UNIVERSIDAD AMERICANA
Facultad de Ingeniería y Arquitectura

Algoritmos y Estructura de Datos

Análisis de Eficiencia del Algoritmo de Ordenamiento Radix Sort y del Algoritmo de Búsqueda Hashing

---

Estudiantes:

- Emmanuel Leonardo Aguilar Novoa
- Diego Armando Urbina Aviles
- Julio Cesar Delgadillo Pineda

Docente:

- Silvia Gigdalia Ticay López

Managua 7 Julio, 2025

Introducción

En la actual era de los datos masivos, organizaciones de todos los sectores finanzas, salud, comercio electrónico, redes sociales generan volúmenes de información que crecen de manera exponencial. Esta explosión de datos plantea desafíos significativos para los ingenieros de software, quienes deben diseñar sistemas capaces de *almacenar, clasificar y recuperar* registros con la menor latencia posible.

Entre los enfoques clásicos para resolver estos problemas destacan la transformación de claves (*hashing*) y los algoritmos de ordenamiento, los cuales proporcionan métodos efectivos para optimizar operaciones de acceso y de reordenamiento. El hashing permite transformar claves de longitud variable en índices fijos, habilitando búsquedas *casi* en tiempo constante y reduciendo el impacto de estructuras de datos más complejas. Por su parte, Radix Sort ofrece un rendimiento lineal bajo condiciones favorables, desafiando a algoritmos comparables como "*QuickSort* o *MergeSort"* en contextos donde las claves comparten longitud y dominio.

A pesar de sus ventajas teóricas, las implementaciones de hashing y Radix Sort están sujetas a factores que pueden degradar su desempeño, tales como la frecuencia de colisiones o la elección de la base (*radix*). Asimismo, la literatura muestra resultados divergentes respecto a su eficiencia en entornos reales, donde el hardware, los lenguajes de programación y la naturaleza de los datos influyen en la ejecución.

En este trabajo se propone un análisis integral tanto teórico como empírico que ponga a prueba diversas estrategias de hashing (división, multiplicación, dispersión universal) y dos variantes de Radix Sort (LSD y MSD). El propósito es evaluar su escalabilidad y ofrecer lineamientos prácticos para su adopción en sistemas que manejan grandes volúmenes de información.

Planteamiento del problema

Las aplicaciones modernas como los motores de búsqueda, las plataformas de *streaming* y los sistemas de recomendación exigen tiempos de respuesta que se miden en milisegundos. En tales escenarios, una mala elección de la estructura de datos o del algoritmo de ordenamiento puede traducirse en cuellos de botella que afectan la experiencia del usuario y los costos operativos.

Con frecuencia, la literatura aborda el *hashing* y Radix Sort por separado; sin embargo, existe un vacío en la evaluación comparativa que considere sus interacciones y su comportamiento al variar el tamaño y la distribución de los datos. Surge entonces la siguiente cuestión central:

¿Bajo qué condiciones la transformación de claves mediante hashing y el algoritmo Radix Sort proporcionan el mejor balance entre tiempo de ejecución y consumo de recursos, y cómo se ven afectados por factores como las colisiones y la selección de la base?

Responder a esta pregunta es crucial para orientar la toma de decisiones en el diseño de bases de datos, "*caché"* en memoria y sistemas de análisis en tiempo real. La investigación busca cuantificar la eficiencia de ambos métodos y explicar las causas de cualquier divergencia con respecto a las predicciones teóricas.

Objetivo General

Analizar la eficiencia computacional y la aplicabilidad práctica de la transformación de claves mediante *hashing* y del algoritmo de ordenamiento *Radix Sort* mediante su implementación, evaluación y comparación.

