UNIVERSIDAD AMERICANA

Algoritmos y estructuras de datos

Documentación Hashing.

Integrantes:

Franco Xavier Aguilera Ortez

David Joel Sánchez Acevedo

Alicia Massiel Estrada Acevedo

Sara Alejandra Zambrana Taylor

Andrea Johanna Duarte Guerrero

Docente:

Ing. Silvia Ticay.

25 de Junio del 2025

experimentos.py:

```
import random
import time
import statistics
import csv
import os
from hash import HashTable
from memoria import profile_memory_insert
```

- **random:** genera números aleatorios para simular claves.
- **time:** mide tiempos de ejecución de operaciones.
- **statistics:** calcula promedios y desviaciones estándar.
- **csv:** permite exportar resultados a archivos .csv.
- **os:** permite limpiar la pantalla en consola.
- **HashTable:** clase de tabla hash importada desde hash.py.
- **profile_memory_insert:** función para medir uso de memoria (desde memoria.py).

```
def clear_screen():
    """Limpia la pantalla de la consola."""
    os.system('cls' if os.name == 'nt' else 'clear')

def print_separator():
    print("=" * 60)

def print_section(title):
    print_separator()
    print(f"{title.center(60)}")
    print_separator()

def print_kv_row(label, value, unit=None, width=30):
    val_str = f"{value:.6f}" if isinstance(value, float) else
str(value)
    unit_str = f" {unit}" if unit else ""
    print(f"{label:<{width}}: {val_str}{unit_str}")
```

Estas funciones mejoran la presentación en la consola:

- **clear_screen()**: limpia la pantalla dependiendo del sistema operativo.
- **print_separator()**: imprime una línea de "=" para separar secciones.
- **print_section(title)**: imprime un título centrado entre separadores.
- **print_kv_row(...)**: imprime una línea con una etiqueta, un valor y una unidad (opcional), con formato limpio.

```
def test_hash_table(n_elements, n_search, n_delete, table_size,
percent_non_existing=0.5):
    """
    Realiza inserciones, búsquedas y eliminaciones sobre la tabla hash.
    Devuelve los tiempos de ejecución de cada operación.
    """
```

Esta función simula operaciones reales sobre una tabla hash para medir cuánto tardan:

- Inserta n_elements claves aleatorias.
- Busca n_search claves (algunas no existen).
- Elimina n_delete claves (algunas no existen).

Devuelve tres tiempos: de inserción, búsqueda y eliminación.

```
    ht = HashTable(table_size)
    # Genera claves aleatorias y claves no existentes
    keys = [random.randint(0, 10 * n_elements) for _ in
range(n_elements)]
    non_existing_keys = [max(keys) + 1 + i for i in range(max(n_search,
n_delete))]
```

- **ht:** se crea una nueva tabla hash.
- **keys**: claves aleatorias que se insertarán.
- **non_existing_keys:** claves que sabemos que no existen en la tabla.

```
    # Inserción
    t0 = time.perf_counter()
    for k in keys:
        ht.insert(k, str(k))
    t1 = time.perf_counter()
    insert_time = t1 - t0
```

t0 = time.perf_counter():

- Toma la “marca de tiempo” antes de comenzar la inserción.
- perf_counter() ofrece alta resolución para medir intervalos breves.

Bucle for k in keys:

- Recorre todas las claves generadas aleatoriamente.
- Llama a ht.insert(k, str(k)) para insertar cada par (clave, valor) en la tabla.

t1 = time.perf_counter():

- Toma la marca de tiempo justo después de terminar las inserciones.

insert_time = t1 - t0:

- Calcula la duración total de la inserción restando los dos instantes.
- Este es el tiempo de inserción que retorna la función.

```
    # Búsqueda (mitad existentes, mitad no existentes)
    t0 = time.perf_counter()
    for i in range(n_search):
        key = non_existing_keys[i] if i < int(n_search *
percent_non_existing) else random.choice(keys)
        ht.search(key)
    t1 = time.perf_counter()
    search_time = t1 - t0
```

1. **t0 = time.perf_counter()**: inicia la medición.
2. **Bucle for i in range(n_search)**:

- Para cada iteración decide si buscar una clave que no existe (primer 50 % si percent_non_existing=0.5) o una existente (resto).
- key = …: escoge de non_existing_keys o de keys.
- ht.search(key): realiza la búsqueda en la tabla.

