

Ejercicio 1. Confeccione una tabla comparativa de las complejidades del 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
- 3. Árbol binarios de búsqueda
- 2. Lista enlazada ordenada
- 4. Árbol AVL

Considerando que no tengo el puntero del índice

| EST                     | PERTENECE   | INSERTAR    | BORRAR      | BUSCAR MIN  | BORRAR MIN  |
|-------------------------|-------------|-------------|-------------|-------------|-------------|
| LISTA ENLAZADA          | $O(m)$      | $O(m)$      | $O(m)$      | $O(m)$      | $O(m)$      |
| LISTA ENLAZADA ORDENADA | $O(m)$      | $O(m)$      | $O(m)$      | $O(1)$      | $O(1)$      |
| ABB                     | $O(m)$      | $O(m)$      | $O(m)$      | $O(m)$      | $O(m)$      |
| AVL                     | $O(\log m)$ |

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  $m = \text{cantmaterias}$  y  $n = \text{cantalumnos}$  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  $O(\log m)$

Como registrar materia en  $\log m$  en UN AVL. Es decir,  
está ordenado por materia.

Registrar nota en un árbol dentro del árbol de materias.  
↳ AVL.

Algo así cumplen 1, 2 y 3: VAR MATERIASALUMNOSNOTA:=DicrLog { MATERIA, DicrLog { LU, NOTA } }

O海, veo que q no lo cumple xq para calcular la CANTALUMNOSCONNOTA en mi lista tardaría mucho más xq el camino sería: x código materia, luego la nota de cada alumno. Considerando que necesito tover las materias, y tover los alumnos la complejidad sería m·m.

↳ PEOR CASO BÚSQUEDA  
EN LOS DOS AVL

Entonces, pues agregar otra variable q tiene para eso.

Más o menos así:  $\text{CANTALUMNOSCONNOTA} = \text{CANTALUMNOS} * \text{CANTNOTAS}$

NOTAS MATERIAS, como en algo regresamos xmas para AVL en peor caso.

VAR NOTASMATERIAS := DICTLOG<MATERIA, ARRAY<NOTASPOSIBLES>>

↳ 0 a 10.

Nota: En este caso, mi INVERSO debe garantizar que todos los métodos en NOTASMATERIAS devuelven MATERIASALUMNOSNOTA (no el nro, porque podría haber < q algunas materias no tenga alumnos)

→ q los demás PROCs las devuelven intactas. Tengo q ver q no se rompe.

¿Le sigue cumpliendo la complejidad? Como en los dos métodos habrá de materia nro los métodos que necesitan a MATERIA: 1, 2, 3.

- Registrar materia:  $O(\log m)$  en peor caso (MATERIASALUMNOSNOTA) y  $O(\log m)$  para insertar en NOTASMATERIAS. Luego  $O(\log m) + O(\log m)$  siendo m la CANT de materias =  $O(\log m)$

Nota: El rebalanceo de un AVL en el peor caso es  $O(\log m)$

- Registrar nota:  $O(\log m + \log m)$  en peor caso (MATERIASALUMNOSNOTA) y  $O(\log m) + O(1)$   
en NOTASMATERIAS q: Busco MATERIA, ACUAS NOTA POR ÍNDICE Y SUMO.

Luego,  $O(\log m + \log m) + O(\log m) + O(1) = O(\log m + \log m)$

- Nota de alumno no cambia q en algo específicas de lenguajes en MATERIASALUMNOSNOTA.
- q' y S no cambian q los nros NOTASMATERIAS.

MÓDULO SISTEMA IMPLEMENTA SISTEMA {

VAR MATERIASALUMNOSNOTA : DICTLOG<MATERIA, DICTLOG<LU, NOTA>>

VAR NOTASMATERIAS : DICTLOG<MATERIA, ARRAY<NOTA>>

| TIPOS                |
|----------------------|
| MATERIA : STRING     |
| LU : INT             |
| NOTA : INT => [0,10] |

PROC NUEVOSISTEMA(): SISTEMA {

RES. MATERIASALUMNOSNOTA := NEW DICTLOG<MATERIA, DICTLOG<LU, NOTA>>();

RES. NOTASMATERIAS := NEW DICTLOG<MATERIA, ARRAY<NOTA>>();

}

PROC REGISTRARMATERIA(inout S:SISTEMA, in M:MATERIA){

→  $O(\log m)$

IF (S.MATERIASALUMNOSNOTA.ESTA(M)) THEN

RETURN;

ENDIF

DICTLOG<LU, NOTA>

S. MATERIAS ALUMNOS NOTA. DEFINIR (S. MATERIAS ALUMNOS NOTA, M, NEW DICTLOG<LU,NOTA>()); O(log m)

VAR NOTASMATERIA: ARRAY<NOTA> := NEW ARRAY<NOTA>(14);

↳ se iniciaiza con 0's.