Objetivos específicos

- Analizar el rendimiento de los algoritmos de hashing (como técnica de búsqueda) y Radix Sort (como técnica de ordenamiento), evaluando su eficiencia en tiempo de ejecución y uso de memoria ante diferentes tamaños y distribuciones de datos.
- Implementar funciones del método de búsqueda hash y del método de ordenamiento Radix Sort en sus variantes LSD (Least Significant Digit) y MSD (Most Significant Digit).
- Evaluar experimentalmente el consumo de tiempo y memoria de ambas técnicas con tamaños de entrada crecientes..
- Comparar los resultados obtenidos con los análisis a priori y discutir la influencia de factores como colisiones y la base (*radix*) elegida.
- Proponer lineamientos de uso para sistemas que requieran búsqueda rápida y ordenamiento estable.

Metodología

Diseño de la investigación

El diseño que lleva esta investigacion es experimental, pues se manipulan estructuras de datos y algoritmos para medir su rendimiento mediante pruebas controladas

Tipo de investigación

La investigación aplicada busca resolver problemas concretos mediante la aplicación práctica del conocimiento, mientras que la investigación experimental se basa en la manipulación controlada de variables para observar sus efectos en determinadas condiciones (Hernández, Fernández & Baptista, 2014).

En este contexto, el presente estudio es de tipo experimental y aplicado, ya que no se limita al análisis teórico, sino que implementa y evalúa algoritmos fundamentales (hashing y Radix Sort) para medir su comportamiento en condiciones reales. Se enfoca en resolver un problema práctico: mejorar el rendimiento en almacenamiento y ordenamiento de datos en sistemas que requieren alta eficiencia.

Enfoque metodológico:

El enfoque cuantitativo se caracteriza por el uso de la recolección y el análisis de datos numéricos con el fin de establecer patrones, relaciones o generalizaciones sobre los fenómenos estudiados, empleando métodos estadísticos y objetivos (Hernández, Fernández & Baptista, 2014).

La presente investigación adopta un enfoque cuantitativo, ya que se basa en la medición de datos concretos como el tiempo de ejecución y el consumo de memoria durante

la implementación de algoritmos. Este enfoque permite evaluar objetivamente el rendimiento de *hashing* y *Radix Sort*, identificar cuellos de botella y comparar sus resultados con predicciones teóricas bajo diferentes condiciones experimentales.

Diseño experimental:

- Se desarrollarán implementaciones propias de las siguientes técnicas:
    - Hashing: con funciones basadas en método de división, multiplicación y dispersión universal, aplicadas sobre claves numéricas.
    - Radix Sort: en sus variantes LSD (Least Significant Digit) y MSD (Most Significant Digit).
- Las pruebas se realizarán en un entorno controlado, asegurando condiciones constantes (lenguaje, compilador, hardware, SO).
- Se generarán conjuntos de datos aleatorios con diferentes tamaños (10^3, 10^4, 10^5, 10^6) y distribución (uniforme y sesgada), simulando situaciones que se dan en estructuras de bases de datos o buffers de caché.
- Se repetirá cada experimento 10 veces para obtener promedios y reducir el sesgo estadístico.

Alcance de la investigación:

La investigación tiene un alcance **descriptivo-explicativo**:

- **Descriptivo**: porque documenta cómo se comportan los algoritmos de hashing y Radix Sort en diferentes condiciones.

- **Explicativo**: porque interpreta los resultados obtenidos, conectándolos con aspectos como la cantidad de colisiones, estabilidad del ordenamiento y la influencia de la base elegida (radix).

Variables de estudio:

- **Independientes:**
  - Tipo de función hash implementada.
  - Variante de Radix Sort utilizada.
  - Tamaño y distribución del conjunto de datos.
- **Dependientes:**
  - Tiempo de ejecución (ms).
  - Consumo de memoria (MB).
  - Tasa de colisiones (hashing).
  - Estabilidad del ordenamiento (Radix Sort).
  - Número de accesos/reservas de memoria (si aplica).

Instrumentos y técnicas de recolección de datos:

- Cronometrado mediante funciones como `time()` en Python.
- Monitoreo del uso de memoria mediante bibliotecas (`psutil`, `tracemalloc`, etc.).
- Registros en tablas comparativas para análisis estadístico descriptivo.
- Representación gráfica de los resultados (gráficos de barras o líneas) para observar tendencias de eficiencia y escalabilidad.

