **árbol ternario** es un árbol en el que cada nodo puede tener hasta tres hijos. Por ejemplo, con 2 nodos hay tres árboles ternarios distintos según se ubique el hijo (izquierda, centro o derecha). Esta estructura generaliza al árbol binario (que admite hasta 2 hijos) permitiendo un mayor número de ramas por nivel. La propiedad clave es similar: la forma precisa de ordenar o comparar depende del tipo de árbol.

**Algunas variantes importantes son:**

## Árbol de búsqueda ternario (Ternary Search Tree, TST)

Cada nodo almacena un elemento y tiene hasta tres punteros, típicamente llamados izquierdo, medio y derecho. Se usa para almacenar cadenas de texto de forma eficiente, ofreciendo búsquedas similares a *tries* pero con un factor de ramificación 3.

## Montículo ternario (ternary heap)

Es un montículo donde cada nodo tiene hasta 3 hijos. Al igual que el montículo binario, satisface la propiedad de heap (por ejemplo, cada padre ≥ hijos) pero el índice de cada hijo se calcula con base 3. Un montículo ternario completo se puede implementar en un arreglo analogamente al caso binario, con fórmulas de índice padre=(i-1)//3 y tres hijos en 3\*i+1, 3\*i+2, 3\*i+3. Según la definición, “cada nodo *up to* tres hijos”.

Desde el punto de vista combinatorio, el número de árboles ternarios con *n* nodos sigue fórmulas de conteo similares a los números de Catalán generalizados. En la práctica, los árboles ternarios se usan cuando hay tres opciones de ramificación principales: por ejemplo, en algoritmos de búsqueda de cadenas con tres comparaciones, o en modelos de decisión trivalente. Los montículos ternarios pueden mejorar ligeramente el factor constante en la construcción de heapsort con base 3, pero siguen ofreciendo complejidad *O(n log n)* en promedio.

# 3. Árboles n-arios (genéricos)

Un **árbol k-ario (o n-ario)** es un árbol en el que cada nodo puede tener hasta *k* hijos, donde *k* es un número fijo. En teoría de grafos, “un árbol k-ario es un árbol arraigado en el que cada nodo no tiene más de *k* hijos”. El caso *k*=2 corresponde al árbol binario. Los árboles k-arios completos tienen todos los nodos con exactamente *k* hijos (hasta el penúltimo nivel). Por ejemplo, el número máximo de nodos en un árbol k-ario de altura *h* es (kh+1−1)/(k−1)(k^{h+1}-1)/(k-1).

**Ejemplos célebres de árboles n-arios incluyen:**

## Árboles B (B-trees) y B+ trees

Son árboles de búsqueda balanceados multiway (genéricos) diseñados para bases de datos y sistemas de archivos. Tienen *orden* *M* (cada nodo puede tener hasta *M* hijos) y mantienen todas las hojas al mismo nivel, garantizando profundidad *O*(log *n*). Un árbol B+ almacena todas las claves en las hojas y enlaza secuencialmente estas hojas para búsquedas en rango. Son estructuras índice multinivel eficientes para acceso directo y ordenación.

## Tries (árboles de prefijos)

Son árboles de tipo *n*-ario donde cada arista representa un símbolo de un alfabeto. Las claves (cadenas) se almacenan a lo largo de las hojas, y las búsquedas recorren el árbol letra a letra. Gracias a esto, “la forma en la que se almacena la información permite hacer búsquedas eficientes de cadenas que comparten prefijos”. Un trie genérico puede considerarse un *árbol n-ario* con *n* igual al tamaño del alfabeto, optimizado para operaciones con cadenas.

Los árboles n-arios proveen un almacenamiento jerárquico y flexible de datos. Implementan índices de múltiples ramas (por ejemplo, árboles de decisión con múltiples salidas) y ayudan a reducir la altura del árbol comparado con árboles binarios de similar número de nodos. Además, permiten diseños eficientes en espacio para claves grandes (como en tries) y optimización de I/O en bases de datos al agrupar múltiples claves en cada nodo (B-trees).