S. NOTASMATERIAS. DEFINIR (S. NOTASMATERIAS, M, NOTASMATERIA); O(log m)

}

*luego que uno admite cambiar nota. Si es ci, si Alumno ya tiene, nota.*

PROC REGISTRARNOTA(inout S:SISTEMA, in M:MATERIA, in A:ALUMNO, in N:NOTA){

VAR MATERIA: DICTLOG<LU,NOTA> := S. MATERIAS ALUMNOS NOTA. OBTENER(M); O(log m)

IF (MATERIA. ESTA(A)) THEN

RETURN;

ENDIF

MATERIA. DEFINIR (MATERIA, A, N); O(log n)

VAR NOTASMATERIA: ARRAY<NOTA> := S. NOTASMATERIAS. OBTENER(M); O(log m)

NOTASMATERIA[N] := NOTASMATERIA[N] + 1;

S. NOTASMATERIAS. DEFINIR (S. NOTASMATERIAS, M, NOTASMATERIA);

}

*Otro que MATERIA y ALUMNO tienen*

PROC NOTADEALUMNO(in S:SISTEMA, in A:ALUMNO, in M:MATERIA): NOTA {

VAR MATERIA: DICTLOG<LU,NOTA> := S. MATERIAS ALUMNOS NOTA. OBTENER(M);

VAR ALUMNO NOTA: NOTA := MATERIA. OBTENER(A);

RETURN ALUMNONOTA

}

*Otro q 3 materia, & 3 nota.*

PROC CANTALUMNOSCONNOTA (in S:SISTEMA, in M:MATERIA, in N:NOTA): INT {

VAR MATERIA NOTAS: ARRAY<NOTA> := S. NOTAS ALUMNOS. OBTENER(M);

RETURN MATERIANOTAS[N];

}

*ojo en NOTA INT  
VA POR COPIA.*

*Otro q 3 materia, & 3 nota.*

PROC CANTALUMNOSAPROBADOS (in S:SISTEMA, in M:MATERIA, in N:NOTA): INT {

VAR MATERIA NOTAS: ARRAY<NOTA> := S. NOTAS ALUMNOS. OBTENER(M);

*ojo en NOTA INT VA POR COPIA.*

RETURN MATERIANOTAS[7] + MATERIANOTAS[8] + MATERIANOTAS[9] + MATERIANOTAS[10];

}

✓

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.

¿Un ARRAY infinito? Al BANCA 1, 2, 3 y 4 → <sup>→</sup> Los guarda OTRA VAR.  
Igual a la inicial pero inv de todo F, todo T.

¿Complementar  $O(1)$ ? Mm, quizás guarda la estructura invertida y cada vez q vaya a modificar la INV también? tq si lo hace cada vez q le matrix en peor caso es m.m  
BAH, no sé qe sería INVERTIR. Supong qe para todo V o F q F o V.

Al me viene algo así:

$$\begin{array}{|c|} \hline FF \\ \hline FF \\ \hline \end{array} \quad \begin{array}{|c|} \hline FT \\ \hline TF \\ \hline \end{array}$$

$$[1][1] = T \Rightarrow \begin{array}{|c|} \hline FF \\ \hline FT \\ \hline \end{array} \quad \begin{array}{|c|} \hline TT \\ \hline TF \\ \hline \end{array}$$

En esta solución necesitás espacio para guardar las matrices infinitas, pero en la UNICA forma de O(1) o. una matriz.

Creo q la idea es ella pero no sé cómo definir algo q e inviertan

MÓDULO MATRIZINFINITABOOL IMPLEMENTA MATRIZINFINITABOOL {

VAR D: VECTOR<VECTOR<BOOL>> ~> [[], []]

VAR INV: BOOL

PROC NUEVAMATRIZINFINITABOOL(): MATRIZINFINITABOOL {

RES.D := NEW VECTOR<VECTOR<BOOL>>();

RES.INV := false

}

PROC VER(IN M: MATRIZINFINITABOOL, IN F: INT, IN C: INT): BOOL {

RETURN M.OBTENER(F).OBTENER(C);

}

PROC ASIGNAR(INOUT M: MATRIZINFINITABOOL, IN F: INT, IN C: INT, IN VAL: BOOL) {

M.OBTENER(F).MODIFICARPOSICION(C, VAL);

}

```
PROC CONPONENTIA (INOUT M:MATRIZ;INFINUMBAOL){
```

```
    M.INV:=INNE
```

```
}
```

```
}
```

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,

$m$  es la cantidad de elementos de la matriz.

- Sé que la cantidad de filas y columnas es fija, es decir, nunca mezclaré a tener que redimensionar para agregar espacios o quitar.
- Definir tiene en valor la alguna posición, el costo es  $\Theta(m)$ .
- #Filas y #Columnas las defino al principio, devolviendo una constante de los índices que fueran.
- Obtener nos piden que sea  $\Theta(m)$  en el PEOR CASO, es decir, no tenemos una estructura de datos indexada.
- Unir matrices supongo que viene algo como  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 3 & 5 \end{bmatrix}$

Definir y obtener con los clave xq tienen  $\Theta(m)$  devolviendo una posición malida.

Unir matrices en  $\Theta(m+m)$ , no distinguo recorrer filas y luego columnas.

