Tarea Examen 2

Julio César Lozano Garnica. No. de cuenta UNAM: 420095390

19 de septiembre del 2025

Esta es la segunda tarea examen de la materia *Estructuras de Datos* impartida por el profesor Erick Quintero Villeda, el ayudante Diego Jesus Vidal Aguilar, y la ayudante de laboratorio Sandra Valeria Rivera Lara.

1. Definición de los algoritmos, cálculo de operaciones elementales, número de localidades de memoria, complejidades en tiempo y espacio de las implementaciones de *ListaLigadaSimple.*

ALGORITMO 1.1 *eliminar*

Entrada: Una lista l de tipo *ListaLigadaSimple* y un elemento e.

Salida: Una lista de tipo *ListaLigadaSimple*

1. Si (l.cabeza.elemento.equals(e)) es verdadero, entonces ir al paso 2, en otro caso ir al paso 4.
2. l.cabeza = l.cabeza.siguiente
3. l.longitud- -
4. Nodo actual = l.cabeza
5. Mientras (actual.siguiente != null && !actual.siguiente.elemento.equals(e)) hacer:
6. actual = actual.siguiente
7. Terminar mientras
8. actual.siguiente = actual.siguiente.siguiente
9. l.longitud- -

Número de operaciones elementales del algoritmo 1.1 *eliminar*. $T(n) = 10n+20$. Su complejidad es $O(n)$

1. 5 operaciones $O(1)$
2. 4 operaciones $O(1)$
3. 4 operaciones $O(1)$

4. 3 operaciones $O(1)$
5. 8 operaciones por iteración $O(n)$
6. 2 operaciones por iteración $O(n)$
8. 4 operaciones $O(1)$
9. 4 operaciones $O(1)$

Número de localidades de memoria del algoritmo 1.1 *eliminar*. $T(n) = n + 1$. Su complejidad es $O(n)$

1. 1 localidad $O(1)$
4. 1 localidad $O(1)$
5. $n - 1$ localidades $O(n)$

ALGORITMO 1.2 *buscar*

Entrada: Una lista l de tipo *ListaLigadaSimple* y un elemento e.

Salida: Verdadero si e es un elemento de l, falso en otro caso.

1. Si (e == null) es verdadero, entonces ir al paso 2, en otro caso ir al paso 3.
2. Regresar falso
3. Nodo actual = l.cabeza
4. Mientras (actual != null) hacer:
5. Si (actual.elemento.equals(e)) es verdadero ir al paso 6, en otro caso ir al paso 7.
6. Regresar verdadero
7. actual = actual.siguiente
8. Terminar mientras
9. Regresar falso

Número de operaciones elementales del algoritmo 1.2 *buscar*. $T(n) = 6n + 7$. Su complejidad es $O(n)$

1. 2 operaciones $O(1)$
2. 1 operación $O(1)$
3. 3 operaciones $O(1)$
4. 1 operación por iteración $O(n)$
5. 3 operaciones por iteración $O(n)$

7. 2 operaciones por iteración $O(n)$
9. 1 operación $O(1)$

Número de localidades de memoria del algoritmo 1.2 *buscar*. $T(n) = n + 1$. Su complejidad es $O(n)$

3. 1 localidad $O(1)$
5. n localidades $O(n)$

ALGORITMO 1.3 *eliminar*

Entrada: Una lista l de tipo *ListaLigadaSimple* y un entero i.

Salida: Una lista de tipo *ListaLigadaSimple*.