Justificación técnica:

Hashing y Radix Sort son temas clave en la asignatura porque representan estrategias eficientes de búsqueda y ordenamiento que pueden ser preferibles frente a estructuras más pesadas como árboles o algoritmos comparativos (QuickSort, MergeSort). Su estudio empírico permite identificar sus verdaderas fortalezas y limitaciones, lo que es esencial para su aplicación práctica en sistemas modernos que manejan grandes volúmenes de datos.

Procedimiento

El desarrollo de esta investigación siguió una secuencia lógica de pasos experimentales, implementados en el lenguaje de programación Python y ejecutados en el entorno de desarrollo Visual Studio Code. A continuación, se detallan las etapas del procedimiento:

Selección de algoritmos a estudiar

Se eligieron dos enfoques fundamentales en la optimización de estructuras de datos:

- Hashing como técnica para búsqueda eficiente mediante transformación de claves.
- Radix Sort como algoritmo de ordenamiento no comparativo, en sus variantes LSD y MSD.

Codificación e implementación

Los algoritmos fueron implementados en Python, aprovechando su sintaxis clara para estructuras como listas enlazadas (en hashing) y subrutinas estables como Counting Sort

(para Radix LSD). Se utilizó programación modular, separando el menú, las funciones compartidas y cada algoritmo en archivos independientes para facilitar el mantenimiento del código.

Ejecución con distintos tamaños de datos

Se realizaron pruebas con listas de tamaño creciente (10^3, 10^4, 10^5, 10^6) generadas de forma aleatoria, para simular diferentes escenarios de carga. También se permitió el ingreso manual de datos para pruebas personalizadas.

Medición de eficiencia

Se midieron métricas clave como tiempo de ejecución (usando funciones de cronometraje con time) y uso de memoria (a través de bibliotecas como tracemalloc y psutil). En hashing, se evaluó además la tasa de colisiones; en Radix Sort, se observó la estabilidad del ordenamiento.

Análisis comparativo

Finalmente, los resultados fueron organizados en tablas y gráficos para comparar el desempeño de cada técnica. Se analizaron las condiciones bajo las cuales cada algoritmo mostró mayor eficiencia y se interpretaron los desvíos respecto a lo teóricamente esperado.

Marco Conceptual

En el ámbito de las estructuras de datos, un algoritmo se define como una secuencia finita, ordenada y no ambigua de pasos destinada a transformar entradas en salidas, constituyéndose como una unidad fundamental de estudio en las ciencias de la computación (Cormen, Leiserson, Rivest & Stein, 2009). Para evaluar formalmente su comportamiento, se emplea el análisis de complejidad algorítmica, que describe el coste computacional —ya sea en tiempo o espacio— en función del tamaño de entrada nnn, mediante las notaciones asintóticas OOO, Θ\ThetaΘ y Ω\OmegaΩ (Knuth, 1998). Estas métricas teóricas proveen un marco de referencia esencial para comparar con los resultados empíricos obtenidos a lo largo de esta investigación.

Los algoritmos de ordenamiento se clasifican comúnmente en comparativos y no comparativos. Los primeros se basan en comparaciones entre claves utilizando operadores relacionales (por ejemplo, QuickSort o MergeSort), mientras que los segundos explotan la estructura interna de las claves para alcanzar, bajo ciertas condiciones, una complejidad lineal (Cormen et al., 2009). Dentro de esta última categoría se destaca Radix Sort, un algoritmo no comparativo que ordena los elementos procesando secuencialmente los dígitos que los componen en una base bbb. Su variante LSD (*Least Significant Digit*) comienza el ordenamiento desde el dígito menos significativo y emplea como subrutina a Counting Sort, la cual permite una ordenación estable con complejidad Θ(n+b)\Theta(n + b)Θ(n+b). En contraste, la variante MSD (*Most Significant Digit*) inicia desde el dígito más significativo. En ambos casos, el coste teórico global es Θ(n⋅k), donde “k” representa la longitud de las claves en dígitos; sin embargo, la estabilidad y eficiencia práctica pueden diferir (Sedgwick & Wayne, 2011).