- CONSUTO: No tendría sentido. No puede ocurrir el elem vector
  - ARRAT: No, si el coste de finar la longitud definida en  $\Theta(1)$ , no  $\Theta(m)$ .  
Odemos si tiene ARRAT, ADDAT la misma matriz sería  $m_1 \cdot m_2 \rightarrow \text{CANT FILAS}$   $m_2 \rightarrow \text{CANT COLUMNAS}$  Grisarlos la longitud se  $m=m$ .
  - AVL: No tendría sentido, la complejidad de inserción es  $\Theta(\log m)$ .
  - ABB: Puede ser siempre y cuando el orden esté dado por los índices, Aquí el DEFINIR y OBTENER sería  $O(n)$ . No, no es posible que ni 00 sea menor que 1, o la izquierda no sea a la derecha menor.
  - LISTAENLAZADA: Sié que OBTENER y DEFINIR SON  $O(n)$  tienen el PRIMER. Sin embargo, deben tener UNA PARA CLAVES y OTRA PARA VALORES.
  - DICCIONARIO: No se garantiza el orden, así que, definir y obtener serían  $\Theta(1)$
- L> DICCIONARIO LINEAL: DEFINIR y OBTENER en  $\Theta(n)$

```
Modulo DiccionarioLineal<K, V> implementa Diccionario<K, V> {
    var claves: ListaEnlazada<K> // Lista de claves. No tiene que tener repetidos
    var valores: ListaEnlazada<V> // Lista de valores. Puede haber repetidos
    ...
}
```

¿Pues para para human VALORVALOR siendo misma clave? Sería necesario algo m<sup>2</sup> veces  
tiendo m=m.

\* MEDIAN INDICE: Busco EN DL1, AGARDO VALOR Y BUSCO EN DL2 EL INDICE, AHÍ SUMO. MUY COSTOSO.

Recuerda tener garantías que si el coste de sumar los matrizes en la longitud en que ya tengo esa

fórmula tipo:  $[1, 2] + [2, 4] + [0, 2] + [1, 0] = [3, 6]$

Ocio el caso de m > m según qué sea más grande.



**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, sum1(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) } ↳ mō se borra de CANT la KEY eliminada
        asegura { d.cant == old(d).cant } ↳ CANT > DATA

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

    aux sum1(k:K, cant: dict<K, int>)
        { if (k in cant) then (cant[k] + 1) else 1 fi }
}

```

Dado  $n$  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  $O(n)$

*↳ "OBTENER" en OTRA LISTA.*

Podría usar dos listas ENLAZADAS. Mi INVERP sería que si  $\exists$  KEY en DATA tiene en CANT, pero no el revér. En CANT la C KEY position.

*↳ (h1n1)*

Ej: D: {"A": "Ton"}      C: {"A": 1}  
 D: {}                      C: {"A": 1}

*VAR CLAVES: LISTAENLAZADA<K>*  
*VAR VALORES: LISTAENLAZADA<V>*

MÓDULO DICCIONARIOCONHISTORIA<K,V> IMPLEMENTA DICCIONARIOCONHISTORIA<K,V>{

VAR DATA : DICCIONARIOLINEAL<K,V>

VAR CANTDEF : DICCIONARIOLINEAL<K,int>

PROC NuevoDiccionario(): DICCIONARIOCONHISTORIA<K,V>{

RES. DATA := NEW DICCIONARIOLINEAL<K,V>();

*→ O(1)*

D: {"A": "Ton"}

C: {"A": 1}

*↳ A: definido*

D: {"A": "T"}

C: {"A": 2}

*↳ P.N.V.P.*

CANT SIGNIFICADOS Genera CANT definiciones de la KEY K.

```
RES.CANTDEF := NEW DICCIONARIOLINEAL<K,INT>();  
}
```

```
PROC ESTA (IN D:DICCIONARIOCONHISTORIA<K,V>, IN K:K):BOOL {  
    RETURN D.DATA.ESTA(D.DATA,K);  
}  
↳ O(m)
```

```
PROC DEFINIR (INOUT D:DICCIONARIOCONHISTORIA<K,V>, IN K:K, IN V:V) {  
    D.DATA.DEFINIR(D.DATA,K,V); → O(m)  
    VAR ESTA:BOOL := D.CANTDEF.ESTA(D.CANTDEF,K) → O(m)  
    VAR CANTDEF:INT := 1;
```

```
IF (ESTA) THEN  
    CANTDEF := D.CANTSIGNIFICADOS(D,K) + 1;  
ENDIF  
↳ O(n)  
D.CANTDEF.DEFINIR(D.CANTDEF,CANTDEF);  
}
```

```
PROC OBTENER (IN D:DICCIONARIOCONHISTORIA<K,V>, IN K:K):V {  
    RETURN D.DATA.OBTENER(K);  
}  
↳ O(m)
```

```
PROC BORRAR (INOUT D:DICCIONARIOCONHISTORIA<K,V>, IN K:K):V {  
    VAR BORRADO:V := D.DATA.OBTENER(D.DATA,K); ¿V es primitivo? Si es así no hay borrado. Por contraria veo aliasing.  
    D.DATA.BORRAR(D.DATA,K); → O(n)  
    D.CANTDEF.BORRAR(D.CANTDEF,K);  
    ↳ O(m)
```

