UNIVERSIDAD ADOLFO IBÁÑEZ
FACULTAD DE INGENIERÍA Y CIENCIAS

Repaso prueba 2

Profesores: Sebastián Sáez, Diego Ramos

Ayudantes: Diego Duhalde, Benjamín Wiedmaier, Fernando Zamora

May 2, 2025

Contents

1 Preguntas de Selección Múltiple

1. En el contexto de hashing, una colisión ocurre cuando:

   A. La función de hash trabaja con enteros

   B. La función de hash trabaja con flotantes

   C. La función de hash retorna un valor correspondiente a un bucket ya utilizado

   D. La función de hash retorna un valor igual al ingresado

2. La complejidad de búsqueda en una lista enlazada es:

   A. $\Omega(1)$

   B. $\Omega(n)$

   C. $\Theta(1)$

   D. $\Theta(n)$

3. ¿Cuál es la complejidad de la búsqueda binaria?

   A. $O(n)$

   B. $O(n \log n)$

   C. $O(1)$

   D. $O(\log n)$

4. ¿Qué es el encadenamiento separado en tablas hash?

   A. Forma de concatenar listas con valores iguales consecutivos

   B. Uso de listas enlazadas para manejar colisiones

   C. Concatenar dos colas independientes

   D. Forma de ordenar los buckets antes de aplicar hashing

5. ¿Cuál afirmación es correcta al comparar arreglos y listas enlazadas?

   A. Un arreglo ocupa más memoria que una lista enlazada

   B. Insertar en un arreglo en la mitad es más eficiente

   C. Las listas almacenan sus elementos de manera contigua

   D. Buscar en listas enlazadas suele requerir más operaciones

2 Algoritmos de Ordenamiento

2.1 Bubble Sort

Ejercicio 1. Dado el arreglo `[5,2,9,1,5,6]`, simula Bubble Sort mostrando el estado tras cada pasada completa.

2.2 Selection Sort

Ejercicio 2. Aplica Selection Sort al arreglo `[6,4,1,3,9]` y explica en qué difiere de Bubble Sort.

2.3 Insertion Sort

Ejercicio 3. Ordena el arreglo `[3,7,4,9,5]` con Insertion Sort y da la complejidad en el mejor caso.

2.4 Merge Sort

Ejercicio 4. Traza división y fusión de `[8,3,7,4,9,2]`.

2.5 Quick Sort

Ejercicio 5. Aplica Quick Sort con pivote el primer elemento en `[10,7,8,9,1,5]` e indique la complejidad del caso promedio.

3 Algoritmos de Búsqueda

3.1 Búsqueda Lineal

Ejercicio 6. Busca 8 en `[15,3,7,8,23,42]` y cuenta comparaciones.

3.2 Búsqueda Binaria

Ejercicio 7. Busca 10 en `[2,4,6,8,10,12,14]`.

3.3 Búsqueda Exponencial

Ejercicio 8. Explica cómo se usaría una búsqueda exponencial seguida de una búsqueda binaria para encontrar un valor x en un arreglo ordenado de 16 elementos.

3.4 Búsqueda Binaria Modificada

Ejercicio 9. Primera ocurrencia de 7 en `[1,3,7,7,7,9,11]`.

4 Tablas Hash

4.1 Encadenamiento Separado

Ejercicio 10. Considere una tabla hash con función $h(x) = x \bmod 5$. Inserte los elementos `[7, 12, 17, 22]` utilizando el método de **encadenamiento separado** para resolver las colisiones.

4.2 Sondeo Lineal

Ejercicio 11. Considere una tabla hash con función $h(x) = x \bmod 7$. Inserte los elementos `[15, 22, 29, 36]` utilizando **sondeo lineal** para resolver las colisiones.

4.3 Redimensionamiento y Rehashing

Ejercicio 12. Considere una tabla hash de tamaño inicial 5 con una función $h(x) = x \bmod$ tamaño. Si el **factor de carga** supera 0.7, la tabla se duplica a tamaño 10 y se rehace la inserción de todos los elementos. Inserte los elementos `[3, 8, 13, 18, 23]` siguiendo este procedimiento.

