Algoritmos y Estructuras de Datos

Guía Práctica 7
Diseño: Elección de estructuras

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ejercicio 1. Confeccione una tabla comparativa de las complejidades de peor caso de las operaciones de pertenencia, inserción, borrado, búsqueda del mínimo y borrado del mínimo para conjuntos de naturales sobre las siguientes estructuras:

1. Lista enlazada
2. Lista enlazada ordenada
3. Árbol binarios de búsqueda
4. Árbol AVL
5. *heap*
6. *trie*

Ejercicio 2. Escriba algoritmos para las operaciones $\cap$ y $\cup$, considerando las siguientes representaciones para conjuntos:

1. Sobre listas enlazadas (ordenadas y no ordenadas).
2. Sobre árboles binarios de búsqueda.
3. Sobre *heaps*.

En cada inciso, calcule los órdenes de complejidad de peor caso de las operaciones.

Ejercicio 3. El TAD *Matriz infinita de booleanos* tiene las siguientes operaciones:

- *Crear*, que crea una matriz donde todos los valores son falsos.
- *Asignar*, que toma una matriz, dos naturales (fila y columna) y un booleano, y asigna a este último en esa coordenada. (Como la matriz es infinita, no hay restricciones sobre la magnitud de fila y columna.)
- *Ver*, que dadas una matriz, una fila y una columna devuelve el valor de esa coordenada. (Idem.)
- *Complementar*, que invierte todos los valores de la matriz.
- *Intersecar*, que toma dos matrices y realiza la conjunción coordenada a coordenada.

Diseñe este TAD de modo que las operaciones *Crear*, *Ver* y *Complementar* tomen $O(1)$ tiempo en peor caso.

Ejercicio 4. Una matriz finita posee las siguientes operaciones:

- *Crear*, con la cantidad de filas y columnas que albergará la matriz.
- *Definir*, que permite definir el valor para una posición válida.
- *#Filas*, que retorna la cantidad de filas de la matriz.
- *#Columnas*, que retorna la cantidad de columnas de la matriz.
- *Obtener*, que devuelve el valor de una posición válida de la matriz (si nunca se definió la matriz en la posición solicitada devuelve cero).
- *SumarMatrices*, que permite sumar dos matrices de iguales dimensiones.

Diseñe un módulo para el TAD *MatrizFinita* de modo tal que dadas dos matrices finitas A y B,

$*$ *Definir* y *Obtener* aplicadas a A se realicen cada una en $\Theta(n)$ en peor caso, y

$*$ *SumarMatrices* aplicada a A y B se realice en $\Theta(n+m)$ en peor caso,

donde n y m son la cantidad de elementos no nulos de A y B, respectivamente.

Ejercicio 5. Considere el TAD *Diccionario con historia*, cuya especificación es la siguiente:

```
TAD DiccionarioConHistoria<K, V> {
    obs data: dict<K, V>
    obs cant: dict<K, int>

    proc nuevoDiccionario(): DiccionarioConHistoria<K, V>
        asegura res.data == {}
        asegura res.cant == {}

    proc esta(in d: DiccionarioConHistoria<K, V>, in k: K): bool
        asegura res <==> k in d.data

    proc definir(inout d: DiccionarioConHistoria<K, V>, in k: K, in v: V)
        asegura d.data == setKey(old(d).data, k, v)
        asegura d.cant == setKey(old(d).cant, k,
            if k in old(d).cant then old(d).cant[k] + 1 else 1 fi
        )

    proc obtener(in d: DiccionarioConHistoria<K, V>, in k: K): V
        requiere k in d.data
        asegura v == d.data[k]

    proc borrar(inout d: DiccionarioConHistoria<K, V>, in k: K): V
        requiere k in d.data
        asegura d.data == delKey(old(d).data, k)
        asegura d.cant == old(d).cant

    proc cantSignificados(in d: DiccionarioConHistoria<K, V>, in k: K): int
        requiere k in d.data
        asegura res == d.cant[k]
}
```

Se debe diseñar este TAD respetando los siguientes órdenes de ejecución en el peor caso:

- `esta` $O(n)$
- `obtener` $O(n)$
- `borrar` $O(n)$
- `nuevoDiccionario` $O(1)$
- `definir` $O(n)$
- `cantSignificados` $O(n)$

donde n es la cantidad de claves que están definidas en el diccionario.