```
RETURN BORRADO  
}
```

PROC CANTSIGNIFICADOS (in D: DICCIONARIO CON HISTORIA <K,V>, in K: K) : INT

RETURN D.CANTDEF.OBTENER(D.CANTDEF, K);  
}

↳ O(m)

EXTRA:

PRED INVERSE (D: DICCIONARIO CON HISTORIA <K,V>) {

1. Todas CLAVES en D.DATA.CLAVES están en D.CANTDEF.CLAVES (no olvidar que puede haber más de una)
2. La cantidad de claves en D.CANTDEF >= cantidad de claves en D.DATA
3. Todas las VALORES de D.CANTDEF son mayores a 0.
4. D.DATA.CLAVES y D.DATA.VALORES NO tienen ciclos.
5. D.CANTDEF.CLAVES y D.DATADEF.CLAVES NO tienen ciclos.

}

↗ TAB

PRED ABS (D: DICCIONARIO CON HISTORIA <K,V>, D': DICCIONARIO CON HISTORIA <K,V>) {

- MOS  
A  
TAD
- TAD  
A  
MOS
1. Todas las claves de D.DATA.CLAVES están en D'.DATA
2. Todas las valores de D.DATA.VALORES están en D'.DATA como valor.
3. Todas las claves de D.CANTDEF están en D'.CANT
4. Todas las valores de D.CANTDEF están en D'.CANT
5. Todas las claves de D'.DATA están en D.DATA.CLAVES
6. Todas las valores de D'.DATA están en D.DATA.VALORES
7. Todas las claves de D'.CANT están en D.CANTDEF.CLAVES
8. Todas las valores de D'.CANT están en D.CANTDEF.VALORES
- } ↳ PREGUNTA: ¿Pueden formar con RECURSIÓN?

}

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 quiero 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 {totalDia == []}

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  $O(1)$ .  
→ Solo O(1)  
Estructuras.
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  $O(n)$ , 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( $i, 1, d$ ) ≤ cantPersonas( $i, d + 1, totDias(i)$ ), restringiendo la operación a los casos donde dicho día existe.

$$[10, 30, 90] =$$

↑                      ↑  
 DIA 1                  DIA 3 = DIA 3 - DIA 2 - DIA 1 = 50

( ) UOY ACOM,

Para range, DIA FIN - DIA INI

**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.

No entiendo xD

$$P_1.\text{EDAD} < P_2.\text{EDAD} \quad \text{||} \quad (P_1.\text{EDAD} == P_2.\text{EDAD}) \wedge P_1.\text{SEXO} < P_2.\text{SEXO}$$

( )

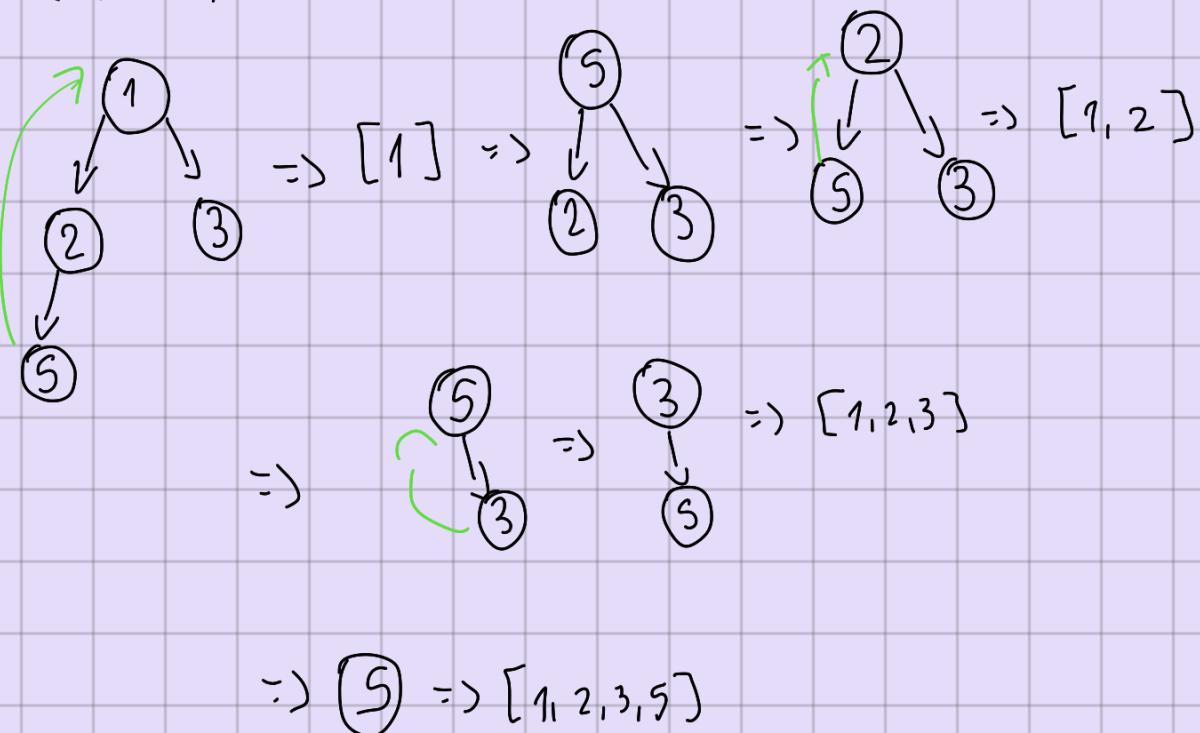
¿Por qué considero que el criterio es menor?

Comparar  $P_1.\text{EDAD}, P_1.\text{SEXO}$  al aux o igual.

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

Poraría la secuencia a MIN-HEAP y luego sacaría de o. uno.

[1, 3, 2, 5]



PROC ORDENAR(*inout* S:ARRAY<T>){

VAR MH: COLAPRIORIDADLOG<T> := NEW COLAPRIORIDADLOG<T>();

VAR i:int := 0;

WHILE (i < S.LENGTH) DO:

MH. ENCOLAR(MH, S[i]);  $\Rightarrow O(m \cdot \log(m))$

i++;

END WHILE

?  $O(m \cdot \log(m))$ ?

i := 0;

WHILE (!MH.VACIA) DO:

S[i] := MH. DESENCOLARMIN(MH);  $\Rightarrow O(m \cdot \log(m))$

i++;

END WHILE

}

### 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?

ABAJO DE TODO.

### Elección de estructuras (heaps y tries)

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

| EST                     | PERTENECE | INSERTAR | BORRAR | BUSCAR MIN | BORRAR MIN |
|-------------------------|-----------|----------|--------|------------|------------|
| LISTA ENLAZADA          | $O(m)$    | $O(m)$   | $O(m)$ | $O(n)$     | $O(m)$     |
| LISTA ENLAZADA ORDENADA | $O(m)$    | $O(m)$   | $O(m)$ | $O(1)$     | $O(1)$     |

|      | $O(m)$      | $O(m)$      | $O(m)$      | $O(m)$      | $O(m)$      |
|------|-------------|-------------|-------------|-------------|-------------|
| AVL  | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log m)$ | $O(\log n)$ |
| HEAP | $O(\log m)$ | $O(\log m)$ | —           | $O(1)$      | $O(\log m)$ |
| TRIE | $O( m )$    |

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  $\rightarrow$  MAX HEAP