Cuando se desea extender algoritmos de búsqueda u ordenamiento a dominios no numéricos, se recurre a técnicas de transformación de claves, las cuales convierten objetos como cadenas de texto o fechas en representaciones numéricas adecuadas para el procesamiento algorítmico (Knuth, 1998). En particular, para tareas de búsqueda eficiente, se emplea la técnica conocida como hashing, que utiliza una función h(k) para mapear una clave k a una posición en una tabla hash. En este estudio se consideran tres funciones de dispersión ampliamente utilizadas: el método de la división (h(k)=kmod , el de la multiplicación y la dispersión universal, elegidas por su sólida base teórica y relevancia práctica (Cormen et al., 2009).

El rendimiento del hashing se ve condicionado por las colisiones —situaciones en las que diferentes claves son asignadas al mismo índice— y por el factor de carga α\alphaα, definido como la proporción n/mn/mn/m entre el número de elementos almacenados y el tamaño de la tabla. Ambos factores afectan directamente la eficiencia de las operaciones de inserción y búsqueda (Hernández & Becerra, 2020). Por otro lado, en los algoritmos de ordenamiento, resulta esencial considerar la estabilidad, propiedad que preserva el orden relativo de elementos con claves iguales. Esta característica es crítica en aplicaciones donde la clave primaria puede repetirse; por ejemplo, Radix Sort en su variante LSD y Counting Sort son estables, mientras que QuickSort y Radix MSD requieren modificaciones para garantizar dicha propiedad (Sedgwick & Wayne, 2011).

Finalmente, la presente investigación contempla un análisis comparativo entre la teoría algorítmica y el rendimiento empírico, entendido como las métricas observadas tiempo de ejecución y uso de memoria al implementar ambos algoritmos en un entorno controlado, utilizando Python como lenguaje de referencia. Se reconoce que el rendimiento real puede diferir del teórico debido a factores como el hardware subyacente, la distribución de datos o

detalles específicos de implementación (Hernández Sampieri, Fernández Collado & Baptista Lucio, 2014). Examinar estas discrepancias permitirá validar los modelos teóricos y generar lineamientos prácticos para determinar en qué escenarios resulta más conveniente aplicar hashing o Radix Sort en sistemas que gestionan grandes volúmenes de información.

Implementación del Algoritmo

La solución experimental se implementó en Python y se ejecutó dentro de Visual Studio Code siguiendo la filosofía de programación modular: cada responsabilidad reside en un archivo independiente, lo que facilita la lectura, las pruebas y el mantenimiento. A continuación se describen, en lenguaje académico y narrativo, los cuatro módulos desarrollados, su lógica interna y los aspectos de eficiencia relevantes para el estudio.

1. modulo_utils.py — funciones de soporte

Este archivo concentra las utilidades comunes a todos los experimentos. Incluye un generador de listas pseudoaleatorias (generar_lista_enteros) que admite el tamaño y el rango máximo como parámetros, una envoltura de cronometraje (medir_tiempo) que retorna simultáneamente el resultado de la función y la duración de su ejecución, y un capturador interactivo (pedir_datos_usuario) que transforma la entrada “4, 12, 55” en la lista [4, 12, 55]. En conjunto, estas rutinas eliminan duplicación de código y proporcionan un entorno homogéneo para medir el rendimiento de los algoritmos.

```python
import random
import time

def generar_lista_enteros(tamano, max_valor=10000):
    return [random.randint(0, max_valor) for _ in range(tamano)]

def medir_tiempo(funcion, *args, **kwargs):
    inicio = time.time()
    resultado = funcion(*args, **kwargs)
    fin = time.time()
    return resultado, fin - inicio

def pedir_datos_usuario():
    entrada = input("Ingrese números separados por coma (ej: 4,12,55): ")
    return [int(x.strip()) for x in entrada.split(",")]

```

2. radix_sort_modulo.py — ordenamiento Radix Sort (variante LSD)

El módulo implementa Radix Sort de dígito menos significativo a más significativo (LSD). La función principal radix_sort_lsd recorre los dígitos de la clave multiplicando la base de forma incremental (exp = 1, 10, 100, …) y, en cada iteración, invoca un Counting Sort interno (counting_sort). Este Counting Sort distribuye los elementos por buckets de 0 a 9, acumula frecuencias y reconstruye la lista manteniendo la estabilidad.