5 Listas Enlazadas, Pilas y Colas

5.1 Pila con Lista Enlazada

Ejercicio 13. Implemente `push`, `pop`, `peek` y simule: `push(10)`, `push(20)`, `push(30)`, `pop()`, `pop()`.

5.2 Cola con Lista Enlazada

Ejercicio 14. Implemente `enqueue`, `dequeue`, `peek` y simule: `enqueue(5)`, `enqueue(15)`, `enqueue(25)`, `dequeue()`, `enqueue(35)`, `dequeue()`.

5.3 Inversión de Lista Enlazada

Ejercicio 15. Implementa una función que invierta una lista enlazada simple. Además, explica paso a paso cómo funciona tu implementación, qué cambios se realizan en los punteros, y analiza en qué situaciones esta operación es útil.

5.4 Inserción en Lista Doblemente Enlazada

Ejercicio 16. Inserta un nodo en una lista doblemente enlazada manteniendo el orden ascendente. Acompaña tu implementación con una explicación lógica del proceso de inserción (ajuste de punteros), y describe qué casos especiales pueden ocurrir (por ejemplo, inserción al inicio, al final o en medio).

5.5 Inversión de Lista Doblemente Enlazada

Ejercicio 17. Implementa una función para invertir una lista doblemente enlazada. Además del código, explica detalladamente el procedimiento, cómo se modifican los punteros en cada paso, y discute posibles ventajas o desventajas respecto a invertir una lista enlazada simple.

Solucionario

Soluciones Selección Múltiple

1. **Respuesta correcta: C**
   Una colisión ocurre cuando dos claves distintas obtienen el mismo resultado al aplicar la función hash, y por tanto intentan ocupar el mismo bucket.

2. **Respuesta correcta: D**
   En una lista enlazada se debe recorrer nodo por nodo hasta encontrar el elemento, lo cual es lineal en el peor caso.

3. **Respuesta correcta: D**
   La búsqueda binaria divide el arreglo ordenado en mitades, reduciendo el espacio de búsqueda logarítmicamente.

4. **Respuesta correcta: B**
   En el encadenamiento separado, cada posición (bucket) en la tabla puede almacenar una lista enlazada para guardar múltiples elementos que colisionan.

5. **Respuesta correcta: D**
   La búsqueda en listas enlazadas es lineal, mientras que en arreglos puedes acceder por índice directamente. Además, las listas no son contiguas en memoria.

Solución Ejercicio 1 — Bubble Sort

```
Arreglo inicial: [5,2,9,1,5,6]

-- Primera pasada --
[2,5,9,1,5,6]
[2,5,9,1,5,6]
[2,5,1,9,5,6]
[2,5,1,5,9,6]
[2,5,1,5,6,9]

-- Segunda pasada --
[2,5,1,5,6,9]
[2,1,5,5,6,9]
[2,1,5,5,6,9]
[2,1,5,5,6,9]
[2,1,5,5,6,9]

-- Tercera pasada --
[1,2,5,5,6,9]
```

Complejidad en el peor caso: $O(n^2)$.

Solución Ejercicio 2 — Selection Sort

```
Arreglo inicial: [6,4,1,3,9]

Pasada 1: [1,4,6,3,9]
Pasada 2: [1,3,6,4,9]
Pasada 3: [1,3,4,6,9]
Pasada 4: [1,3,4,6,9]
```

Selection Sort hace un solo intercambio por pasada, mientras que Bubble Sort puede intercambiar múltiples veces.

Solución Ejercicio 3 — Insertion Sort

```
[3,7,4,9,5] → [3,7,4,9,5]
[3,4,7,9,5]
[3,4,7,9,5]
[3,4,5,7,9]
```

Complejidad en el mejor caso: $O(n)$.

Solución Ejercicio 4 — Merge Sort

1. Divide `[8,3,7]` en `[8]` y `[3,7]`.
2. Divide `[3,7]` en `[3]` y `[7]`.
3. Fusiona `[3]` y `[7]` → `[3,7]`.
4. Fusiona `[8]` y `[3,7]` → `[3,7,8]`.
5. Divide `[4,9,2]` en `[4]` y `[9,2]`.
6. Divide `[9,2]` en `[9]` y `[2]`.
7. Fusiona `[9]` y `[2]` → `[2,9]`.
8. Fusiona `[4]` y `[2,9]` → `[2,4,9]`.
9. Finalmente, fusiona `[3,7,8]` y `[2,4,9]` → `[2,3,4,7,8,9]`.

