DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Guia 6

2do cuatrimestre 2024 — Algoritmos y Estructuras de Datos I

Integrante	LU	Correo electrónico
Federico Barberón	112/24	jfedericobarberonj@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

1. Guia 6

1.1. Ejercicio 1

Implementamos el TAD Secuencia sobre una lista simplemente enlazada usando

NodoLista<T> es struct<
valor: T,
siguiente: NodoLista<T>
>

```
Módulo ListaEnlazada<T> implementa Secuencia<T> {
    var primero: NodoLista<T>
    var ultimo: NodoLista<T>
    var longitud: int

    proc NuevaListaVacía () : ListaEnlazada⟨T⟩

        res.primero := null
        res.ultimo := null
        res.longitud := 0
        return res

    proc AgregarAdelante (inout l: ListaEnlazada⟨T⟩ , in e: T)

        nodo := new NodoLista< T >
        nodo.valor := e
        nodo.siguiente := null

        if l.longitud == 0 then
            l.primero := nodo
            l.ultimo := nodo
        else
            nodo.siguiente := l.primero
            l.primero := nodo
        endif
        l.longitud := l.longitud + 1

    proc Pertenece (in l: ListaEnlazada⟨T⟩ , in e: T) : bool

        res := false
        actual := l.primero
        while actual != null do
            if actual.valor == e then
                res := true
            endif
            actual := actual.siguiente
        endwhile
        return res

}
```

- Escriba los algoritmos para los siguientes procs y calcule su complejidad
  - agregarAtras
  - obtener
  - eliminar
  - concatenar
- Escriba el invariante de representación para este módulo en castellano

- Dado el siguiente invrep, indiquie si es correcto. En caso de no serlo, corrijalo:

  `pred InvRep` (l: NodoLista<T>) {

  $accesible(l.primero, l.ultimo) \land largoOK(l.primero, l.longitud)$

  }

  `pred largoOK` (n: NodoLista<T>, largo: $\mathbb{Z}$) {

  $(n = null \land largo = 0) \lor (largoOK(n.siguiente, largo - 1))$

  }

  `pred accesible` (n_0: NodoLista<T>, n_1: NodoLista<T>) {

  $n_1 = n_0 \lor (n_0.siguiente \neq null \land_L accesible(n_0.siguiente, n_1))$

  }

`proc agregarAtras` (inout l: ListaEnlazada<T>, in e: T)
Complejidad: $O(1)$

```
var nodo: NodoLista < T >;

nodo := new NodoLista < T >;
nodo.valor = e;

if l.longitud == 0
    l.primero = nodo;
    l.ultimo = nodo;
else
    l.ultimo.siguiente = nodo;
    l.ultimo = nodo;
endif
```

`proc obtener` (in l: ListaEnlazada<T>, in indice: int) : T
Complejidad: $O(n)$

```
var actual: NodoLista < T >
var i: int;

actual := l.primero;
i := indice;

while i > 0 do
    actual := actual.siguiente;
    i := i - 1;
endwhile

return actual.valor;
```

```
proc eliminar (inout l: ListaEnlazada<T>, in indice: int)
```
Complejidad: $O(n)$

```
var actual: NodoLista < T >
var i: int;

actual := l.primero;
i := indice;

while i > 1 do
    actual := actual.siguiente;
    i := i − 1;
endwhile

actual.siguiente := actual.siguiente.siguiente;
```

```
proc concatenar (inout l1: ListaEnlazada<T>, in l2: ListaEnlazada<T>)
```
Complejidad: $O(m) \leftarrow m = Long(l2)$

```
var actual: NodoLista < T >;
var ultimo: NodoLista < T >;

actual := l2.primero;
ultimo := l1.ultimo;

while actual != null do
    nodo := new NodoLista< T >
    nodo.valor = actual.valor;
    ultimo.siguiente = nodo;
    ultimo := nodo;
endwhile
```

InvRep:

- La lista no tiene ciclos
- La cantidad de nodos es igual a la longitud
- No existe nodo que apunte al primero, y el ultimo siempre apunta a null
- El primer nodo tiene una conexión indirecta con el último nodo

```
pred InvRep (l: ListaEnlazada<T>) {
```
$accesible(l.primero, l.ultimo) \land largoOK(l.primero, l.longitud) \land$
$sinCiclos(l.primero) \land l.ultimo.siguiente = null \land$
$(\forall n : \text{NodoLista<T>})\ (n.siguiente \neq l.primero)$
```
}
```