**Ejemplo: definición simplificada de un trie (árbol de prefijos) en Python**

class TrieNode:

def \_\_init\_\_(self):

self.children = {} *# diccionario de hijos {carácter: nodoTrie}*

self.is\_end = False

class Trie:

def \_\_init\_\_(self):

self.root = TrieNode()

def insert(self, word):

nodo = self.root

for ch in word:

nodo = nodo.children.setdefault(ch, TrieNode())

nodo.is\_end = True

def search(self, word):

nodo = self.root

for ch in word:

if ch not in nodo.children:

return False

nodo = nodo.children[ch]

return nodo.is\_end

*# Insertar y buscar en el trie*

trie = Trie()

trie.insert("hola")

print(trie.search("hola")) *# True*

print(trie.search("ole")) *# False*

# 4. Montículos (Heaps)

Un **montículo** (o *heap* en inglés) es un árbol binario casi completo que satisface la propiedad de montículo: en un *montículo máximo*, cada nodo padre es mayor o igual que sus hijos; en un *montículo mínimo*, el padre es menor o igual que sus hijos. Al ser casi completo (todos los niveles llenos salvo posiblemente el último, que se llena de izquierda a derecha), un montículo se puede almacenar eficientemente en un arreglo lineal sin punteros. En este arreglo, para un índice *i* el padre se calcula como ⌊(*i*−1)/2⌋ y los hijos como 2*i*+1 y 2*i*+2.

El mayor (o menor) elemento siempre está en la raíz. Por ejemplo, “en un montículo de prioridad, el mayor elemento está siempre en el nodo raíz… por esta razón, los montículos son útiles para implementar colas de prioridad”. Las operaciones básicas son:

## Insertar

Añadir un nuevo elemento al final del arreglo (última hoja) y “flotar” este nodo comparándolo repetidamente con su padre y, si es mayor (en max-heap), intercambiándolo hasta restaurar la propiedad de montículo.

## Extraer raíz

Quitar el elemento en la raíz (máximo/min) y reemplazarla por la última hoja del árbol, luego “hundir” ese elemento comparándolo con sus hijos (y eligiendo el hijo mayor) para que descienda hasta la posición correcta.

## Construcción de montículo (build-heap)

Dado un arreglo, se puede convertir en montículo en tiempo *O*(*n*) aplicando *sift-down* desde la mitad hacia el inicio.

Por su estructura completa, todas estas operaciones tienen complejidad *O*(log *n*) en tiempo en el peor caso (ya que la altura es logarítmica). Además, se puede acceder al elemento máximo o mínimo en *O*(1) (la raíz) y el tamaño en *O*(1) observando el número de elementos. Gracias a su implementación con arreglos, un montículo es compacto y evita la sobrecarga de punteros.

Los montículos tienen variantes avanzadas: por ejemplo, los **montículos binomiales** o **montículos de Fibonacci** soportan operaciones de disminución de clave más rápidas o unificación de montículos (merge) más eficiente, útiles en algoritmos avanzados (como Dijkstra con decrease-key en *O*(1) amortizado). No obstante, en la práctica el montículo binario estándar es muy utilizado (por ejemplo, la biblioteca estándar de Python incluye heapq para heaps mínimos).

**Ejemplo: implementación básica de un montículo máximo usando un arreglo**

def heapify(arr):

n = len(arr)

*# Convertir arr en un montículo máximo*

for i in range(n//2 - 1, -1, -1):

siftdown(arr, i, n)

def siftdown(arr, i, n):

largest = i

izq = 2\*i + 1

der = 2\*i + 2

if izq < n and arr[izq] > arr[largest]:

largest = izq

if der < n and arr[der] > arr[largest]:

largest = der

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

siftdown(arr, largest, n)

*# Uso: construir montículo máximo y extraer el máximo*

arr = [3, 1, 6, 5, 2, 4]

heapify(arr)

print(arr) *# arr ahora reordenado como montículo (p.ej. [6,5,4,3,2,1])*

# 5. Métodos de ordenación relacionados

**Los árboles y montículos se emplean en varios algoritmos de ordenación:**

## Heapsort

Utiliza un montículo para ordenar un arreglo. Primero se construye un *max-heap* con todos los elementos (*O(n)*), luego se extrae sucesivamente la raíz (máximo) intercambiándola con la última hoja y reordenando (*siftdown*), obteniendo el arreglo ordenado. Heapsort es *no estable* y tiene complejidad Θ(n log n) en todos los casos. Como lo resume la definición: “el algoritmo consiste en almacenar todos los elementos en un montículo y luego extraer el nodo raíz repetidamente obteniendo el conjunto ordenado”.

## Heapsort ternario

Variante que usa un *montículo ternario* en lugar de binario. El procedimiento es análogo, pero cada nodo tiene 3 hijos. Esto cambia ligeramente los pasos de **heapify** y la altura del heap, pero la complejidad sigue siendo *O(n log n)* (solo cambia la base del logaritmo). En la práctica, la base 3 reduce un poco las comparaciones por nivel a costa de más hijos que verificar.