Solución Ejercicio 5 — Quick Sort

1. Pivote = 10. Partición de `[7,8,9,1,5,10]`:
   - Elementos < 10: `[7,8,9,1,5]`, pivote: 10, mayores: `[]`.
   - Arreglo tras partición: `[7,8,9,1,5,10]`.
2. Llamada recursiva en subarreglo izquierdo `[7,8,9,1,5]`, pivote = 7:
   - Elementos < 7: `[1,5]`, pivote: 7, mayores: `[8,9]`.
   - Arreglo tras partición: `[1,5,7,8,9]`.
3. Llamadas recursivas en `[1,5]` (pivote 5) y en `[8,9]` (pivote 9) dejan estos subarreglos ordenados como `[1,5]` y `[8,9]`.
4. Resultado final: `[1,5,7,8,9,10]`.

Complejidad promedio: $O(n \log n)$.

Solución Ejercicio 6 — Búsqueda Lineal

- 15 vs 8 (1), 3 vs 8 (2), 7 vs 8 (3), 8 vs 8 (4).
- Total: 4 comparaciones.

Complejidad: $O(n)$.

Solución Ejercicio 7 — Búsqueda Binaria

1. Medio = 8: descarta izquierda.
2. Intervalo `[10,12,14]`, medio = 12: descarta derecha.
3. Queda 10: encontrado.

Comparaciones: 3. Complejidad: $O(\log n)$.

Solución Ejercicio 8 — Búsqueda Exponencial + Binaria

- En la fase exponencial, se compara x con los elementos en los índices $1, 2, 4, 8, 16$ (o hasta pasar el tamaño del arreglo o encontrar un valor mayor o igual a x).
- Esto determina un intervalo $[i, j]$ donde x podría encontrarse.
- Luego, se aplica una búsqueda binaria dentro del intervalo $[i, j]$, realizando $\log_2(j-i+1)$ comparaciones en el peor caso.

Solución Ejercicio 9 — Búsqueda Binaria Modificada

```
low=0; high=6; res=-1;
while low <= high:
    mid = (low+high)//2
    if arr[mid] == 7:
        res = mid       % posible respuesta
        high = mid - 1  % busco a la izquierda
    elif arr[mid] < 7:
        low = mid + 1
    else:
        high = mid - 1
return res  % → 2
```

Solución Ejercicio 10 — Encadenamiento Separado

Con $h(x) = x \bmod 5$, calcular para cada elemento:

$$\begin{aligned} 7 \bmod 5 &= 2, \\ 12 \bmod 5 &= 2, \\ 17 \bmod 5 &= 2, \\ 22 \bmod 5 &= 2, \end{aligned}$$

por lo que todos colisionan en el mismo bucket:

$$\text{Bucket}[2] = [7, 12, 17, 22].$$

Solución Ejercicio 11 — Sondeo Lineal

Con $h(x) = x \mod 7$, inserta `[15,22,29,36]`: 15→1, 22→2, 29→3, 36→4.

Solución Ejercicio 12 — Redimensionamiento y Rehashing

1. Inserta hasta 18 en una tabla de tamaño 5:

   $$\text{Elementos: } [3, 8, 13, 18] \;\Rightarrow\; \text{factor de carga } = 4/5 = 0.8 > 0.7$$

   $\longrightarrow$ redimensiona la tabla a tamaño 10.