3. **t1 = time.perf_counter():** marca el final.
4. **search_time = t1 - t0:** duración total de las búsquedas.

```
    # Eliminación (mitad existentes, mitad no existentes)
    t0 = time.perf_counter()
    for i in range(n_delete):
        key = non_existing_keys[i] if i < int(n_delete *
percent_non_existing) else random.choice(keys)
        ht.delete(key)
    t1 = time.perf_counter()
    delete_time = t1 - t0
```

Es idéntico al de búsqueda, pero aplicando ht.delete(key) en lugar de search.

Mide cuánto tardan las operaciones de eliminación, tanto de claves válidas como de intentos fallidos.

```
    return insert_time, search_time, delete_time
```

Devuelve los tres valores medidos para que quien invoque la función pueda procesarlos o exportarlos.

```
def prueba_personalizada_export():
    """
    Ejecuta una sola prueba personalizada e **exporta** el detalle a
'personalizada_hash.csv'.
    Muestra resumen en pantalla y tiempo total de ejecución.
    """
    clear_screen()
    print_section("PRUEBA PERSONALIZADA CON EXPORTACIÓN")
    total_t0 = time.perf_counter()
```

- Limpia pantalla y muestra título.
- Inicia cronómetro general.

```

    n = int(input("Ingrese cantidad de elementos: "))
    m = int(input("Ingrese tamaño de la tabla hash: "))
    b = int(input("Número de búsquedas: "))
    e = int(input("Número de eliminaciones: "))
    print("\nEjecutando prueba, por favor espere...\n")
```

Lee parámetros del usuario:

- **n:** elementos a insertar.
- **m**: tamaño de la tabla.
- **b**: búsquedas a realizar.

- **e:** eliminaciones a realizar.

```
    ins, bus, elim = test_hash_table(n, b, e, m)
    mem = profile_memory_insert(n, m)
    total_t1 = time.perf_counter()
```

- Lanza la prueba de inserción/búsqueda/eliminación.
- Mide uso de memoria con profile_memory_insert.
- Toma cronómetro final.

```
    row = {
        'n_elements': n,
        'table_size': m,
        'factor_carga': round(n / m, 2),
        'insert_time': ins,
        'search_time': bus,
        'delete_time': elim,
        'memory_usage': mem,
        'total_exec_time': total_t1 - total_t0
    }
    filename = 'personalizada_hash.csv'
    with open(filename, 'w', newline='') as f:
        writer = csv.DictWriter(f, fieldnames=row.keys())
        writer.writeheader()
        writer.writerow(row)
```

- Construye un diccionario con todas las métricas
- Crea (o sobrescribe) el CSV con una sola fila de resultados.

```
    print_section("RESUMEN DE LA PRUEBA")
    print_kv_row("Elementos insertados", n)
    print_kv_row("Tamaño de la tabla", m)
    print_kv_row("Factor de carga", row['factor_carga'])
    print_kv_row("Tiempo de inserción", ins, "s")
    print_kv_row("Tiempo de búsqueda", bus, "s")
    print_kv_row("Tiempo de eliminación", elim, "s")
    print_kv_row("Uso máx. de memoria", mem, "MB")
    print_kv_row("Tiempo total de ejecución", row['total_exec_time'],
"s")
```

```
    print_separator()
    print(f"\n¡Resultados exportados a {filename}!\n")
```