Ejercicio 6. Dada la siguiente implementación basada en Arboles Binarios, dar algoritmos para resolver las siguientes operaciones, y calcular su orden de ejecución en peor caso:

```
Nodo<T> es struct<
    izq: Nodo,
    der: Nodo,
    dato: T
>

Impl ArbolBinario<T> {
    var raiz: Nodo<T>

    proc nivelCompleto(a: ArbolBinario<T>, n: int): bool
        requiere hayCaminoDeAltura(n)

    proc estaCompleto(in a: ArbolBinario<T>): bool

    pred hayCaminoDeAltura(a: ArbolBinario<T>, n): int {
```

```
            exists s: seq<Nodo> :: esCamino(a, s) && |s| == n
        }

        pred esCamino(a: ArbolBinario<T>, s: seq<Nodo>) {
            |s| == 0 || (
                s[0] == raiz &&
                s[|s|-1].izq == nil &&
                s[|s|-1].der == nil &&
                forall i: int :: 0 <= i < |s|-1 ==> s[i].izq == s[i+1] || s[i].der == s[i+1]
            )
        }
    }
```

Ejercicio 7. Diseñe el TAD *Cola de prioridad*, escribiendo los algoritmos e indicando sus órdenes de complejidad de peor caso sobre las siguientes estructuras:

1. Árbol binario con invariante de *heap*.
2. Arreglo con invariante de *heap*.

Ejercicio 8. Diseñar un conjunto de naturales que satisfaga los siguientes requerimientos de complejidad temporal:

- Ag debe costar tiempo $O(1)$ si el elemento a agregar es menor que el mínimo elemento del conjunto hasta el momento o mayor que el máximo, y tiempo lineal en la cantidad de elementos del conjunto en cualquier otro caso.
- $\cup$ y $\cap$ deben tomar tiempo $O(1)$ si cada elemento de uno de los conjuntos pasados como parámetro es mayor que todos los del otro, y tiempo $O(n + m)$ en cualquier otro caso, siendo n y m las cardinalidades de los dos conjuntos pasados como parámetros.

1. Elegir una estructura adecuada, justificando la decisión, y escribir los algoritmos para las tres operaciones anteriores. Documentar claramente los aspectos de aliasing.
2. ¿Puede usarse este mismo diseño para conjuntos que no sean de naturales? ¿Qué condición debe cumplir el tipo al que pertenecen los elementos para poder usar este módulo?

Ejercicio 9. Diseñar las relaciones binarias entre números naturales. Para una relación $\mathcal{R}$, se quiere preguntar:

- Dado un número a, cuáles son los números b tales que $a\mathcal{R}b$.
- Dado un número a, cuáles son los números b tales que $b\mathcal{R}a$.

Las relaciones se construyen por extensión, es decir, agregando y borrando parejas de la relación.

1. Suponiendo que las relaciones son entre números de 0 a 100, elegir estructuras de datos adecuadas. Justificar. Dar el invariante de representación y la función de abstracción de las estructuras. Implementar las operaciones sobre las estructuras elegidas. Indicar el orden de complejidad de las operaciones.
2. Elegir las estructuras de datos adecuadas si el rango sobre el cual se una relación será fijado al momento de la creación de la misma. Justificar.
3. Elegir las estructuras de datos adecuadas si las relaciones son entre naturales cualesquiera (i.e., no hay un rango prefijado), de manera que ninguna operación tenga un costo demasiado alto. Justificar.

Ejercicio 10. Se desea diseñar un sistema de estadísticas para la cantidad de personas que ingresan a un banco. Al final del día, un empleado del banco ingresa en el sistema el total de ingresantes para ese día. Se desea saber, en cualquier intervalo de días, la cantidad total de personas que ingresaron al banco. La siguiente es una especificación del problema.

```
TAD IngresosAlBanco {
    obs totales: seq<int>

    proc nuevoIngresos(): IngresosAlBanco
        asegura totalDia == []

    proc registrarNuevoDia(inout i: IngresosAlBanco, in cant: int)
        requiere cant >= 0
        asegura i.totales == old(i).totales + [cant]

    proc cantDias(in i: IngresosAlBanco): int
        asegura res == |i.totales|

    proc cantPersonas(in i: IngresosAlBanco, in desde: int, in hasta: int): int
        requiere 0 <= desde <= hasta <= |i.totales|
        asegura res == sum i: int :: if desde <= i < hasta then i.totales[i] else 0 fi
}
```

1. Dar una estructura de representación que permita que la función *cantPersonas* tome $O(1)$.
2. Calcular cuánta memoria usa la estructura, en función de la cantidad de días que pasaron n.
3. Si el cálculo del punto anterior fue una función que no es $O(n)$, piense otra estructura que permita resolver el problema utilizando $O(n)$ memoria.
4. Agregue al diseño del punto anterior una operación *mediana* que devuelva el último (mayor) día d tal que cantPersonas$(i, 1, d)$ $\leq$ cantPersonas$(i, d+1$,totDias$(i))$, restringiendo la operación a los casos donde dicho día existe. Si este ítem no sale, se recomienda dejarlo pendiente hasta avanzada la práctica de dividir y conquistar. No olvidar retomarlo luego.