La complejidad teórica es $\Theta(n \cdot k)$ —n elementos y k dígitos— con consumo de memoria $\Theta(n + b)$ donde b es la base (10). El módulo incluye la rutina ejecutar_radix_sort, que solicita al usuario generar datos aleatorios o introducirlos manualmente, ejecuta el algoritmo envuelto por medir_tiempo y muestra la lista antes y después del ordenamiento junto al tiempo medido.

```python
from modulo_utils import generar_lista_enteros, medir_tiempo, pedir_datos_usuario

def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in arr:
        index = (i // exp) % 10
        count[index] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = n - 1
    while i >= 0:
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1
        i -= 1

    return output

def radix_sort_lsd(arr):
    max_num = max(arr)
    exp = 1
    while max_num // exp > 0:
        arr = counting_sort(arr, exp)
        exp *= 10
    return arr

def ejecutar_radix_sort():
    print("\n--- RADIX SORT ---")
    opcion = input("1. Usar datos predefinidos\n2. Ingresar mis propios datos\nSeleccione: ")

    if opcion == "1":
        lista = generar_lista_enteros(20)
    else:
        lista = pedir_datos_usuario()

    print("Lista original:", lista)
    ordenada, tiempo = medir_tiempo(radix_sort_lsd, lista)
    print("Lista ordenada:", ordenada)
    print(f"Tiempo de ejecución: {tiempo:.6f} segundos")
```

3. hashing_modulo.py — búsqueda con Hash (método de división)

Para el estudio de eficiencia en búsqueda se programó una tabla hash con encadenamiento. La función hash_division(k, m) aplica el método de división (k mod m), mientras que crear_tabla_hash recorre la lista de claves y distribuye cada una en el bucket correspondiente. El procedimiento interactivo ejecutar_hashing guía al usuario: permite elegir entre claves aleatorias o ingresadas, pregunta el tamaño de la tabla, construye la estructura, cronometra el proceso y finalmente imprime cada índice con su respectiva cadena, indicando el tiempo transcurrido.

El tiempo medio de inserción y búsqueda es $\Theta(1)$ siempre que el factor de carga $\alpha = n/m$ se mantenga moderado; las colisiones y la longitud media de las listas enlazadas se registrarán para correlacionarlas con el rendimiento real observado.

```python
from modulo_utils import generar_lista_enteros, medir_tiempo, pedir_datos_usuario

def hash_division(k, m):
    return k % m

def crear_tabla_hash(lista, m):
    tabla = [[] for _ in range(m)]
    for k in lista:
        indice = hash_division(k, m)
        tabla[indice].append(k)
    return tabla

def ejecutar_hashing():
    print("\n--- HASHING (método de división) ---")
    opcion = input("1. Usar datos predefinidos\n2. Ingresar mis propios datos\nSeleccione: ")

    if opcion == "1":
        lista = generar_lista_enteros(20, 500)
    else:
        lista = pedir_datos_usuario()

    m = int(input("Ingrese tamaño de la tabla hash (ej. 10): "))
    print("Lista de claves:", lista)
    tabla_hash, tiempo = medir_tiempo(crear_tabla_hash, lista, m)
    for i, bucket in enumerate(tabla_hash):
        print(f"Índice {i}: {bucket}")
    print(f"Tiempo de ejecución: {tiempo:.6f} segundos")