```
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  $O(\log(m))$
- ventasACliente en  $O(1)$

donde m es la cantidad de pedazos de listón que hay en el depósito

- Escriba el algoritmo para la operación venderUnListon

$\rightarrow$  POSIBLE PROBLEMA: LISTONES REPETIDOS

1) Los CLIENTES ESTÁN EN UN TRIE, y en  $O(1)$  xq los dígitos están

Acotadon

$\rightarrow$  Es importante

Eg:  $\{ "123": [ ("11/03/01", 10), ("21/03/02", 12) ] \}$

$O(1)$

El problema con un array / lista es que cuando la persona compra, si el array está lleno tiene que redimensionar. Mejor usar algo más flexible (conjunto).

$\{ "123": \{ (13/03, 10), (21/03, 12) \} \}$

$\rightarrow$  si se vende la compra, debría quitarla (en el otro (w) ej)

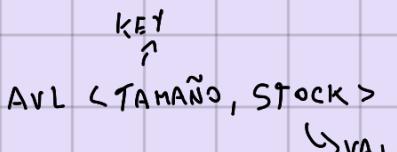
2) Comprar (afegir) y vender (modificación / eliminar)

No me preocupo por afegir al cliente luego de vender xq es  $O(1)$ ,

Orientado a fechas sin problemas y comprobando que no haga la misma compra

(xq dentro el set en ADL) - me dice el tipo de lista - Punto en punto

Son estructuras de datos para organizar, modificar o eliminar en el AVL.  
 El HEAP no tiene rama ni hoja de prioridad ni nodo, entonces el  
 el eliminar es  $O(n)$ .



LISTON: TAMAÑO es INT, STOCK es INT, FECHA es INT (123467)

FECHA  
 ↑

MODULO MADERERA IMPLEMENTA MADERERA {  
 ↗ MAX HEAP

VAR LISTONES: COLADEPRIODAD < LISTON >

VAR CLIENTES: DICCIONARIO DIGITAL < CLIENTE, CONJUNTO LINEAL < (FECHA, LISTON) > >

NOTA: FUNC "CONS MARIN" se dejan más largas, critica. Supongamos que el heap para el resto puede darse.

PROC VENDERUNLISTON (INOUT M: MADERERA, IN TAMAÑO: INT, IN CL: CLIENTE, IN F: FECHA) {

REQUIERE: {CL E M.CLIENTES} // si el no especifica en el TAB, creamos que ya lo registraron anterior.

REQUIERE: {F > 0} // FECHA ES POSITIVA (esta debería ir en el INV ref).

VAR MAX: INT := M.LISTONES. CONSULTARMAX()

IF (TAMAÑO > MAX) RETURN;

VAR TAMMAX := M.LISTONES. DESENCOLAR();

VAR NESTO: INT := TAMMAX - TAMAÑO

M.LISTONES. DEFINIR ((NESTO, NESTO)) //  $\Theta(\log(n))$  LA TUPLA TIENE PRIORIDAD EL SEGUNDO ELEMENTO

VAR CLIENTE := M.CLIENTES(CL); //  $\Theta(1)$

CLIENTE. AGREGARRAPIDO ((F, TAMAÑO)) //  $\Theta(1)$   
 ↗ copia

}

}

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 no permiten una consulta directa, se permite una búsqueda de los libros en la estantería.