Ejercicio 11. Considere el siguiente TAD que describe la estructura de un texto:

```
Palabra es string

TAD Texto {
    obs palabras: seq<Palabra>

    proc nuevoTexto(): Texto
        asegura palabras == []

    proc agregarPalabra(inout t: Texto, p: Palabra)
        asegura t.palabras == old(t).palabras + [p]

    proc cambiarPalabra(inout t: Texto, in vieja: Palabra, in nueva: Palabra)
        asegura forall i: int :: 0 <= i < |t.palabras| ==>L
            (t.palabras[i] != vieja ==> t.palabras[i] == old(t).palabras[i]) &&
            (t.palabras[i] == vieja ==> t.palabras[i] == nueva)

    proc posiciones(in t: Texto, in p: Palabra): Conjunto<Palabra>
        asegura forall i: int ::
            i in res <==> 0 <= i < |t.palabras| &&L t.palabras[i] == p

    proc subtexto(in t: Texto, in desde: int, in hasta: int): Texto
        requiere 0 <= desde <= hasta < |t.palabras|
        asegura res.palabras == t.palabras[desde..hasta]

    proc masRepetidas(in t: Texto): Conjunto<Palabra>
        asegura forall p: Palabra :: p in res <==> masApariciones(t.palabras, p)
```

```
        pred masApariciones(s: seq<Palabra>, p: Palabra) {
            !exists p': Palabra :: apariciones(s, p') > apariciones(s, p)
        }
    }
```

Se desea diseñar el módulo correspondiente al TAD texto. En particular, asumimos que trabajaremos sólo con textos en español, y por lo tanto podemos dar una cota para la longitud de la palabra más larga que puede aparecer en el texto.

Se requiere que las operaciones que se listan a continuación cumplan con la complejidad temporal indicada:

- `subtexto(in t: Texto, in desde: int, in hasta: int): Texto`

  Devuelve el texto correspondiente al fragmento de t que comienza en la posición *desde* y finaliza en la posición *hasta*.

  $O(hasta - desde)$ en el peor caso

- `cambiarPalabra(inout t: Texto, in vieja: Palabra, in nueva: Palabra)`

  Cambia todas las ocurrencias en el texto de la palabra anterior por la nueva.

  $O(k)$ en el peor caso, donde k es la cantidad de veces que se repite la palabra a cambiar.

- `masRepetidas(in t: Texto): Conjunto<Palabra>` Devuelve el conjunto de palabras que más se repiten en el texto.

  $O(1)$ en el peor caso. Puede generar aliasing.

a) Describir la estructura a utilizar, documentando claramente cómo la misma resuelve el problema y cómo cumple con los requerimientos de eficiencia. El diseño debe incluir sólo la estructura de nivel superior. Para justificar los órdenes de complejidad, describa las estructuras soporte. **Importante**: si alguna de las estructuras utilizadas requiere que sus elementos posean una función especial (por ejemplo, comparación) deberá describirla.

b) Escribir el Rep y Abs.

c) Escribir el algoritmo para `cambiarPalabra`. Justifique la complejidad sobre el código.

d) Escribir el algoritmo para `subtexto`. Justifique la complejidad sobre el código.

e) Escribir el algoritmo para `agregarPalabra`.

Ejercicio 12. Se desea diseñar un sistema para manejar el ranking de posiciones de un torneo deportivo. En el torneo hay un conjunto fijo de equipos que juegan partidos (posiblemente más de un partido entre cada pareja de equipos) y el ganador de cada partido consigue un punto. Para el ranking, se decidió que entre los equipos con igual cantidad de puntos no se desempata, sino que todos reciben la mejor posición posible para ese puntaje. Por ejemplo, si los puntajes son: A: 5 puntos, B: 5 puntos, C: 4 puntos, D: 3 puntos, E: 3 puntos, las posiciones son: A: 1ro, B: 1ro, C: 3ro, D: 4to, E: 4to.

