Algoritmos y Estructuras de Datos

Guía Práctica 7.5
Diseño: Elección de estructuras (parte 2)

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

Diseño con listas enlazadas y árboles

En esta práctica nos concentraremos en la elección de estructuras y los algoritmos asociados. Cuando el enunciado diga "diseñe este módulo", salvo que indique lo contrario, sólo nos interesa: 1. la estructura, 2. las operaciones con sus parámetros y su complejidad (no los Requiere y Asegura) y 3. los algoritmos correspondientes.

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

NOTA: Este ejercicio es conceptual, no hay que usar módulos básicos sino imaginar los algoritmos y analizar sus complejidaded en caso de que utilizáramos las estructuras mencionadas para implementar los conjuntos.

Ejercicio 2. Se desea diseñar un sistema para registrar las notas de los alumnos en una facultad. Al igual que en Exactas, los alumnos se identifican con un número de LU. A su vez, las materias tienen un nombre, y puede haber una cantidad no acotada de materias. En cada materia, las notas están entre 0 y 10, y se aprueban si la nota es mayor o igual a 7.

```
TAD Sistema {
      proc RegistrarMateria(inout s: Sistema, in m: materia)

      proc RegistrarNota(inout s: Sistema, in m: materia, in a: alumno, in n: nota)

      proc NotaDeAlumno(in s: Sistema, in a: alumno, m: materia): nota

      proc CantAlumnosConNota(in s: Sistema, in m: materia, n: nota): ℤ

      proc CantAlumnosAprobados(in s: Sistema, in m: materia): ℤ
}
```

Dados $m = cantmaterias$ y $n = cantalumnos$ se desea diseñar un módulo con los siguientes requerimientos de complejidad temporal:

- RegistrarMateria en O(log m)
- RegistrarNota en O(log n + log m)
- NotaDeAlumno en O(log n + log m)
- CantAlumnosConNota y CantAlumnosAprobados en O(log m)

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.

Elija la estructura y escriba los algoritmos 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.

Dado n y m son la cantidad de elementos no nulos de A y B, respectivamente, diseñe un módulo (elegir una estructura y escribir los algoritmos) para el TAD *MatrizFinita* de modo tal que dadas dos matrices finitas A y B,

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

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

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, suma1(k, old(d).cant)}

      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 == delKey(old(d).cant, k)}

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

      aux suma1(k:K, cant: dict<K, int>)
         { if (k in cant) then (cant[k] + 1) else 1 fi }
}
```

1. Si n es la cantidad de claves que están definidas en el diccionario, se debe diseñar este TAD respetando los siguientes órdenes de ejecución en el peor caso:

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

2. Si quisieramos conservar la historia de cada clave cuando se borran y luego vuelve a definir, ¿es posible mantener las complejidades de todas las operaciones? Si es posible, explique las modificaciones necesarias a la estructura. Si no, explique las modificaciones necesarias para que cambien la menor cantidad posible de complejidades, cuáles cambian y por qué.

Ejercicio 6. 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<ℤ>

    proc nuevoIngresos(): IngresosAlBanco
      asegura {totales = []}

    proc registrarNuevoDia(inout i: IngresosAlBanco, in cant: ℤ)
      requiere { cant ≥ 0}
      asegura {i.totales = old(i).totales + [cant]}

    proc cantDias(in i: IngresosAlBanco): ℤ
      asegura {res = |i.totales|}

    proc cantPersonas(in i: IngresosAlBanco, in desde: ℤ, in hasta: ℤ): ℤ
      requiere { 0 ≤ desde ≤ hasta < |i.totales|}
      asegura {res = ∑_{j=desde}^{hasta} i.totales[j]}
}
```

1. Dar una estructura de representación que permita que la función *cantPersonas* tome $O(1)$.
2. Calcular cómo crece el tamaño de la estructura en función de la cantidad de días que pasaron.
3. Si el cálculo del punto anterior fue una función que no es $O(n)$ en memoria, piense otra estructura que permita resolverlo con esa complejidad.
4. Agregue al diseño del punto anterior una operación *mediana* que devuelva el último (mayor) día d tal que cantPersonas$(i, 0, d)$ $\leq$ cantPersonas$(i, d + 1$,cantDias$(i))$, restringiendo la operación a los casos donde dicho día existe.

Diseño con Heaps

Ejercicio 7. Repasemos la implementación de ColaDePrioridad acotada utilizando heaps implementada con arreglos.