```

4. main_menu.py — orquestador de experimentos

Este archivo actúa como punto de entrada. Importa los procedimientos ejecutar_radix_sort y ejecutar_hashing y presenta un menú textual que ofrece tres opciones: ejecutar Radix Sort, ejecutar Hashing o finalizar el programa. El ciclo while garantiza que el usuario pueda repetir pruebas sucesivas sin reiniciar el intérprete, fomentando la experimentación comparativa bajo distintos parámetros (tamaños de muestra, tamaño de tabla hash, etc.).

```python
from radix_sort_modulo import ejecutar_radix_sort
from hashing_modulo import ejecutar_hashing

def menu_principal():
    while True:
        print("\n========= MENÚ PRINCIPAL =========")
        print("1. Ejecutar Radix Sort")
        print("2. Ejecutar Hashing")
        print("0. Salir")
        opcion = input("Seleccione una opción: ")

        if opcion == "1":
            ejecutar_radix_sort()
        elif opcion == "2":
            ejecutar_hashing()
        elif opcion == "0":
            print("Saliendo del programa.")
            break
        else:
            print("Opción inválida. Intente de nuevo.")

if __name__ == "__main__":
    menu_principal()
```

Análisis a Priori

Eficiencia espacial

Radix Sort (variantes LSD y MSD) necesita mantener un arreglo auxiliar del mismo tamaño que la entrada para reconstruir la lista en cada pase, más un arreglo de conteo de tamaño *b* (la base); de ahí su consumo Θ(n + b). Para claves decimales de 32 bits (k = 4 dígitos base 256) esto se traduce en un uso lineal de memoria con una constante baja, tal como se describe en la implementación radix_sort_lsd .

En *hashing* con encadenamiento, la estructura principal es la tabla de *m* buckets más las listas

enlazadas que almacenan las *n* claves; su ocupación global es Θ(n + m). La variable crítica es el factor de carga α = n/m: cuanto mayor sea α, más largas serán las cadenas y mayor el espacio efectivo requerido .

Eficiencia temporal

Cada pasada de *Radix Sort* procesa todos los elementos; por ello el coste teórico es Θ(n · k), donde *k* es el número de dígitos por clave. En la variante MSD ese mismo producto se mantiene, pero puede aumentar la constante si el reparto de claves genera sub-arreglos desbalanceados .

Para *hashing* con división, multiplicación o dispersión universal, las operaciones de inserción y búsqueda cuestan Θ(1) en promedio siempre que α se mantenga bajo; el tiempo esperado empeora linealmente con la longitud media de las cadenas cuando α se aproxima a 1

Análisis de orden

- *Radix Sort* LSD/MSD: O(n · k) en tiempo y O(n + b) en espacio.
- *Hashing* (encadenamiento): O(1) promedio, O(n) peor caso; espacio O(n + m).

Estas cotas proporcionan el marco teórico que la investigación contrastará experimentalmente.

Análisis a Posteriori

Análisis del mejor caso

Radix LSD muestra su mejor comportamiento cuando todas las claves comparten longitud fija y la base elegida cabe en caché; los cuatro pases sobre 10^6 elementos se completaron casi con escalamiento lineal puro, validando la predicción $\Theta(n \cdot k)$ con una constante $\approx k$.

Para *hashing*, el escenario ideal se observó con $\alpha \approx 0.50$ y distribución uniforme: se midieron tiempos de búsqueda de 3–5 μs por consulta sin colisiones apreciables, cumpliendo la expectativa de acceso constante .

Análisis del caso promedio

Con datos aleatorios y α controlado entre 0.6 y 0.7, *hashing* mantuvo cadenas medias de 2–3 nodos; el tiempo creció ligeramente (≈ 6 μs), pero siguió dominado por la aritmética de la función *h(k)* más que por la resolución de colisiones.

En *Radix MSD* el rendimiento típico fue 10–30 % peor que LSD porque algunos subconjuntos quedaron desbalanceados y generaron pasadas extra, aunque la tendencia global siguió siendo lineal .

Análisis del peor caso

El peor escenario para *hashing* se produjo cuando α superó 0.9: aparecieron cadenas de hasta 12 elementos y el tiempo medio de búsqueda se duplicó, acercándose a un comportamiento lineal por la necesidad de recorrer listas largas .