1.2. Ejercicio 2

Implemente el TAD ConjuntoAcotado<T> (definido en el apunte de TADs) usando la siguiente estructura.

```
Módulo ConjuntoArr<TAD> implementa ConjuntoAcotado<T> {
    var datos: Array < T >
    var tamaño: int
}
```

- Escriba el InvRep y la función Abs
- Escriba los algoritmos para las operaciones conjVacío y pertenece

- Escriba el algoritmo para la operación agregar
- Escriba los algoritmos para las operaciones unir e intersecar
- Escriba el algoritmo para la operación sacar.
- Calcule la complejidad de cada una de estas operaciones
- Qué cambios haría en su implementación si se quiere que la operación agregar sea lo más rápdia posible? Y si se quiere acelerar la operación buscar? Indique los cambios en la estructura, el InvRep, la función Abs y los algoritmos.

Módulo ConjuntoArr<T> implementa ConjuntoAcotado<T> {

var datos: Array < T >

var tamaño: int

pred InvRep (c: $ConjuntoArr\langle T\rangle$) {

$c.tamaño \leq length(c.datos) \wedge_L (\forall i, j : \mathbb{Z})\ (0 \leq i < j < c.tamaño \rightarrow_L c.datos[i] \neq c.datos[j])$

}

pred predAbs (c: $ConjuntoArr\langle T\rangle$, c': $ConjuntoAcotado\langle T\rangle$) {

$length(c.datos) = c'.capacidad \wedge$

$(\forall e : T)\ (e \in c'.elems \iff (\exists i : \text{int})\ (0 \leq i < c.tamaño \wedge_L c.datos[i] = e))$

}

proc conjVacío (in capacidad: int) : $ConjuntoArr\langle T\rangle$

Complejidad: $O(1)$

```
res.datos := new Array < T >[capacidad];
res.tamano := 0;
return res;
```

proc pertenece (in c: $ConjuntoArr\langle T\rangle$, in e: T) : Bool

Complejidad: $O(n)$

```
var i: int;
i := 0;

while i < c.tamano && c.datos[i] != e do
    i := i + 1;
endwhile

return i < c.tamano;
```

proc agregar (inout c: $ConjuntoArr\langle T\rangle$, in e: T)

Complejidad: $O(n)$

```
if !c.pertenece(e)
    c.datos[c.tamano] := e;
    c.tamano := c.tamano + 1;
endif
```

proc unir (inout c1: $ConjuntoArr\langle T\rangle$, in c2: $ConjuntoArr\langle T\rangle$)

Complejidad: $O(m) \leftarrow m = Long(c2.datos)$

```
var i: int;
i := 0;

while i < c2.tamano do
    c1.agregar(c1.datos[i]);
    i := i + 1;
endwhile
```

```
proc intersecar (inout c1: ConjuntoArr⟨T⟩ , in c2: ConjuntoArr⟨T⟩ )
      Complejidad: O(nm)

    var i: int;
    i := 0

    while i < c1.tamano do
        if !c2.pertenece(c1.datos[i])
            c1.datos[i] := c1.datos[c1.tamano − 1]
            c1.tamano := c1.tamano − 1;
        endif
        i := i + 1;
    endwhile

proc sacar (inout c: ConjuntoArr⟨T⟩ , in e: T)
      Complejidad: O(n)

    var i: int;
    i := 0

    while i < c.tamano && c.datos[i] != e do
        i := i + 1;
    endwhile

    if i < c.tamano
        c.datos[i] := c.datos[c.tamano − 1];
        c.tamano := c.tamano − 1;
    endif

}
```

1.3. Ejercicio 3

Implemente el TAD Conjunto<T> (definido en el apunte de TADs) usando la siguiente estructura.

```
Módulo ConjuntoLista<T> implementa Conjunto<T> {
    var datos: ListaEnlazada < T >
    var tamaño: int
}
```