1. Escriba en castellano el invariante de representación.
2. Escriba los algoritmos para las operaciones `apilar`, `desapilarMax` y `cambiarPrioridad`. Justifique la complejidad de cada operación.
3. Escriba un algoritmo que verifique si un arreglo de números representa un heap.

Ejercicio 8. ¿Cómo implementaría una ColaDePrioridad no acotada sobre heap en arreglos? Descríba los cambios con palabras o escribiendo el pseudocódigo de las operaciones que cambian.

Ejercicio 9. ¿Cómo haría para implementar una ColaDePrioridad ordenada por dos criterios? Por ejemplo, se quiere tener una cola de personas donde el criterio de ordenamiento es por edad y, en caso de empate, por sexo. Describa todos los cambios necesarios.

Ejercicio 10. ¿Cómo utilizaría un heap para ordenar una secuencia de elementos? Escriba el algoritmo y calcule su complejidad.

Ejercicio 11. Un heap k-ario (*k-ary heap* o *d-ary heap*) es una generalización del heap tradicional en la cuál cada nodo tiene a lo sumo k hijos. La propiedad de heap se mantiene, es decir, el valor de cada nodo es mayor o igual que el valor de sus hijos.

1. Diseñe el módulo `HeapKArio`. Elija la estructura y escriba los algoritmos
2. Calcule la complejidad de las operaciones
3. ¿Qué diferencias le vé respecto del heap tradicional? ¿Qué ventajas y desventajas observa?

Ejercicio 12. Escriba un algoritmo que resuelva el siguiente problema usando heaps. Intente resolverlo con la menor complejidad posible: Dado un arreglo de n números enteros y un entero positivo k, encuentre los k elementos del arreglo con mayor frecuencia. Si dos números tienen la misma frecuencia, el número con mayor valor debe ser priorizado. Se asume que el arreglo tiene al menos k elementos.

Diseño con Tries

Ejercicio 13. Implemente un Trie utilizando arreglos y listas enlazadas para los nodos.

- Describa en castellano el invariante de representación
- Escriba los algoritmos para las operaciones `buscar` y `agregar` y justifique la complejidad de cada operación.
- ¿Qué diferencias observa entre ambas implementaciones? ¿Qué ventajas y desventajas tiene cada una? En qué casos utilizaría cada una?

Ejercicio 14. Utilizando una estructura de Trie para almacenar palabras, escriba los siguientes algoritmos, justificando la complejidad de peor caso de cada uno:

1. `primeraPalabra`: Devuelve la primera palabra en orden lexicográfico.
2. `ultimaPalabra`: Devuelve la última palabra en orden lexicográfico.
3. `palabrasConPrefijo`: Devuelve todas las palabras que tienen un determinado prefijo, ordenadas en orden lexicográfico.

Ejercicio 15. Considere el siguiente caso de uso: necesitamos un trie que, sabemos, se va a modificar constantemente, insertando y borrando un gran número de elementos en un corto período de tiempo. Para optimizar nuestra implementación del módulo Trie en este contexto, decidimos modificar la operación `borrar` para que no elimine todos los nodos intermedios que ya no son necesarios.

Luego, cada cierto período de tiempo, ejecutaremos una nueva operación llamada `limpiar` que deberá eliminar todos los nodos intermedios que no están siendo usados (es decir, que no forman parte de un camino que termina en un elemento del Trie).

1. Describa en castellano los cambios al invariante de representación (si los hubiera)
2. Escriba los nuevos algoritmos para las operaciones `buscar` y `borrar`
3. Escriba el algoritmo para la operación `limpiar`

Ejercicio 16. Dada una matriz binaria (de unos y ceros), identifique si existen filas repetidas. Debe realizar esto recorriendo la matriz sólo una vez.

Ejemplo:

```
Entrada:
[1  0  0  1  0]
[0  1  1  0  0]
[1  0  0  1  0]
[0  0  1  1  0]
[0  1  1  0  0]

Salida:
{2, 4} (la fila 2 es igual a la fila 0 y la fila 4 es igual a la fila 1)
```

Ejercicio 17. Tenemos un conjunto de palabras escritas en notación "camel case" (por ejemplo `BuscarPalabra` o `DevolverMinimoDelConjunto`). A partir de una secuencia de letras en mayúsculas, se quiere buscar todas las palabras del conjunto que tengan ese patrón.