1. Si ($i == 0$) es verdadero, entonces ir al paso 2, en otro caso ir al paso 3.
2. l.cabeza = l.cabeza.siguiente
3. Nodo actual = l.cabeza
4. Para todo $0 \leq$ índice $< i - 1$ hacer:
5. actual = actual.siguiente
6. actual.siguiente = actual.siguiente.siguiente
7. l.longitud - -

Número de operaciones elementales del algoritmo 1.2 *buscar*. $T(n) = 6n + 15$. Su complejidad es $O(n)$

1. 2 operaciones $O(1)$
2. 2 operaciones $O(1)$
3. 3 operaciones $O(1)$
4. 4 operaciones por iteración $O(n)$
5. 2 operaciones por iteración $O(n)$
6. 4 operaciones $O(1)$
7. 4 operaciones $O(1)$

Número de localidades de memoria del algoritmo 1.2 *buscar*. $T(n) = 2$. Su complejidad es $O(1)$

3. 1 localidad $O(1)$
4. 1 localidad $O(1)$

ALGORITMO 1.4 *acceder*

Entrada: Una lista l de tipo *ListaLigadaSimple* y un entero i.

Salida: El i-ésimo elemento de la lista.

1. Si (i <0) es verdadero, entonces ir al paso 2, en otro caso ir al paso 3.
2. Devolver ERROR: El índice proporcionado es negativo, no se puede buscar en la lista.
3. Si (l.longitud $\leq i$) es verdadero ir al paso 4, en otro caso ir al paso 5.
4. Devolver ERROR: La longitud de la lista es menor al índice proporcionado.
5. Nodo actual = l.cabeza
6. int contador = 0
7. Mientras (contador <i) hacer
8. actual = actual.siguiente
9. contador ++
10. Terminar mientras
11. Devolver actual.elemento

Número de operaciones elementales del algoritmo 1.4 *acceder*. $T(n) = 5n + 11$. Su complejidad es $O(n)$

1. 2 operaciones $O(1)$
2. 1 operación $O(1)$
3. 2 operaciones $O(1)$
4. 1 operación $O(1)$
5. 3 operaciones $O(1)$
6. 1 operación por iteración $O(n)$
7. 2 operaciones por iteración $O(n)$
8. 2 operaciones por iteración $O(n)$
10. 2 operaciones $O(1)$

Número de localidades de memoria del algoritmo 1.4 *acceder*. $T(n) = 2$. Su complejidad es $O(1)$

5. 1 localidad $O(1)$
6. 1 localidad $O(1)$

2. Definición de los algoritmos, cálculo de operaciones elementales, número de localidades de memoria, complejidades de tiempo y espacio de las implementanciones del TDA Conjunto con antributo principal un objeto de tipo *ListaLigadaSimple*.

Atributos en la implementación

Decidí no usar otros atributos aparte de elemento: ListaLigadaSimple. Podría añadirse cardinalidad, pero resulta redundante pues ListaLigadaSimple ya mantiene longitud según la nota 8, por lo que no es necesario. Mantener sólo la lista evita redundancia y problemas de sincronización entre longitud y cardinalidad.

ALGORITMO 2.1 *pertenece*

Entrada: Un conjunto c de tipo *ConjuntoListaLigadaSimple* y un elemento e.

Salida: *Verdadero* si e es un elemento del conjunto y *falso* en otro caso.

1. Devolver buscar(c.elemento, e)

Número de operaciones elementales del algoritmo 2.1 *pertenece*. $T(n) = 3$. Su complejidad es $O(1)$

1. 3 operaciones $O(1)$

Número de localidades de memoria del algoritmo 2.1 *pertenece*. $T(n) = 1$. Su complejidad es $O(1)$

1. 1 localidad $O(1)$

ALGORITMO 2.2 *agregarElemento*

Entrada: Un conjunto c de tipo *ConjuntoListaLigadaSimple* y un elemento e.

Salida: El conjunto c con un nuevo elemento e.