- Muestra en consola un resumen bonito y confirma la exportación.

```
def perfil_memoria_simple_export():
    """
    Mide el uso de memoria para una inserción y **exporta** el
resultado a 'memoria_hash.csv'.
    Muestra resumen en pantalla y tiempo total de ejecución.
    """
    clear_screen()
    print_section("PERFILADO DE MEMORIA SIMPLE CON EXPORTACIÓN")
    total_t0 = time.perf_counter()
    n = int(input("Ingrese cantidad de elementos: "))
    m = int(input("Ingrese tamaño de la tabla hash: "))
    print("\nEjecutando perfilado de memoria, por favor espere...\n")
    uso = profile_memory_insert(n, m)
    total_t1 = time.perf_counter()
    row = {
        'n_elements': n,
        'table_size': m,
        'factor_carga': round(n / m, 2),
        'memory_usage': uso,
        'total_exec_time': total_t1 - total_t0
    }
    filename = 'memoria_hash.csv'
    with open(filename, 'w', newline='') as f:
        writer = csv.DictWriter(f, fieldnames=row.keys())
        writer.writeheader()
        writer.writerow(row)
    print_section("RESUMEN DEL PERFIL DE MEMORIA")
    print_kv_row("Elementos insertados", n)
    print_kv_row("Tamaño de la tabla", m)
    print_kv_row("Factor de carga", row['factor_carga'])
    print_kv_row("Uso máx. de memoria", uso, "MB")
    print_kv_row("Tiempo total de ejecución", row['total_exec_time'],
"s")
    print_separator()
    print(f"\n¡Resultados exportados a {filename}!\n")
```

Muy parecido al anterior, pero enfocado solo en memoria:

- Limpia pantalla y título.
- Toma n y m por input.
- Ejecuta uso = profile_memory_insert(n, m).
- Cronómetro total.
- Empaqueta en row con solo:
    - n_elements, table_size, factor_carga, memory_usage, total_exec_time.

- Exporta a memoria_hash.csv.
- Imprime resumen.

```
def run_experiments_export(
    sizes=[1000, 5000, 10000],
    table_sizes=[1000, 2000, 5000],
    n_search=1000,
    n_delete=1000,
    repetitions=10,
    csv_filename='resultados_hash.csv'
):
    """
    Ejecuta experimentos repetidos para cada combinación de n_elements
y table_size.
    Exporta promedios y desviaciones estándar a CSV y muestra resumen.
    """
    clear_screen()
    print_section("EXPERIMENTO COMPLETO CON REPETICIONES Y
EXPORTACIÓN")
    total_t0 = time.perf_counter()
    results = []
    for n_elements in sizes:
        for table_size in table_sizes:
            factor_carga = n_elements / table_size
            insert_times, search_times, delete_times, mem_usages = [],
[], [], []
            for i in range(repetitions):
                print(f"Prueba {i+1}/{repetitions} - n={n_elements},
tabla={table_size}, α={factor_carga:.2f}")
                ins_t, sch_t, del_t = test_hash_table(
                    n_elements, n_search, n_delete, table_size,
percent_non_existing=0.5
                )
```

```
                insert_times.append(ins_t)
                search_times.append(sch_t)
                delete_times.append(del_t)
                mem_usages.append(profile_memory_insert(n_elements,
table_size))
            results.append({
                'n_elements': n_elements,
                'table_size': table_size,
                'factor_carga': round(factor_carga, 2),
                'insert_time_avg': statistics.mean(insert_times),
                'insert_time_std': statistics.stdev(insert_times) if
len(insert_times) > 1 else 0.0,
                'search_time_avg': statistics.mean(search_times),
                'search_time_std': statistics.stdev(search_times) if
len(search_times) > 1 else 0.0,
                'delete_time_avg': statistics.mean(delete_times),
                'delete_time_std': statistics.stdev(delete_times) if
len(delete_times) > 1 else 0.0,
                'memory_usage_avg': statistics.mean(mem_usages),
                'memory_usage_std': statistics.stdev(mem_usages) if
len(mem_usages) > 1 else 0.0,
            })
            print("... Listo.")
    with open(csv_filename, mode='w', newline='') as csv_file:
        writer = csv.DictWriter(csv_file, fieldnames=[
            'n_elements', 'table_size', 'factor_carga',
            'insert_time_avg', 'insert_time_std',
            'search_time_avg', 'search_time_std',
            'delete_time_avg', 'delete_time_std',
            'memory_usage_avg', 'memory_usage_std'
        ])
        writer.writeheader()
        for row in results:
            writer.writerow(row)
    total_t1 = time.perf_counter()
    print_section("RESUMEN DEL EXPERIMENTO")
    print_kv_row("Configuraciones ejecutadas", len(results))
    print_kv_row("Repeticiones por configuración", repetitions)
    print_kv_row("Tiempo total de ejecución", total_t1 - total_t0, "s")
    print_separator()
    print(f"\n¡Resultados exportados a {csv_filename}!\n")
```