Ejemplo:

```
Entrada: [BuscarPalabra, BuscarPalabraMasLarga, BuscarPalabraMasCorta, SacarTodo, BuscarTodo]

Si el patrónn es BPM, la salida debería ser [BuscarPalabraMasLarga, BuscarPalabraMasCorta]
Si el patrón es S, la salida debería ser [SacarTodo]
Si el patrón es B, la salida debería ser [BuscarPalabra, BuscarPalabraMasLarga,
    BuscarPalabraMasCorta, BuscarTodo]
```

Elección de estructuras (heaps y tries)

Ejercicio 18. Extienda la tabla confeccionada en el ejercicio 1 agregando heaps y tries

Ejercicio 19. La Maderera San Blas vende, entre otras cosas, listones de madera. Los compra en aserraderos de la zona, los cepilla y acondiciona, y los vende por menor del largo que el cliente necesite.

Tienen un sistema un poco particular y ciertamente no muy eficiente: Cuando ingresa un pedido, buscan el listón más largo que tiene en el depósito, realizan el corte del tamaño que el cliente pidió, y devuelven el trozo que queda al depósito.

Por otra parte, identifican a cada cliente con un código alfanumérico de 10 dígitos y cuentan con un fichero en el que registran todas las compras que hizo cada cliente (con la fecha de la compra y el tamaño del listón vendido).

Este sería el TAD simplificado del sistema:

```
Cliente es string

TAD Maderera {
    comprarUnListon(inout m: Maderera, in tamaño: int)
        // comprar en el aserradero un listón de un determinado tamaño

    venderUnListon(inout m: Maderera, in tamaño: int, in cli: Cliente, in f: Fecha)
        // vender un listón de un determinado tamaño a un cliente particular en una fecha determinada

    ventasACliente(in m: Maderera, cli: in Cliente): Conjunto<Tupla<fecha,int>>
        // devolver el conjunto de todas las ventas que se le hicieron a un cliente
        // (para cada venta, se quiere saber la fecha y el tamaño del listón)
}
```

Se pide:

- Escriba una estructura que permita realizar las operaciones indicadas con las siguientes complejidades:
  - comprarUnListon en $O(\log(m))$
  - venderUnListon en $O(\log(m))$
  - ventasAClient en $O(1)$

  donde m es la cantidad de pedazos de listón que hay en el depósito
- Escriba el algoritmo para la operación `venderUnListon`

Ejercicio 20. Se quiere implementar el TAD Biblioteca que modela una biblioteca con su colección de libros. Se modela la biblioteca como una sola estantería de capacidad arbitraria, dentro de la cual cada libro ocupa una posición. La biblioteca cuenta con un registro de socios que pueden retirar y devolver libros en cualquier momento. Por restricciones del sistema que se utiliza, un socio no puede registrarse con un nombre de más de 50 caracteres.

Cuando la biblioteca adquiere un nuevo libro o cuando un libro es devuelto, éste es insertado en el primer espacio libre de la estantería. Es decir, si los lugares ocupados son 1, 2, 3, 4 y se presta el libro en la posición 2, al agregar un nuevo libro al catálogo éste será ubicado en la posición 2. Cuando el libro que estaba originalmente en la posición 2 sea devuelto, será ubicado en la primera posición libre, que será la 5.

El TAD tiene las siguientes operaciones, para las que se nos piden las complejidades temporales de peor caso indicadas, donde

- L es la cantidad de libros en la colección
- r es la cantidad de libros que el socio en cuestión tiene retirados
- k la cantidad de posiciones libres en la estantería

`proc AgregarLibroAlCatálogo(inout b: Biblioteca, in l: idLibro)`

Requiere: {`l` no pertenece a la colección de libros de `b`}

Descripción: la biblioteca adquiere un nuevo libro, lo suma a su catálogo y lo pone en la estantería en el primer espacio disponible.

Complejidad: $O(log(k) + log(L))$

`proc PedirLibro(inout b: Biblioteca, in l: idLibro, in s: Socio)`

Requiere: {`s` es socio de la biblioteca y el libro `l` no está entre los libros prestados}

Descripción: el socio pasa a retirar un libro que se retira de la estantería y se acumula en sus libros prestados.

Complejidad: $O(log(r) + log(k) + log(L))$

`proc DevolverLibro(inout b: Biblioteca, in l: idLibro, in s: Socio)`

Requiere: {`s` es socio de la biblioteca y el libro `l` está entre sus libros prestados}

Descripción: el socio pasa a devolver un libro que previamente había tomado prestado. Vuelve a la estantería en el primer espacio disponible.

Complejidad: $O(log(r) + log(k) + log(L))$

`proc Prestados(in b: Biblioteca, in s: Socio): Conjunto<Libro>`

Requiere: {`s` es socio de la biblioteca}

Descripción: este procedimiento retorna los libros que el socio tomó prestados de la biblioteca y aún no devolvió.