## Smoothsort

Algoritmo de ordenación adaptativo creado por Dijkstra como variante de heapsort. Usa montones de **números de Leonardo** en lugar de una sola estructura binaria. Como resultado, Smoothsort mantiene Θ(n log n) en el peor caso, pero puede alcanzar *O(n)* cuando la entrada está casi ordenada. Según la descripción, “proporciona una mejora de su rendimiento de O(n log n) a O(n) si la entrada tiene algún nivel de ordenación”.

## Ordenación con árboles binarios (Tree Sort)

Inserción de todos los elementos en un árbol binario de búsqueda y luego recorrido inorden para extraerlos en orden ascendente. Esto da un coste promedio *O(n log n)*, pero puede ser *O(n²)* en el peor caso si el árbol se desbalancea. Sin embargo, usando árboles equilibrados (AVL, Rojo-Negro) se garantiza *O(n log n)* siempre.

Por ejemplo, en Python puede implementarse heapsort usando heapq (montículo mínimo), invirtiendo el signo para máximo:

import heapq

def heapsort(arr):

h = []

for x in arr:

heapq.heappush(h, x) *# O(log n)*

return [heapq.heappop(h) for \_ in range(len(h))] *# O(n log n)*

print(heapsort([4,1,7,3,8,5]))

# 6. Recorridos en árboles (tree traversal)

**Los recorridos sistemáticos de árboles permiten visitar todos los nodos una vez. Hay dos categorías principales:**

## Recorridos en profundidad (DFS)

Exploran tan lejos como sea posible por cada rama antes de retroceder. En árboles binarios se distinguen tres ordenamientos clásicos: *preorden* (raíz, luego subárbol izquierdo, luego subárbol derecho), *inorden* (izq, raíz, der) y *postorden* (izq, der, raíz). La diferencia radica en cuándo se procesa la raíz. En general, se implementan recursivamente:

def preorder(nodo):

if nodo:

print(nodo.value) *# visita la raíz*

preorder(nodo.left) *# recorre subárbol izquierdo*

preorder(nodo.right) *# recorre subárbol derecho*

def inorder(nodo):

if nodo:

inorder(nodo.left)

print(nodo.value) *# raíz entre hijos*

inorder(nodo.right)

def postorder(nodo):

if nodo:

postorder(nodo.left)

postorder(nodo.right)

print(nodo.value) *# visita la raíz al final*

## Recorrido en anchura (BFS o level-order)

Visita por niveles, de arriba hacia abajo. Se procesan todos los nodos de un nivel antes de pasar al siguiente. Suele implementarse con una cola:ue

from collections import deque

class Nodo:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

def bfs(root):

q = deque([root])

while q:

nodo = q.popleft()

print(nodo.value) *# visita el nodo actual*

if nodo.left:

q.append(nodo.left)

if nodo.right:

q.append(nodo.right)

*# Ejemplo de uso*

if \_\_name\_\_ == "\_\_main\_\_":

a = Nodo('A')

b = Nodo('B')

c = Nodo('C')

d = Nodo('D')

e = Nodo('E')

f = Nodo('F')

a.left = b

a.right = c

b.left = d

b.right = e

c.right = f

bfs(a)

En un **árbol genérico** n-ario, los recorridos análogos dependen de las operaciones *pre*, *in* y *post* definidas. Por ejemplo, para profundidad-primero se haría *preorden*, luego para cada hijo recursivamente (izq a der) *inorden* intercalando con alguna acción, etc. Los recorridos se usan en múltiples aplicaciones: inorden en árboles de búsqueda binaria produce la lista ordenada; postorden es útil para liberar memoria o calcular valores de los hijos antes de la raíz; preorden sirve para copiar la estructura del árbol; y BFS se emplea en algoritmos de grafos para encontrar el nodo más cercano.

# 7. Conexiones entre temas

Los temas tratados están íntimamente relacionados. Un montículo es en esencia un árbol binario completo con propiedad especial; por tanto, los algoritmos sobre heaps aprovechan los recorridos en árboles para *flotar* o *hundir* nodos, y el heapsort utiliza este recorrido para ordenar. Los árboles de búsqueda (binarios o B-trees) aplican recorridos *inorden* para obtener resultados ordenados. Por otro lado, las estructuras de ordenación basadas en árboles (como tree sort) muestran que insertar en un árbol de búsqueda y recorrerlo es equivalente a ordenar. La evaluación de complejidades también mezcla los conceptos: operaciones de inserción/búsqueda en un árbol balanceado son *O*(log *n*), similar a las operaciones en un montículo, mientras que los recorridos completos son *O*(n).