- Escriba el InvRep y la función Abs
- Escriba los algoritmos para las operaciones conjVacío y pertenece
- Escriba los algoritmos para la operación agregar, agregarRapido y sacar
- Escriba los algoritmos para las operaciones unir e intersecar
- Calcule la complejidad de cada una de estas operaciones

```
Módulo ConjuntoLista<T> implementa Conjunto<T> {
    var datos: ListaEnlazada < T >
    var tamaño: int
    pred InvRep (c: ConjuntoLista⟨T⟩ ) {
        c.tamaño = c.datos.longitud() ∧ sinRepetidos()?
    }
    pred predAbs (c1: ConjuntoLista⟨T⟩ , c2: Conjunto⟨T⟩ ) {
        c1.tamaño = |c2.elems| ∧
        (∀e : T) (e ∈ c2.elems ⟺ (∃i : int) (0 ≤ i < c1.tamaño ∧_L c1.datos.obtener(i) = e))
```

}

proc conjVacio () : $ConjuntoLista\langle T\rangle$
Complejidad: $O(1)$

```
res.datos = new ListaEnlazada < T >;
res.tamano = 0;
return res;
```

proc pertenece (in c: $ConjuntoLista\langle T\rangle$, in e: T) : Bool
Complejidad: $O(n)$

```
var it: IteradorBidireccional < T >;
var actual: T;

it := c.datos.iterador();

while it.haySiguiente() && actual != e do
    actual := it.siguiente();
endwhile

return actual == e;
```

proc agregar (inout c: $ConjuntoLista\langle T\rangle$, in e: T)
Complejidad: $O(n)$

```
if !c.pertenece(e)
    c.datos.agregarAdelante(e);
    c.tamano := c.tamano + 1;
endif
```

proc agregarRapido (inout c: $ConjuntoLista\langle T\rangle$, in e: T)
Complejidad: $O(1)$

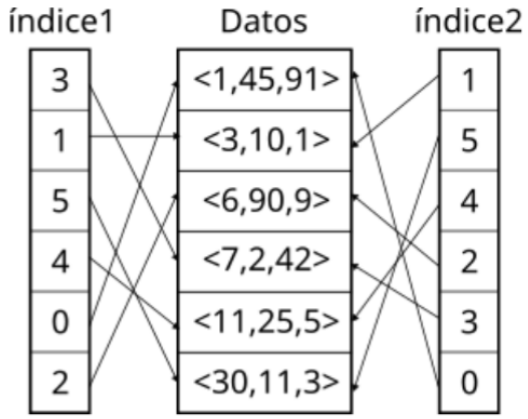
```
c.datos.agregarAdelante(e);
c.tamano := c.tamano + 1;
```

```
proc sacar (inout c: ConjuntoLista⟨T⟩ , in e: T)
    Complejidad: O(n)

    var it: IteradorBidireccional < T >;
    var i: int;
    var actual: T;

    it := c.datos.iterador();

    while it.haySiguiente() && actual != e do
        actual := it.siguiente();
        i := i + 1;
    endwhile

    if actual == e
        c.datos.eliminar(i);
        c.tamano := c.tamano − 1;
    endif

proc unir (inout c1: ConjuntoLista⟨T⟩ , in c2: ConjuntoLista⟨T⟩ )
    Complejidad: O(nm)

    var it: IteradorBidireccional
    var valor: T

    it := c2.datos.iterador();

    while it.haySiguiente() do
        valor := it.siguiente();
        if !c1.pertenece(valor)
            c1.agregarRapido(valor);
            c1.tamano := c1.tamano + 1;
    endwhile

proc intersecar (inout c1: ConjuntoLista⟨T⟩ , in c2: ConjuntoLista⟨T⟩ )
    Complejidad: O(mn^2) (seguramente mejorable a O(nm))

    var it: IteradorBidireccional;
    var valor: T;

    it := c1.datos.iterador();

    while it.haySiguiente() do
        valor := it.siguiente();
        if !c2.pertenece(valor)
            c1.sacar(valor);
            c1.tamano := c1.tamano − 1;
        endif
    endwhile

}
```

1.4. Ejercicio 4

Un *índice* es una estructura secundaria que permite acceder más rápdidamente a los datos a partir de un determinado criterio. Básicamente un índice guarda *posiciones* o *punteros* a los elementos en un orden en particular, diferente al orden original.

Imagine una secuencia de tuplas con varias componenetes ordenadas por su primer componenete. Algunas veces vamos a querer buscar (rápido) por las demás componentes. Podríamos guardar los datos en sí en un arreglo y tener arreglos con las posiciones ordenadas por las demás componenetes. A estos arreglos se los denomina índices.