Cuando la biblioteca adquiere un nuevo libro o cuando un libro es devuelto, este 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

```

Socio es string; Libro, Posición es int

Módulo BibliotecaImpl implementa Biblioteca <
var socios: Diccionario<Socio, ConjuntoLibro>; // 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

```

- $L$  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

2. Escribir los algoritmos mostrando cómo se cumplen las cotas de complejidad requeridas.

```

proc AgregarLibroALCatalogo(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 UbicacionDelLibro(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

Con datos del enunciado sé que:

- 1) Al los nombres son ordenados, pues tienen un trie por cliente.
- 2) Sé que para devolver el libro se tiene que sacarlos de mi estructura aux del cliente (DEVOLVER LIBROS), x lo tanto una estructura sería en un conjunto (por Buscar, Quitar) AVL

### CLIENTE: TRIE CON UN AVL DE LIBROS.

- 3) Si  $k$  es considerada una posición libre, tendré que eliminarlo / Agregar rápido en  $O(\log n)$  (de vez en cuando presta o devuelva). Cuando saque libro de por, debié fijarme que se agrega otro libro.
- 4) Los libros no tienen "stock", solo 3 ems. Yo hago relación x libro y posición libre.
- 5) No hay libros repetidos (requiere).
- 6) En el AVL de libros no me importa manejar el orden, las posiciones libres ni xq necesito el primer lugar maria
- 7) Si no hay pos libro: inserir en catalogo xq las pos van en orden
- 8) Como PRESTADOS es  $\Theta(1)$  NO deviamos olvidar

MÓDULO BIBLIOTECAIMPLEMENTA BIBLIOTECA {

```

VAR Socios: DiccionarioDigital<Socio, ConjuntoLog<Libro>>

VAR CATALOGO: DiccionarioLog<Libro, Posicion>
    ↗ EN AVL. ABAJO BUSCAR MIN

VAR POSICIONESLIBROS: ColaPrioritarios<Posiciones>
    ↗ "PRIMERA POSICIÓN LIBRE"
    ↗ LA PRIORIDAD ES POR EL NÚMERO MENOR.

PROC AGREGARLIBROALCATALOGO(inout b: Biblioteca, in l: idLibro){}

VAR VACIO:Bool := B.POSICIONESLIBROS.VACIO();

```

IF(VACIO) THEN

IF (VALOR) THEN

POSICION := B.CATALOGO.TAMAN $\tilde{O}$  //Libros por Pos Libro, LIBROS POS S.

ELSE

VAR POSICION: POSICION := B.POSICIONESLIBRESCONSULTARMIN(); // $\Theta(1)$

B.POSICIONESLIBRESDESENCOLARMIN(); // $\Theta(B.POSICIONESLIBRES) \Rightarrow \Theta(\log PL)$

ENDIF

B.CATALOGO.DEFINIRRAPIDO(L, POSITION); // $\Theta(\log C)$

}

Todo LIST:

// SACO POS DE LIBRO, LO OFERO EN POSICIONES LIBRES

// SE LO AGREGO AL SOCIO

// LO BORRA DEL CATALOGO

PROC PEDIRLIBRO (INOUT B:BIBLIOTECA, IN L:IDLIBRO, IN S:SOCIO){

VAR POSLIBRO: INT := B.CATALOGO.OBTENER(L); // $\Theta(\log C)$

B.POSICIONESLIBRESENCOLAR(POSLIBRO); // $\Theta(\log PL)$

VAR LSOCIOS: CONJUNTOLOG<LIBRO> := B.OBTENER(S); // $\Theta(1)$  q en $\circ$  costo

LSOCIOS.AGREGAR(L); // $\Theta(\log S)$

B.CATALOGO.BORRAR(L); // $\Theta(\log C)$

}

Todo LIST:

// SACO EL LIBRO DEL SOCIO

// BUSCO PRIMER POS LIBRO EN POSICIONES SI ES NULL, OFERO AL FINAL DEL CATALOGO

BUSCANDO ANTES MAYOR POS EN CATALOGO

//AGREGO EN CATALOGO

PROC DEVOLVERLIBROS (INOUT B:BIBLIOTECA, IN L:IDLIBRO, IN S:SOCIO){

VAR LSOCIOS: CONJUNTOLOG<LIBRO> := B.OBTENER(S); // $\Theta(1)$  q en $\circ$  costo

LSOCIOS.BORRAR(L); // $\Theta(\log S)$

VAR POSICION: INT := B.POSICIONESLIBRESCONSULTARMIN(); // $\Theta(1)$

IF (!Posicion) THEN

Posicion := B.CATALOGO.TAMAN $\tilde{O}$

ELSE

B.POSICIONESLIBRES. DESENCOLARMIN(l); // $\Theta(\log PL)$

ENDIF

B.CATALOGO. AGREGAR(l, POS: CION); // $\Theta(\log C)$

)

PROC PRESTADOS(in B:BIBLIOTECA, in S:SOCIO): CONJUNTOLOG<LIBRO>

RETURN B.SOCIOS.DETENER(S); // $\Theta(1) \rightarrow$  el socio tiene este libro.

)

PROC UBICACIONDELIBRO(in B:BIBLIOTECA, in l: LIBRO): POSICION {

RETURN B.CATALOGO.DETENER(l); // $\Theta(\log C)$

)

}

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 registrar las actividades debe indicar su hora de inicio y su hora de finalización (por ejemplo, 21:00 horas) y terminar en un horario posterior a su inicio. Tampoco pueden empezar y terminar en el mismo momento. 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 como 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 dia: Dia,
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<dia: Dia, 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 HoraMasOcupada(in ag: Agenda, in d: Dia): Hora
Requiere: el día indicado tiene al menos una actividad registrada.
Descripción: se devuelve la hora que tiene más actividades registradas.
Complejidad:  $O(\log(d))$ . → LA HORA LA TIENE EN OTRO.
proc ActividadesPorTag(in ag: Agenda, in t: Tag): Conjunto<IdActividad>
Requiere: true.
Descripción: se devuelven las actividades que tienen registrado el tag.
Complejidad:  $O(1)$ .
```

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

7

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.