2. Rehash de cada elemento con $h(x) = x \bmod 10$ utilizando sondeo lineal:

   $$\begin{aligned} 3 \bmod 10 = 3 &\;\to \text{slot 3 libre} \;\Rightarrow\; \text{coloca 3 en } [3], \\ 8 \bmod 10 = 8 &\;\to \text{slot 8 libre} \;\Rightarrow\; \text{coloca 8 en } [8], \\ 13 \bmod 10 = 3 &\;\to \text{slot 3 ocupado} \xrightarrow{+1} \text{slot 4 libre} \;\Rightarrow\; \text{coloca 13 en } [4], \\ 18 \bmod 10 = 8 &\;\to \text{slot 8 ocupado} \xrightarrow{+1} \text{slot 9 libre} \;\Rightarrow\; \text{coloca 18 en } [9]. \end{aligned}$$

3. Inserta 23:

   $$23 \bmod 10 = 3 \;\to\; [3] \text{ ocupado, } [4] \text{ ocupado, } [5] \text{ libre} \;\Rightarrow\; \text{coloca 23 en } [5].$$

Solución Ejercicio 13 — Pila con Lista Enlazada

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.next = None

class Pila:
    def __init__(self):
        self.top = None

    def push(self, x):
        nuevo = Nodo(x)
        nuevo.next = self.top
        self.top = nuevo

    def pop(self):
        if not self.top:
            raise IndexError("Pila vacía")
        val = self.top.dato
        self.top = self.top.next
        return val
# Simulación:
# push(10), push(20), push(30), pop() → 30, pop() → 20
```

Solución Ejercicio 14 — Cola con Lista Enlazada

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.next = None

class Cola:
    def __init__(self):
        self.head = None
        self.tail = None

    def enqueue(self, x):
        nuevo = Nodo(x)
        if not self.head:
            self.head = nuevo
            self.tail = nuevo
        else:
            self.tail.next = nuevo
            self.tail = nuevo
```

```
    def dequeue(self):
        if not self.head:
            raise IndexError("Cola vacía")
        val = self.head.dato
        self.head = self.head.next
        if not self.head:
            self.tail = None
        return val
# Simulación:
# enqueue(5), enqueue(15), enqueue(25), dequeue()→5, enqueue(35), dequeue()→15
```

Solución Ejercicio 15 — Inversión de Lista Enlazada

Explicación: La función recorre la lista una sola vez y va invirtiendo los punteros de cada nodo. Se utiliza una variable `prev` para mantener el nodo anterior, que se convertirá en el nuevo `next` del nodo actual. Esta operación es útil cuando se necesita recorrer la lista en sentido inverso o para algoritmos que requieren modificar la estructura de la lista de forma temporal.

```
def invertir_lista(head):
    prev = None
    curr = head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    return prev
```

—

Solución Ejercicio 16 — Inserción en Lista Doblemente Enlazada

Explicación: Se busca la posición correcta para insertar el nuevo nodo manteniendo el orden ascendente. Hay que manejar tres casos: (1) inserción al inicio, (2) inserción al final, y (3) inserción en el medio. Es fundamental actualizar correctamente tanto los punteros `next` como `prev` del nodo nuevo y de los nodos adyacentes para mantener la estructura doblemente enlazada. Este tipo de operación es común en listas ordenadas o estructuras como colas de prioridad.

```
function insertarNodo(ref lista, x):
    nuevo = crearNodo(x)
    if lista==NULL or x <= lista.dato:
        nuevo.next = lista
```

```
        if lista != NULL:
            lista.prev = nuevo
        lista = nuevo
    else:
        temp = lista
        while temp.next != NULL and temp.next.dato < x:
            temp = temp.next
        nuevo.next = temp.next
        if temp.next != NULL:
            temp.next.prev = nuevo
        temp.next = nuevo
        nuevo.prev = temp
```

—

Solución Ejercicio 17 — Inversión de Lista Doblemente Enlazada

Explicación: El proceso es similar al de la lista simple, pero aquí hay que intercambiar tanto el puntero `next` como el `prev` de cada nodo. Después del intercambio, se avanza usando el puntero `prev` (ya invertido). La variable `temp` se queda apuntando al antiguo `prev` del último nodo procesado, que será el nuevo `head`. Esta inversión es más costosa que la de una lista simple, pero permite recorrer eficientemente la lista en ambas direcciones después de invertirla.

```
function invertirLista(ref lista):
    curr = lista
    temp = NULL
    while curr != NULL:
        temp = curr.prev
        curr.prev = curr.next
        curr.next = temp
        curr = curr.prev
    if temp != NULL:
        lista = temp.prev
```