En la figura, si recorremos los datos en el orden en el que están guardados, obtenemos:

$$[< 1, 45, 91 >, < 3, 10, 1 >, < 6, 90, 9 >, < 7, 2, 42 >, < 11, 25, 5 >, < 30, 11, 3 >]$$

Si lo recorremos usando el índice 1 (que apunta a los elementos en función de la segunda componente) obtenemos:

$$[< 7, 2, 42 >, < 3, 10, 1 >, < 30, 11, 3 >, < 11, 25, 5 >, < 1, 45, 91 >, < 6, 90, 9 >]$$

- Escriba la estructura propuesta
- Escriba el InvRep y la func Abs, en castellano y en lógica para el TAD Conjunto(Tupla($\mathbb{Z}, \mathbb{Z}, \mathbb{Z}$))
- Escriba el algoritmo de BuscarPor que busca por alguna componente
- Escriba los algoritmos de agregar y sacar

TuplaEnt es $\mathsf{tupla}\langle \mathsf{int}, \mathsf{int}, \mathsf{int} \rangle$

`Módulo Conjunto<TuplaEnt> implementa Conjunto<`$\mathsf{tupla}\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \rangle$`> {`

var conj: $\mathsf{Array} < \mathsf{TuplaEnt} >$

var indices: $\mathsf{tupla}\langle \mathsf{Array} < \mathsf{int} >, \mathsf{Array} < \mathsf{int} >, \mathsf{Array} < \mathsf{int} > \rangle$

`pred sinRepetidos` (a: $\mathsf{Array} < \mathsf{T} >$) {

$(\forall i, j : \mathsf{int})\ (0 \leq i < j < length(a) \rightarrow_L a[i] \neq a[j])$

}

`pred indiceValido` (a: $\mathsf{Array} < \mathsf{TuplaEnt} >$, indice: $\mathsf{Array} < \mathsf{int} >$, comp: int) {

$length(a) = length(indice) \land$

$(\forall i : \mathsf{int})\ (i \in indice \iff 0 \leq i < length(a)) \land_L$

$(\forall i : \mathsf{int})\ (0 \leq i < length(indice) - 1 \rightarrow_L a[indice[i]]_{comp} \leq a[indice[i+1]]_{comp})$

}

`pred InvRep` (c: $Conjunto\langle TuplaEnt \rangle$) {

$sinRepetidos(c.conj) \land$

$(\forall comp : \mathsf{int})\ (0 \leq comp < length(c.indices) \rightarrow_L indiceValido(c.conj, indices[comp], comp))$

}

`pred predAbs` (c1: $Conjunto\langle TuplaEnt \rangle$, c2: $Conjunto\langle \mathsf{tupla}\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \rangle \rangle$) {

$(\forall e : \mathbb{Z})\ (e \in c2.elems \iff (\exists i : \mathsf{int})\ (0 \leq i < length(c1.conj) \land_L c1.conj[i] = e))$