1. Si (pertenece(c, e)) es verdadero, entonces ir al paso 2, en otro caso ir al paso 3.
2. Devolver c
3. Nodo nuevo = new Nodo(e)
4. nuevo.siguiente = c.elemento.cabeza
5. c.elemento.cabeza = nuevo
6. c.elemento.longitud = c.elemento.longitud + 1
7. Devolver c

Número de operaciones elementales del algoritmo 2.2 *agregarElemento*. $T(n) = 20$. Su complejidad es $O(1)$

1. 3 operaciones $O(1)$
2. 1 operación $O(1)$
3. 2 operaciones $O(1)$
4. 4 operaciones $O(1)$
5. 3 operaciones $O(1)$
6. 6 operaciones $O(1)$
7. 1 operación $O(1)$

Número de localidades de memoria del algoritmo 2.2 *agregarElemento*. $T(n) = 2$. Su complejidad es $O(1)$

1. 1 localidad $O(1)$
2. 1 localidad $O(1)$

ALGORITMO 2.3 *obtenerCardinalidad*

Entrada: Un conjunto c de tipo *ConjuntoListaLigadaSimple*.

Salida: La cardinalidad del conjunto c.

1. Devolver c.elemento.longitud

Número de operaciones elementales del algoritmo 2.3 *obtenerCardinalidad*. $T(n) = 2$. Su complejidad es $O(1)$

1. 2 operaciones $O(1)$

Número de localidades de memoria del algoritmo 2.3 *obtenerCardinalidad*. $T(n) = 1$. Su complejidad es $O(1)$

1. 1 localidad $O(1)$

ALGORITMO 2.4 *contieneConjunto*

Entrada: Un conjunto c de tipo *ConjuntoListaLigadaSimple* y un conjunto c_2 de tipo *Conjunto*.

Salida: *Verdadero* si c_2 está contenido en c *falso* en otro caso.

1. Para cada x en c_2.elemento hacer:
2. Si !pertenece(c, x) ir al paso 3, en otro caso ir al paso 4.
3. Devolver falso
4. Devolver verdadero

Número de operaciones elementales del algoritmo 2.4 ***contieneConjunto***. $T(n) = 4n + 1$. Su complejidad es $O(n)$

1. 1 operación por iteración $O(n)$
2. 3 operaciones por iteración $O(n)$
4. 1 operación $O(1)$

Número de localidades de memoria del algoritmo 2.4 ***contieneConjunto***. $T(n) = 2$. Su complejidad es $O(1)$

1. 1 localidad $O(1)$
2. 1 localidad $O(1)$

ALGORITMO 2.5 *union*

Entrada: Un conjunto c de tipo *ConjuntoListaLigadaSimple* y un conjunto c_2 de tipo *Conjunto*.

Salida: Un conjunto c_3 de tipo *Conjunto* que resulta ser la unión de c con c_2.

1. c_3 = new ConjuntoListaLigadaSimple<>()
2. Para cada x en c.elemento hacer:
3. agregarElemento (c_3, x)
4. Para cada y en c_2.elemento hacer:
5. agregarElemento (c_3, y)
6. Devolver c_3

Número de operaciones elementales del algoritmo 2.5 ***union***. $T(n) = 4m+4n+3$. Su complejidad es $O(n)$

1. 2 operaciones $O(1)$
2. 1 operación por m iteraciones $O(n)$
3. 3 operaciones por m iteraciones $O(n)$
4. 1 operación por n iteraciones $O(n)$

5. 3 operaciones por n iteraciones $O(n)$
6. 1 operación $O(1)$

Número de localidades de memoria del algoritmo 2.5 *union*. $T(m+n) = 4m + 4n + 4$. Su complejidad es $O(n)$

1. 2 localidades $O(1)$
2. 1 localidad $O(1)$
3. 4 localidades por m iteraciones $O(n)$
4. 1 localidad $O(1)$
5. 4 localidades por n iteraciones $O(n)$

ALGORITMO 2.6 *interseccion*

Entrada: Un conjunto c de tipo *ConjuntoListaLigadaSimple* y un conjunto c_2 de tipo *Conjunto*.

Salida: Un conjunto c_3 de tipo *Conjunto* que resulta ser la intersección de c con c_2.