- Triple bucle:
    1. Sobre cada valor de sizes
    2. Sobre cada table_size
    3. Repite repetitions veces para estabilidad estadística
- Almacena listas de resultados parciales.
- Tras el bucle interno, calcula media y desviación con statistics.

Al final, exporta results completo a resultados_hash.csv y muestra un resumen global.

hash.py:

```
# hash.py
"""
Estructura de datos HashTable con encadenamiento separado.
Contiene las clases Node (nodo de lista enlazada) y HashTable para
inserción, búsqueda y eliminación eficiente.
"""

class Node:
    """Nodo simple para lista enlazada, usado en colisiones de la tabla
hash."""
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
```

Cada nodo almacena:

- key, value
- next: referencia al siguiente nodo en la lista encadenada.

```
class HashTable:
    """ Tabla hash con encadenamiento separado.
    - size: número de slots/buckets
    - table: lista de referencias a la cabeza de cada lista enlazada
    """
    def __init__(self, size):
        self.size = size
        self.table = [None] * size
```

- **size:** número de buckets.
- **table:** lista inicializada con None.

```
    def hash_function(self, key):
        """Devuelve el índice hash de la clave."""
        return hash(key) % self.size
```

Esta función recibe una clave y devuelve un índice dentro de los límites de la tabla.

- La función hash() convierte cualquier tipo de dato hashable (números, cadenas, etc.) en un número entero.
- Luego, se aplica el módulo con self.size para asegurarse de que el resultado esté en el rango [0, size - 1].
- Esto permite ubicar cada clave en una posición válida dentro del arreglo table.

```
    def insert(self, key, value):
        """Inserta o actualiza un par clave-valor."""
        idx = self.hash_function(key)
        node = self.table[idx]
        while node:
            if node.key == key:
                node.value = value
                return
            node = node.next
        new_node = Node(key, value)
        new_node.next = self.table[idx]
        self.table[idx] = new_node
```

Esta función inserta un nuevo par clave-valor o actualiza uno existente.

- Se calcula el índice de la clave usando la función hash.
- Se recorre la lista enlazada en la posición idx para verificar si la clave ya existe.
- Si se encuentra una coincidencia, se actualiza el valor.
- Si no se encuentra, se crea un nuevo nodo y se agrega al inicio de la lista enlazada.

```
    def search(self, key):
        """Devuelve el valor de la clave o None si no existe."""
        idx = self.hash_function(key)
```

```
        node = self.table[idx]
        while node:
            if node.key == key:
                return node.value
            node = node.next
        return None
```

Esta función busca una clave en la tabla y devuelve su valor asociado si existe.