Complejidad: $O(1)$

- `proc UbicaciónDeLibro(in b: Biblioteca, in l: idLibro): Posicion`

  Requiere: {`l` pertenece a la colección de libros de `b` y no está prestado}

  Descripción: obtiene la posición del libro en la estantería.

  Complejidad: $O(log(L))$

Los datos se van a guardar en la siguiente estructura incompleta

```
Socio es string; Libro, Posición es int

Módulo BibliotecaImpl implementa Biblioteca <
   var socios: Diccionario<Socio, Conjunto<Libro>> //Socios y los libros que tienen prestados
   var catálogo: Diccionario<Libro, Posición> //Libros y su posición en la estantería
   var posicionesLibres: Conjunto<Posición> //Posiciones vacías en la estantería
>
```

1. Definir qué estructura hay que implementar los diccionarios y conjuntos para cumplir las complejidades de las operaciones
2. Escribir los algoritmos mostrando cómo se cumplen las cotas de complejidad requeridas.

Ejercicio 21. Se quiere implementar el `TAD Agenda` que modela una agenda semanal donde se registran actividades. Cada actividad tiene un identificador, un horario de inicio y uno de finalización. No puede haber dos actividades con el mismo identificador. Para simplificar, las actividades sólo pueden comenzar y terminar en horarios en punto (por ejemplo, 21:00 hs) y terminan en un horario posterior a su inicio. Tampoco pueden empezar y terminar en días diferentes. Para contar la cantidad de actividades que transcurren en un determinado horario no se debe tener en cuenta aquellas que finalizan en ese momento. En cada actividad se pueden agregar tags que permiten agrupar las distintas actividades por temáticas. Los tags tienen cómo máximo 20 caracteres.

Dadas las siguientes operaciones y de acuerdo a las complejidades temporales de peor caso indicadas, donde

- d es la cantidad de días registrados hasta el momento
- a la cantidad total de actividades

```
proc RegistrarActividad(inout ag: Agenda, in act: IdActividad, in día: Día,
                        in inicio: Hora, in fin: Hora)
```

Requiere: la hora de fin sea mayor que la de inicio y la actividad no está actualmente registrada.

Descripción: se agrega la actividad a la agenda, marcando en el día indicado la hora de inicio y finalización.

Complejidad: $O(\log(a) + \log(d))$

```
proc VerActividad(in ag: Agenda, in act: IdActividad):
                struct<día: Día, inicio: Hora, fin: Hora>
```

Requiere: la actividad debe estar registrada en la agenda.

Descripción: se devuelven el día y horario en que se realiza la actividad.

Complejidad: $O(\log(a))$.

```
proc AgregarTag(inout ag: Agenda, in act: IdActividad, in t: Tag)
```

Requiere: la actividad está registrada en la agenda y aún no tiene registrado ese tag.

Descripción: se agrega el tag a la actividad indicada.

Complejidad: $O(1)$.

```
proc HoraMásOcupada(in ag: Agenda, in d: Día): Hora
```

Requiere: el día indicado tiene al menos una actividad registrada.

Descripción: se devuelve la hora que tiene más actividades registradas.

Complejidad: $O(\log(d))$.

```
proc ActividadesPorTag(in ag: Agenda, in t: Tag): Conjunto<IdActividad>
```

Requiere: true.

Descripción: se devuelven las actividades que tienen registrado el tag t.

Complejidad: $O(1)$.

1. Definir la estructura de representación del módulo `AgendaImpl`, que provea las operaciones solicitadas.

2. Escribir en castellano el invariante de representación.

3. Escribir los algoritmos de todas las operaciones, explicando cómo se cumplen las complejidades pedidas para cada una.

Ejercicio 22. 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 {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 {(∀ e: Equipo)(e ∈ res ↔ t.cantConMasPuntos(puntosDe(t, e)) = 0) }

   aux puntosDe(t: Torneo, e: Equipo): int
         {∑_{i∈ℤ} (if 0 ≤ i < t.partidos ∧ t.partidos[i].ganador = e then 1 else 0 fi)}
      aux cantConMasPuntos(t: Torneo, p: int): int
         {∑_{e:Equipo} if 0 ≤ i < 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.

1. Describir la estructura a utilizar.

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

En caso de emergencia, ver las **pistas** al final de la práctica

---

Pista para el Ejercicio 22: ¿Cuántas posiciones diferentes pueden haber? ¿Cuántos puntajes distintos pueden haber?

Pista extra: ¿Cuántas posiciones cambian con cada partido?