• A: iD, Hi, HF

•  $Ho \exists dia Act dia$  (en minus iD).

•  $For A tienen Hi y HF \neq$  Terminan los fines.

•  $HF > Hi$

• Empiezan y terminan en mismo día.

• Al comienzo comienza y transcurren en

en determinados horarios NO se sella

comienza los y finalizan en ese horario.

Ej: 20:00

A<sub>1</sub>: Hi: 19 HF: 20

A<sub>2</sub>: Hi: 20 HF: 21

↳ Amb A<sub>2</sub>.

• ACT puede tener tags

↳ MAX 20 (AN).

ACT  $\xrightarrow{\text{TIENE}}$  TAG

TAG  $\xrightarrow{\text{TIENE}}$  ACT

TAGS: Diccionario Digital <TAG, Conjunto Lineal <IDActividad>>

El registro me Ofrece &

la ACT no tiene el TAG.

ACTIVIDAD: Diccionario Log <IDActividad, Struct <Dia:DIA, Hi:Hi, HF:HF>>

DIAS: AVL <Dia, Vector<int>{24}>

↓  
Los HS en punto

2) - TODA IDACTIVIDAD EN TAGS ES CLAVE EN ACTIVIDAD ✓

· HF > Hi EN ACTIVIDAD ✓

· POR CADA DIA EN DIAS, LA CANTIDAD EN CADA INDICE CORRESPONDE ✓  
A LASUMA DE ACTIVIDADES EN ESE DIA, HORA INICIO (FIN NO SE CONTEMPLA) ✓

· EN CADA ACT, Dia, Hi y HF Están dentro de DIAS donde la clave es Dia, y la Hi es igual ✓

· TODOS DIA EN DIAS ESTAN EN ACTIVIDAD, PRIMER ELEMENTO DIA

3) MÓDULO AGENDA IMPLEMENTA AGENDA {

VAR TAGS: Diccionario Digital <TAG, Conjunto Lineal <IDActividad>>

VAR ACTIVIDAD: Diccionario Log <IDActividad, Tupla(Dia, Hi, HF)>

VAR DIAS: Diccionario Log <Dia, ARRAY<HORA>{24}>

PROC REGISTRARACTIVIDAD (INOUT AG:AGENDA, IN ACT:IDActividad, IN DIA:DIA, IN INICIO:HORA, IN FIN:HORA){

AG.ACTIVIDAD.DEFINIR(ACT, TUPLA(DIA, HI, HF))

VAR HORASDIA:VECTOR<INT> := AG.DIAS.OBTENER(DIA);

HORASDIA[HI] := HORASDIA[HI] + 1; // Asume evento dia. HF No cuenta

}

PROC VERACTIVIDAD (IN AG:AGENDA, IN ACT:IDActividad): STRUCT<Dia:DIA, INICIO:HORA, FIN:HORA> {

RETURN AG.ACTIVIDAD.OBTENER(ACT);

}

PROC AGREGARTAGS(inout AG:AGENDA, in ACT: CONJUNTO LINEAL<ACTIVIDAD>; in T:TAG){

VAR ACTTAGS: CONJUNTO LINEAL<ACTIVIDAD> := AG.TAGS.OBTENER(T);

ACTTAGS.AGREGARRAPIDO(ACT); // X REQUIERE Q ACT NO TIENE ESE TAG

}

PROC HORAMASOCUPADA(in AG:AGENDA, in D:DIA):HORA{

VAR HORASDIAACT: VECTOR<HORA> := AG.DIAS.OBTENER(D);

VAR MAX:HORA:= HORASDIAACT[0];

FOR(int i:=1, i < HORASDIAACT.LENGTH(); i++) {

IF(HORASDIAACT[i] > MAX) THEN

MAX:= HORASDIAACT[i]

ENDIF

}

RETURN MAX

}

PROC ACTIVIDADESPORTAG(in AG:AGENDA, in T:TAG): CONJUNTO LINEAL<ACTIVIDAD> {

RETURN AG.TAGS.OBTENER(T);

}

}

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 + [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 \leftrightarrow t.cantConMasPuntos(puntosDe(t, e)) = 0)}  
  
 aux puntosDe(t: Torneo, e: Equipo): int  
 { \sum\_{i \in t} (if 0 \leq i < t.partidos \wedge t.partidos[i].ganador = e then 1 else 0 fi) }  
 aux cantConMasPuntos(t: Torneo, p: int): int  
 { \sum\_{e: Equipo} if 0 \leq i < t.equipos \wedge 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.

• Recorrer guardan punjtos y por.

En caso q haya mas de PTO aplica

o minima por guardada

Objetivo

1. Describir la estructura a utilizar.
2. Escribir un pseudocódigo del algoritmo para las operaciones con requerimientos de complejidad.