- Se calcula el índice usando la función hash.
- Se recorre la lista en la posición correspondiente.
- Si se encuentra la clave, se retorna el valor.
- Si no se encuentra, se retorna None.

```
    def delete(self, key):
        """Elimina el nodo con la clave dada; True si lo elimina, False
si no existe."""
        idx = self.hash_function(key)
        node = self.table[idx]
        prev = None
        while node:
            if node.key == key:
                if prev:
                    prev.next = node.next
                else:
                    self.table[idx] = node.next
                return True
            prev = node
            node = node.next
        return False
```

Esta función elimina un nodo por su clave y devuelve True si la eliminación fue exitosa, o False si no se encontró.

- Se calcula el índice correspondiente.
- Se recorren los nodos de la lista enlazada.
- Se mantiene una referencia al nodo anterior (prev) para poder ajustar los punteros al eliminar.
- Si la clave coincide, se ajustan los enlaces y se retorna True.
- Si al final no se encuentra la clave, se retorna False.

main.py:

```
# main.py
"""
Menú principal del experimento. Solo contiene la lógica del menú y las
llamadas a funciones.
"""

# main.py
from experimentos import (
    prueba_personalizada_export,
    perfil_memoria_simple_export,
    run_experiments_export,
    clear_screen
)

def main_menu():
    while True:
        # Limpiar pantalla (opcional, depende del sistema operativo)
        clear_screen()
        print("\n=== Menú principal (Hash Experimental, todo se
exporta) ===")
        print("1. Prueba personalizada (exporta
personalizada_hash.csv)")
        print("2. Solo perfil de memoria (exporta memoria_hash.csv)")
        print("3. Experimento completo (repeticiones, exporta
resultados_hash.csv)")
        print("4. Salir")
        op = input("Seleccione una opción: ")
        match op:
            case "1":
                prueba_personalizada_export()
            case "2":
                perfil_memoria_simple_export()
            case "3":
                run_experiments_export(
                    sizes=[20000,55000, 100000, 200000, 500000,
1000000],
                    table_sizes=[1000, 2000, 5000],
                    n_search=1000,
```

```
                    n_delete=1000,
                    repetitions=10,
                    csv_filename='resultados_hash.csv'
                )
            case "4":
                print("¡Hasta luego!")
                break
            case _:
                print("Opción no válida. Intente de nuevo.")
```

- Bucle infinito hasta que elija “4”.
- Usa match (Python 3.10+) para despachar cada opción.
- Llama a las funciones de experimentos.py.

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

memoria_hash.py:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('resultados_hash.csv')
# Filtrar por tamaño de tabla, por ejemplo 2000
table_size = 2000
df_filtrado = df[df['table_size'] == table_size]
```

Se importan las librerías necesarias:

- pandas para manipular datos.
- matplotlib.pyplot para graficar.

Se carga el archivo CSV que contiene los resultados experimentales.
Se filtran solo las filas que corresponden a un tamaño de tabla específico (por ejemplo, 2000).

```
plt.figure(figsize=(8, 5))
plt.plot(df_filtrado['n_elements'], df_filtrado['memory_usage_avg'],
marker='o', label=f'Tabla {table_size}')
```

- Se establece el tamaño del gráfico.

- Se crea una curva donde el eje X representa el número de elementos insertados, y el eje Y el uso promedio de memoria.
- Los puntos son marcados con círculos para mayor claridad visual.

```
plt.xlabel('Número de elementos insertados (n_elements)')
plt.ylabel('Uso promedio de memoria (MB)')
plt.title('Carga de datos vs Uso de memoria\n(resultados_hash.csv,
tabla=2000)')
plt.grid(True)
plt.tight_layout()
plt.legend()
plt.show()
```

- Se agregan etiquetas a los ejes.
- Se establece un título.
- Se activa la cuadrícula.
- Se ajusta automáticamente el espacio para que no se solapen etiquetas.
- Se añade leyenda y se muestra el gráfico final.

memoria.py:

```
# memoria.py
"""
Funciones para perfilamiento y medición de uso de memoria usando
memory_profiler.
Requiere instalar memory_profiler: pip install memory_profiler
"""

from memory_profiler import memory_usage
from hash import HashTable
import random

def profile_memory_insert(n_elements, table_size):
    """
    Mide el consumo de memoria durante la inserción de n_elements en
una HashTable.
    Devuelve el uso máximo en MB.
    """
    def run():
        ht = HashTable(table_size)
```

```
        keys = [random.randint(0, 10 * n_elements) for _ in
range(n_elements)]
        for k in keys:
            ht.insert(k, str(k))
    mem = memory_usage(run, max_iterations=1, interval=0.01)
    return max(mem) - min(mem)
```

resultados_hash.py:

```
import pandas as pd
import matplotlib.pyplot as plt

# Cargar los datos
df = pd.read_csv('resultados_hash.csv')

# Seleccionar el tamaño de tabla a graficar
table_size = 5000
df_filtrado = df[df['table_size'] == table_size]

# Asegurarse que n_elements es int
df_filtrado['n_elements'] = df_filtrado['n_elements'].astype(int)
```

- Se cargan los datos desde el archivo resultados_hash.csv.
- Se filtran los registros para que solo se graficen los que usan table_size = 5000.
- Se asegura que la columna n_elements sea de tipo entero, lo cual es importante para el eje X.

```


plt.figure(figsize=(12, 8))
```

Se define un tamaño más grande que en el gráfico anterior, ideal para comparar múltiples curvas.

```

# Graficar la curva con marcadores
plt.plot(
    df_filtrado['n_elements'],
    df_filtrado['memory_usage_avg'],
    marker='o',
    label=f'Tabla {table_size}'
)
```

Se grafica la curva de uso de memoria promedio por número de elementos insertados.

```
# Graficar la curva de tiempo de inserción
plt.plot(
    df_filtrado['n_elements'],
    df_filtrado['insert_time_avg'],
    marker='o',
    label=f'Tabla {table_size}'
)
plt.ylabel('Tiempo promedio de inserción (s)')
```

Se grafica también el tiempo promedio de inserción en segundos para cada cantidad de elementos.

```
# Etiquetas de los ejes y título
plt.xlabel('Número de elementos insertados (n_elements)')
plt.ylabel('Uso promedio de memoria (MB)')
plt.title(f'Carga de datos vs Uso de memoria\n(resultados_hash.csv,
tabla={table_size})')
plt.grid(True)
plt.tight_layout()
plt.legend()
```

Se ajustan los ejes, el título y se agrega la leyenda.

```
# Mostrar solo los valores exactos de n_elements como ticks del eje X
plt.xticks(
    df_filtrado['n_elements'],
    [f"{n:,}" for n in df_filtrado['n_elements']],  # Muestra 20,000 en
vez de 20000
    rotation=45
)
```

- Se muestra el número de elementos con comas como separador de miles (por ejemplo, 20,000).

- Las etiquetas se giran 45 grados para evitar superposición.

```
# Agregar etiquetas encima de cada punto con el valor de memoria
(opcional, pero profesional)
for x, y in zip(df_filtrado['n_elements'],
df_filtrado['memory_usage_avg']):
    plt.annotate(
        f"{y:.1f}",
        (x, y),
        textcoords="offset points",
        xytext=(0,8),
        ha='center',
        fontsize=9,
        color='navy'
    )
```

Sobre cada punto de memoria, se coloca una etiqueta que indica el valor con un decimal.

```
# Y etiquetas encima del punto:
for x, y in zip(df_filtrado['n_elements'],
df_filtrado['insert_time_avg']):
    plt.annotate(
        f"{y:.4f}",
        (x, y),
        textcoords="offset points",
        xytext=(0,8),
        ha='center',
        fontsize=9,
        color='darkgreen'
    )
```

Sobre cada punto de tiempo, se coloca una etiqueta con cuatro decimales para mejor precisión.

```
plt.show()
```

Finalmente, se renderiza y muestra la figura completa en pantalla.