1. c_3 = new ConjuntoListaLigadaSimple<>()
2. Si c.elemento.longitud $\leq c_2$.elemento.longitud es verdadero ir al paso 3, en otro caso ir al paso 4.
3. base $= c$ y compara $= c_2$
4. base $= c_2$ y compara $= c$
5. Para cada x en base.elemento hacer:
6. Si pertenece(compara, x) es verdadero ir al paso 7.
7. agregarElemento (c_3, x)
8. Devolver c_3

Número de operaciones elementales del algoritmo 2.6 *interseccion*. $T(n) = 7n + 12$. Su complejidad es $O(n)$

1. 2 operaciones $O(1)$
2. 5 operaciones $O(1)$
4. 4 operaciones $O(1)$
5. 1 operación por iteración $O(n)$
6. 3 operaciones por iteración $O(n)$
7. 3 operaciones por iteración $O(n)$
8. 1 operación $O(1)$

Número de localidades de memoria del algoritmo 2.6 *interseccion*. $T(n) = 5n+5$. Su complejidad es $O(n)$

1. 2 localidades $O(1)$
3. 2 localidades $O(1)$
6. 1 localidad $O(1)$
7. 5 localidades por iteración $O(n)$

ALGORITMO 2.7 *iguales*

Entrada: Un conjunto c de tipo *ConjuntoListaLigadaSimple* y un conjunto c_2 de tipo *Conjunto*.

Salida: *Verdadero* si c y c_2 tienen los mismos elementos, *falso* en otro caso.

1. Devolver contieneConjunto(c, c_2) && contieneConjunto(c_2, c)

Número de operaciones elementales del algoritmo 2.7 *iguales*. $T(n) = 8$. Su complejidad es $O(1)$

1. 8 operaciones $O(1)$

Número de localidades de memoria del algoritmo 2.7 *iguales*. $T(n) = 4$. Su complejidad es $O(1)$

1. 2 localidades $O(1)$
2. 2 localidades $O(1)$

3. Ventajas y desventajas de la implementación de conjuntos usando listas y arreglos

Ventajas

- Con la implementación de conjuntos por medio de arreglos, obtener un elemento es muy sencillo, pues únicamente se ingresa el índice del elemento.
- Con la implementación de conjuntos por medio de listas, eliminar o agregar elementos, es muy sencillo pues todo se resume a manejar correctamente las referencias de los nodos que componen las listas.

Desventajas

- Con la implementación de conjuntos por medio de listas, obtener elementos de las listas, es más complejo, pues se tiene que recorrer en el peor de los casos toda la lista, para conocer el elemento buscado.
- Con la implementación de conjuntos por medio de arreglos, eliminar o agregar elementos, puede llegar a ser muy complejo, pues hay que dimensionar el arreglo alterado para obtener los resultados convenientes.

4. Definición del algoritmo, cálculo de operaciones elementales, número de localidades de memoria, complejidades en tiempo y espacio del ejercicio de pares ordenados

---

ALGORITMO 3.1 *amistadesReciprocas*

Entrada: S un subconjunto de R de tipo *ConjuntoListaLigadaSimple* con elementos de pares ordenados (x, y).

Salida: *Verdadero* si para todos los elementos (x, y) en S existe (y, x) en S, *falso* en otro caso.

1. Para cada par u en S.elemento hacer:
2. v = (u.segundoElemento, u.primerElemento)
3. Si !pertenece(S, v) ir al paso 4, en otro caso ir al paso 5
4. Devolver falso
5. Devolver verdadero

Número de operaciones elementales del algoritmo 3.1 *amistadesReciprocas*. $T(n) = 7n + 1$. Su complejidad es $O(n)$

1. 1 operación por iteración $O(n)$
2. 4 operaciones por iteración $O(n)$
3. 2 operaciones por iteración $O(n)$
5. 1 operación $O(1)$

Número de localidades de memoria del algoritmo 3.1 *amistadesReciprocas*. $T(n) = 5n$. Su complejidad es $O(n)$

1. 1 localidad por iteración $O(n)$
2. 3 localidades por iteración $O(n)$
3. 1 localidad por iteración $O(n)$