## PUNTOS, REGISTROS PARTIDOS Y POSICIÓN EN O( $\log m$ )

$m = \text{CANT EQUIPOS}$  (NO JUEGAN TODOS)

### 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?

INVERP:

- El ARRAZ tiene (máx) 256 hijos (ancho).
- Si el nodo tiene significado → en Camino ancho que llegan a él y que Caminos se unen
- Cada sub-ancho tiene 1 father después de ningún → es null

STRUCT NODO<T> {

Significado : T

Hijos : ARRAY<NODO<T>>

}

MÓDULO TRIE {

VAR RAIZ : NODO<T>

PROC NUEVOTRIE () : TRIE {

RES.RAIZ := NEW NODO(null);

RES.RAIZ.HIJOS := NEW ARRAY<T>() {256}; //Alguno empieza en null para evitar meter un círculo

PROC AGREGAR(inout T:TRIE, in C:STRING) {

VAR i:int := 0;

VAR NODO:NODO<T>:= T.RAIZ;

VAR NUEVONODO: NODO<T> := NULL; //INICIALIZA EL ARREGLO DE 256 VALES

WHILE(i < C.LENGTH) DO

IF (i == C.LENGTH) THEN

NUEVONODO := NEW NODO(C);

ELSE

NUEVONODO := NEW NODO(NULL);

ENDIF

ASCII DE LOS

NODO.HIJOS[C.CHARAT(i)] := NUEVONODO; //LE SIGUE EL HIJO.

NODO := NUEVONODO; //MUEVE EL PUNTERO AL SIGUIENTE NODO

i++;

ENDWHILE

}

PROC BUSCAR(IN R:TRIE, IN C:T):T {

VAR i:INT:=0;

VAR RAIZ:NODO<T>:= T.RAIZ;

WHILE(i < C.LENGTH) DO

RAIZ := RAIZ.HIJOS[C.CHARAT(i)];

i++;

ENDWHILE

RETURN RAIZ.SIGNIFICADO

}

}

STRUCT NODO<T> {

SIGNIFICADO:T

HIJOS: LISTAENLAZADA<NODO<T>>

}

MÓDULO TRIE {

VAR RAIZ:NODO<T>

PROC NUEVOTRIE():TRIE {

RES.RAIZ := NEW NODO(NULL);

RES.RAIZ.HIJOS := NEW LISTAENLAZADA<TRIE>(); // OBTENER SON 256 POSIBLES

}

PROC AGREGAR(inout T:TRIE, in C:STRING) {

VAR i:int:=0;

VAR Nodo:Nodo<T>:= T.RAIZ;

VAR NUEVONODO:Nodo<T>:= NULL; //INICIALIZA LA LISTA ENLAZADA VACIA (HIJOS)

WHILE(i < C.LENGTH) DO

IF (i == C.LENGTH) THEN

NUEVONODO := NEW NODO(C);

ELSE

NUEVONODO := NEW NODO(NULL);

ENDIF

NODO.HIJOS.AGREGAR(NUEVONODO); // le agrega el hijo.

NODO := NUEVONODO; //mover el puntero al otro nodo

i++;

ENDWHILE

}

PROC BUSCAR(in r:TRIE, in c:T):T {

VAR i:int:=0;

VAR RAIZ:Nodo<T>:= T.RAIZ;

WHILE(i < C.LENGTH) DO

RAIZ := RAIZ.HIJOS.OBTENER(C.CHARAT(i)); //O(M)

i++;

↳ Esto nos lleva a tiempo constante

en un indice fijo en la lista enlazada por

ENDWHILE

añadir el valor en orden.

RETURN RAIZ. SIGNIFICADO

}

}

Con lista enlazada es más difícil de hacer ya que no tenemos indice para los nodos del array en la raíz tenemos q ir agrandando si se llena.

Ol no tener indice necesitas guardar algo más del significado y obtener busca por índice y así así frases el significado al final.

BÁSICAMENTE: si los C en T0, debes rotarlos los 69 hijos de la lista enlazada.

Impresión GREECE más

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.

RDO: Floyd para de heap o ARRAY en  $O(M)$

INVERP:

El primer elemento del arreglo es el más grande y es la raíz.

Los elementos le agregan al final ( $\times 2^k$ ).

Cada elemento tiene un hijo en  $2P(v)+1$  y  $2P(v)+2$ .

→ invariante  
Cada vez q se rebalancea, cambiar prioridades linea elem y se cambia el nodo q le "da" su prioridad, DESAPILAR MAX tamb rebalancea.

TODO: lo haré con rebalancear sobre array.

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.  $\Rightarrow O(|P|)$
2. `ultimaPalabra`: Devuelve la última palabra en orden lexicográfico.  $\Rightarrow O(\sum_p |P|)$
3. `palabrasConPrefijo`: Devuelve todas las palabras que tienen un determinado prefijo, ordenadas en orden lexicográfico.

→ Considero la q armé con ARRAYS.

MÓDULO TRIE {

VAR RAIZ: NODOCTO

PROC PRIMERAPALABRA (INT:TRIE): T {

VAR Nodo; Nodo<T>; = T.RAIZ;

VAR i:int := 0;

WHILE (i < 156) Do

IF (Nodo.Hijos[(CHAR)i] != NULL) THEN

Nodo := Nodo.Hijos[(CHAR)i];

i:=0; //YA ENTRE A UNA, AHORA BORRAR HIJO NO NULO

ENDIF

i++

END WHILE

RETURN Nodo.Significado;

}

PROC ULTIMAPALABRA (in T:TRIE):T {