En síntesis, todos estos conceptos exploran formas de organizar datos en jerarquías; así, un árbol binario de búsqueda promueve búsquedas rápidas, los montículos aceleran colas de prioridad y el uso de múltiples hijos (n-ario) optimiza espacio e I/O. La elección entre ellos depende del problema: árboles balanceados o B-trees para índices de bases de datos, montículos para algoritmos voraces (heapsort, Dijkstra), tries para colecciones de cadenas, etc. Las complejidades combinan estos elementos: por ejemplo, un heapsort es *O(n log n)* gracias al recorrido descendente por niveles del heap, y un tree sort balanceado es *O(n log n)* promedio gracias al equilibrio del árbol.

# 8. Conclusión y futuras líneas

Los árboles y montículos son estructuras fundamentales que ofrecen potentes herramientas para organizar y procesar datos jerárquicos y ordenados. A modo comparativo, los árboles binarios son sencillos y rápidos para muchos problemas, mientras que los árboles n-arios (B-trees, tries) sacrifican simplicidad por escalabilidad en I/O o datos voluminosos. Los montículos facilitan la implementación de colas de prioridad con complejidad logarítmica. En la práctica, todas estas estructuras se emplean junto con algoritmos de ordenación y búsqueda para obtener sistemas eficientes.

En líneas avanzadas, se investigan árboles autobalanceados más sofisticados (AVL, árboles rojo-negro, árboles 2-3-4) y variantes persistentes o concurrentes que permiten mantener versiones históricas o acceso paralelo. También es relevante el estudio de árboles en nuevas memorias (NVRAM) o en contextos distribuidos (árboles Paxos en sistemas distribuidos). Otro campo activo es la optimización de recorridos y ordenaciones adaptativas (como smoothsort o splay trees) para casos particulares de entrada. En resumen, los conceptos vistos aquí son la base para explorar numerosas extensiones algorítmicas en estructuras de datos y bases de datos.

# Referencias

Github. <https://github.com/allnrvao/Expo_Arboles>

Presentación.<https://www.canva.com/design/DAGTRWWVnPc/PjOQkK4pNVqQief-t-6s0Q/edit?utm_content=DAGTRWWVnPc&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton>

ApInEm SEO Web. (s. f.). *Árboles en programación: ¿Para qué sirven?*. ApInEm.<https://www.apinem.com/arboles-programacion/>

Heapsort. (s. f.). *Wikipedia*.<https://es.wikipedia.org/wiki/Heapsort>

Montículo (informática). (s. f.). *Wikipedia*.<https://es.wikipedia.org/wiki/Mont%C3%ADculo_%28inform%C3%A1tica%29>

Petit, J., & Roura, S. (s. f.). *Árboles ternarios (problema P80399)* [Página web]. Jutge.org.<https://jutge.org/problems/P80399_es>

Recorrido de árboles. (s. f.). *Wikipedia*. [https://es.wikipedia.org/wiki/Recorrido\_de\_árboles](https://es.wikipedia.org/wiki/Recorrido_de_%C3%A1rboles)

Smoothsort. (s. f.). *Numerentur*.<https://numerentur.org/ordenacion-de-vectores-smooth-sort/>

technobcn. (2012, agosto 19). *Construyendo un Árbol B+ (B+Tree)* [Entrada en blog]. WordPress.com.<https://technobcn.wordpress.com/2012/08/19/construyendo-un-arbol-b-btree/>

Trie. (s. f.). *Wikipedia*.<https://es.wikipedia.org/wiki/Trie>

Use The Index, Luke. (s. f.). *El árbol de búsqueda equilibrado (B-Tree) en las bases de datos SQL*. [https://use-the-index-luke.com/es/sql/índice-anatomía/b-tree](https://use-the-index-luke.com/es/sql/%C3%ADndice-anatom%C3%ADa/b-tree)

Wikipedia. (s. f.-a). *Árbol B+*. [https://es.wikipedia.org/wiki/Árbol\_B+](https://es.wikipedia.org/wiki/%C3%81rbol_B+)

Wikipedia. (s. f.-b). *Árbol k-ario*. [https://es.wikipedia.org/wiki/Árbol\_k-ario](https://es.wikipedia.org/wiki/%C3%81rbol_k-ario)