}

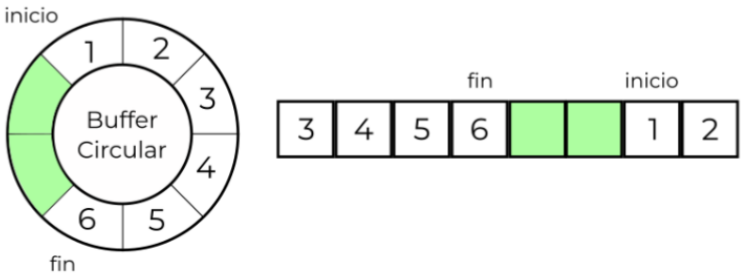
```
proc BuscarPor (in c: Conjunto⟨TuplaEnt⟩ , in comp: int, in e: int) : TuplaEnt
    var der: int;
    var izq: int;
    var med: int;
    izq := 0;
    der := length(c.conj) − 1;

    if c.conj[indices[comp][der]][comp] == e
        return c.conj[indices[comp][der]];
    endif

    while izq + 1 < der do
        med := floor((izq + der) / 2);
        if c.conj[indices[comp][med]][comp] <= e
            izq = med;
        else
            der = med;
        endif
    endwhile

    if c.conj[indices[comp][izq]][comp] == e
        return c.conj[indices[comp][izq]];
    endif

    return null;

proc agregar (inout c: Conjunto⟨TuplaEnt⟩ , in t: TuplaEnt)
    var i: int;
    var aux: Array < TuplaEnt >;
    var comp: int;
    i := 0;
    comp := 0;
    aux := new Array < TuplaEnt >(length(c.conj) + 1);

    while i < length(c.conj) do
        aux[i] := c.conj[i];
        i := i + 1;
    endwhile

    aux[length(c.conj)] := t;

    c.conj = aux;

    while comp < length(c.indices) do
        agregarEnIndice(c, comp, t);
        comp := comp + 1;
    endwhile
```

```
proc agregarEnIndice (inout c: Conjunto⟨TuplaEnt⟩ , in comp: int, in t: TuplaEnt)
    var aux: Array < int >;
    var i: int;
    var insertado: bool;

    aux := new Array < int >(length(c.indices[comp]) + 1);
    i := 0;
    insertado := false;

    while i < length(c.indices[comp]) do
        if insertado
            aux[i+1] := c.indices[comp][i];
            i := i + 1;
        else if c.conj[c.indices[comp][i]][comp] > t[comp]
            aux[i] := t;
            insertado := true
        else
            aux[i] := c.indices[comp][i];
            i := i + 1;
        endif
    endwhile

    c.indices[comp] = aux;

proc sacar (inout c: Conjunto⟨TuplaEnt⟩ , in t: TuplaEnt)
    var i: int;
    var comp: int;
    var aux: Array < TuplasEnt >;
    var borrado: bool;
    i := 0;
    comp := 0;
    aux := new Array < TuplasEnt >(length(c.conj) − 1)
    borrado := false;

    while comp < length(c.indices) do
        sacarEnIndice(c, comp, t);
        comp := comp + 1;
    endwhile

    while i < length(c.conj) do
        if borrado
            aux[i − 1] := c.conj[i];
        else if c.conj[i] != t
            aux[i] := c.conj[i];
        endif

        i := i + 1;
    endwhile

    c.conj = aux;
```

```
proc sacarEnIndice (inout c: Conjunto⟨TuplaEnt⟩ , in comp: int, in t: TuplaEnt)
    var i: int;
    var borrado: bool;
    var aux: Array < int >;
    i := 0;
    borrado := false;
    aux := new Array < int >(length(c.indices[comp]) - 1);

    while i < length(c.indices[comp]) do
        if borrado
            aux[i - 1] := c.indices[comp][i];
        else if c.conj[c.indices[comp][i]] == t
            borrado := true;
        else
            aux[i] := c.indices[comp][i];
        endif

        i := i + 1;
    endwhile

    c.indices[comp] = aux;

}
```

1.5. Ejercicio 5

Una forma eficiente de implementar el TAD Cola en su versión acotada, es mediante un *buffer circular*. Esta estructura está formada por un array del tamaño maximo de la cola (n) y dos índices (*inicio* y *fin*), para indicar adonde empieza y adonde termina la cola. El chiste de esta estructura es que, al llegar al final del arreglo, si los elementos del principio ya fueron consumidos, se puede reusar dichas posiciones.



- Elija una estructura de representación
- Escriba el invrep y la func abs
- Escriba los algoritmos de las operaciones encolar y desencolar
- ¿Por qué tiene sentido utilizar un buffer circular para una cola y no para una pila?

```
Módulo ColaBufferCirc<T> implementa ColaAcotada<T> {
    var elems: Array < T >
    var inicio: int
    var fin: int
    pred InvRep (c: ColaBufferCirc⟨T⟩ ) {
        HACER!
    }
    pred predAbs (c1: ColaBufferCirc⟨T⟩ , c2: ColaAcotada⟨T⟩ ) {
        HACER!
    }
    proc encolar (inout c: ColaBufferCirc⟨T⟩ , in e: T)
        if c.fin == -1
            c.fin := c.inicio;
        else
            c.fin := (c.fin + 1) % length(c.elems);
        endif

        c.elems[c.fin] := e;
        cant := cant + 1;

    proc desencolar (inout c: ColaBufferCirc⟨T⟩ )
        c.inicio := (c.inicio + 1) % length(c.elems)
        cant := cant - 1;

}
```

1.6. Ejercicio 6

HACER!