El siguiente TAD es una especificación para este problema.

```
Equipo es int
Partido es struct<ganador: Equipo, perdedor: Equipo>

TAD Torneo {
    obs equipos: conj<Equipo>
    obs partidos: seq<Partido>

    proc nuevoTorneo(): Torneo
        asegura res.equipos == {}
        asegura res.partidos == []

    proc registrarPartido(inout t: Torneo, in ganador: Equipo, in perdedor: Equipo)
        asegura t.partidos == old(t).partidos + [<ganador: ganador, perdedor: perdedor>]
        asegura t.equipos == old(t).equipos + {ganador, perdedor}

    proc posicion(in t: Torneo, in e: Equipo): int
        requiere e in t.equipos
        asegura res == cantConMasPuntos(t, puntosDe(t, e)) + 1
```

```
    proc puntos(in t: Torneo, in e: Equipo): int
        requiere e in t.equipos
        asegura res == t.puntosDe(e)

    proc masPuntos(in t: Torneo): Conjunto<Equipo>
        asegura forall e: Equipo :: e in res <==> t.cantConMasPuntos(puntosDe(t, e)) == 0

    aux puntosDe(t: Torneo, e: Equipo): int =
        sum i: int :: if 0 <= i < t.partidos && t.partidos[i].ganador == e then 1 else 0 fi

    aux cantConMasPuntos(t: Torneo, p: int): int =
        sum e: Equipo :: if e in t.equipos && puntosDe(t, e) > p then 1 else 0 fi

}
```

Se desea diseñar el sistema propuesto, teniendo en cuenta que las operaciones `puntos`, `registrarPartido` y `posicion` deben realizarse en $O(\log n)$, donde n es la cantidad de equipos registrados.

a) Describir la estructura a utilizar.

b) Escribir un pseudocódigo del algoritmo para las operaciones con requerimientos de complejidad.

Ejercicio 13. Se nos encargó el diseño del nuevo Sistema Unificado de Estadísticas para Colectivos y Omnibuses (SUECO). Dicho sistema recolecta los datos de todos los boletos pagados mediante el sistema electrónico y genera estadísticas de la cantidad de boletos expedidos y la cantidad de plata electrónica que ingresó al sistema en un determinado período. Al agregar un boleto al sistema se especifica su precio y además un natural que representa el momento en que fue sacado ese boleto. Dado que el sistema de pago electrónico no se sincroniza al instante, es posible que los boletos se agreguen al sistema no ordenados cronológicamente. Al consultar, siempre se da un intervalo de tiempo y se debe responder la cantidad de boletos expedidos o plata recaudada en el intervalo propuesto.

El siguiente TAD es una especificación para este problema.

```
Tiempo es int
Plata es int

TAD SUECO {
    obs boletos: dict<Tiempo, int>
    obs plata: dict<Tiempo, Plata>

    proc iniciarDia(): SUECO
        asegura res.boletos == {}
        asegura res.plata == {}

    proc agregarBoleto(inout s: SUECO, t: Tiempo, p: Plata)
        asegura s.boletos == setKey(
            old(s).boletos, t,
            if t in old(s).boletos then old(s).boletos[t] else 0 fi + 1
        )
        asegura s.plata == setKey(
            old(s).plata, t,
            if t in old(s).plata then old(s).plata[t] else 0 fi + p
        )

    proc cantidadIntervalo(in s: SUECO, inicio: Tiempo, fin: Tiempo): int
        asegura res == sum t: Tiempo :: t in s.boletos && inicio <= t <= fin :: s.boletos[t]

    proc plataIntervalo(in s: SUECO, inicio: Tiempo, fin: Tiempo): int
        asegura res == sum t: Tiempo :: t in s.plata && inicio <= t <= fin :: s.plata[t]
```

```
}
```

Se pide dar una estructura de representación que permita realizar la operación *agregarBoleto* en $O(n)$, y *#Intervalo* en $O(\log n)$, dónde n es la cantidad de boletos agregados hasta ese momento.

a) Describir la estructura a utilizar.

b) Para ambas operaciones, explicar como se usa la estructura de manera de cumplir con la complejidad pedida (no hace falta dar pseudocódigo de cada operación, pero se puede darlo si resulta conveniente para la explicación).

c) Agregar lo que sea necesario para proveer la operación *plataIntervalo* también en $O(\log n)$. Describa las modificaciones a la estructura, a las operaciones del punto anterior que precisen modificaciones, y la forma de implementar *plataIntervalo*.