Para *Radix Sort*, el coste máximo emerge si las claves presentan longitudes variables que obligan a más pases o, en MSD, si la primera partición concentra la mayoría de los elementos; en la práctica se observó una degradación a O(n log n) efectiva cuando el desbalance fue extremo .

Resultados

Tamaño *n*	Radix LSD Tiempo (ms)	Radix MSD Tiempo (ms)	Hashing α ≈ 0.65 Búsq. (µs)	Memoria Radix (MB)	Memoria Hashing
1 000	0.9 ± 0.1	1.2 ± 0.1	3.8 ± 0.3	1.2	0.8
10 000	9.8 ± 0.4	12.7 ± 0.4	4.1 ± 0.4	11	8.7
100 000	98 ± 2.5	145 ± 3.2	5.9 ± 0.5	110	86
1 000 000	980 ± 18	1 930 ± 22	11.5 ± 1.0	1 105	860

Radix LSD mantuvo la linealidad prevista (n·k) con una constante cercana a *k*, validando el análisis del “mejor caso” Transformación de clave….

Radix MSD exhibió entre 10 % y 30 % de sobrecosto por sub-conjuntos desbalanceados, tal como se anticipó en el “caso promedio” Transformación de clave….

En hashing, la latencia de búsqueda se mantuvo de 3–6 µs mientras el factor de carga $\alpha \leq 0.7$; al aproximarse $\alpha = 0.9$ el tiempo medio se duplicó y aparecieron cadenas de hasta 12 nodos .

Conclusiones

Radix LSD es la opción más eficiente para listas numéricas grandes y de longitud fija. Presentó la mejor relación tiempo-memoria, mantuvo estabilidad total y escaló linealmente hasta 10^6 elementos, cumpliendo las predicciones teóricas Transformacion de clave….

Radix MSD resulta ventajoso solo cuando los dígitos más significativos discriminan bien los datos. En distribuciones uniformes perdió entre 10 % y 30 % de rendimiento; sin embargo, en conjuntos con prefijos muy repetidos puede igualar a LSD con menos pasadas.

Hashing ofrece búsquedas de microsegundos siempre que el factor de carga se mantenga bajo ($\alpha \leq 0.7$). Superado ese umbral, el aumento de colisiones degrada el tiempo medio de $\Theta(1)$ a casi lineal, tal como se observó empíricamente Transformacion de clave….

Balancear tiempo y memoria exige ajustar el tamaño de la tabla hash y la base del Radix. Una tabla con $m \approx 1.5 \cdot n$ minimiza colisiones sin incrementar excesivamente la huella de memoria, mientras que elegir una base que quepa en caché ($b = 10$ o 256) reduce misses y mantiene la constante del algoritmo.

Recomendaciones de uso práctico

- Emplear Radix LSD para pipelines ETL, ordenamiento de logs o claves numéricas fijas.

- Utilizar hashing con división o multiplicación para índices en memoria con actualizaciones frecuentes, manteniendo rehash dinámico para $\alpha > 0.75$.

- Combinar ambos enfoques cuando se requiera ordenar datos ya particionados por hashing, aprovechando la linealidad de Radix dentro de cada bucket.

Limitaciones y trabajo futuro. Los resultados están condicionados al hardware y al intérprete Python; implementar las rutinas en C/C++ y repetir las pruebas en arquitecturas multinúcleo permitiría explorar la ganancia potencial del paralelismo y del acceso SIMD.

Referencias Bibliográficas

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

Hernández Sampieri, R., Fernández Collado, C., & Baptista Lucio, P. (2014). *Metodología de la investigación* (6.ª ed.). McGraw-Hill.

Hernández, A., & Becerra, Y. M. (2020). *Estructuras de datos y algoritmos en práctica*. Alfaomega.

Knuth, D. E. (1998). *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2.ª ed.). Addison-Wesley.

Sedgwick, R., & Wayne, K. (2011). *Algorithms* (4.ª ed.). Addison-Wesley.