



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. Guía 6	3
1.1. Ejercicio 1	3
1.2. Ejercicio 2	5
1.3. Ejercicio 3	7
1.4. Ejercicio 4	9
1.5. Ejercicio 5	13
1.6. Ejercicio 6	14

1. Guía 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 := new ListaEnlazada< T >
    res.primeros := 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.primeros := nodo
      l.ultimo := nodo
    else
      nodo.siguiente := l.primeros
      l.primeros := nodo
    endif
    l.longitud := l.longitud + 1

  proc Pertenece (in l: ListaEnlazada<T> , in e: T) : bool
    res := false
    actual := l.primeros
    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, indique si es correcto. En caso de no serlo, corrija-lo:

```

pred InvRep (l: NodoLista<T>) {
    accesible(l.primerio, l.ultimo)  $\wedge$  largoOK(l.primerio, l.longitud)
}

pred largoOK (n: NodoLista<T>, largo:  $\mathbb{Z}$ ) {
    (n = null  $\wedge$  largo = 0)  $\vee$  (largoOK(n.siguiente, largo - 1))
}

pred accesible (n0: NodoLista<T>, n1: NodoLista<T>) {
    n1 = n0  $\vee$  (n0.siguiente  $\neq$  null  $\wedge_L$  accesible(n0.siguiente, n1))
}

```

```

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.primerio = 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.primerio;
    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.primerio;
    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)  $\wedge$  largoOK(l.primerO, l.longitud)  $\wedge$ 
  sinCiclos(l.primerO)  $\wedge$  l.ultimo.siguiente = null  $\wedge$ 
  ( $\forall n$  : 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 intersectar
- 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 rápida 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<T> ) {
     $c.tamaño \leq length(c.datos) \wedge_L (\forall i, j : \mathbb{Z}) (0 \leq i < j < c.tamaño \rightarrow_L c.datos[i] \neq c.datos[j])$ 
  }
  pred predAbs (c: ConjuntoArr<T> , c': ConjuntoAcotado<T> ) {
     $length(c.datos) = c'.capacidad \wedge$ 
     $(\forall e : T) (e \in c'.elems \iff (\exists i : int) (0 \leq i < c.tamaño \wedge_L c.datos[i] = e))$ 
  }
  proc conjVacio (in capacidad: int) : ConjuntoArr<T>
    Complejidad:  $O(1)$ 

    var conj: ConjuntoArr < T >;

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

  proc pertenece (in c: ConjuntoArr<T> , 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<T> , 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<T> , in c2: ConjuntoArr<T> )
    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])
            if i = c1.tamano - 1
                c1.datos[i] := null;
            else
                c1.datos[i] := c1.datos[c1.tamano - 1]
                c1.datos[c1.tamano - 1] := null;
            endif
            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
        if i == c.tamano - 1
            c.datos[i] := null;
        else
            c.datos[i] := c.datos[c.tamano - 1];
            c.datos[c.tamano - 1] := null;
        endif
        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<TAD> 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<TAD> implementa Conjunto<T> {
  var datos: ListaEnlazada < T >
  var tamaño: int
  pred InvRep (c: ConjuntoLista<TAD> ) {
     $c.tamaño = c.datos.longitud() \wedge sinRepetidos()$ ?
  }
  pred predAbs (c1: ConjuntoLista<TAD> , c2: Conjunto<T> ) {
     $c1.tamaño = |c2.elems| \wedge$ 
     $(\forall e : T) (e \in c2.elems \iff (\exists i : int) (0 \leq i < c1.tamaño \wedge_L c1.datos.obtener(i) = e))$ 
  }
  proc conjVacio () : ConjuntoLista<TAD>
    Complejidad:  $O(1)$ 

    var conj: ConjuntoLista < T >;
    conj := new ConjuntoLista < T >;
    conj.datos = new ListaEnlazada < T >;
    conj.tamano = 0;
    return conj;

  proc pertenece (in c: ConjuntoLista<TAD> , 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<TAD> , 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<TAD> , in e: T)
    Complejidad:  $O(1)$ 

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

```



```

proc sacar (inout c: ConjuntoLista(TAD) , 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(TAD) , in c2: ConjuntoLista(TAD) )
    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(TAD) , in c2: ConjuntoLista(TAD) )
    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

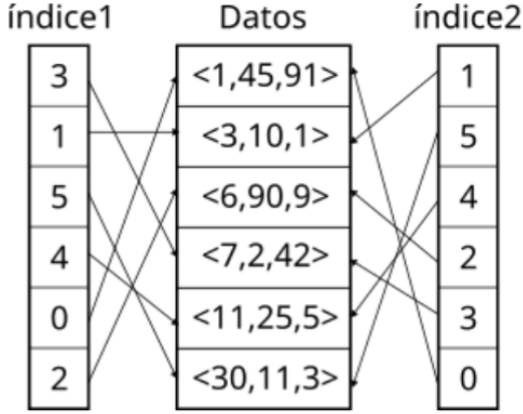
}

```

1.4. Ejercicio 4

Un *índice* es una estructura secundaria que permite acceder más rápidamente 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 componentes. 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 $\text{tupla}(\text{int}, \text{int}, \text{int})$

```

Módulo Conjunto<TuplaEnt> implementa Conjunto<tupla< $\mathbb{Z}$ ,  $\mathbb{Z}$ ,  $\mathbb{Z}$ >> {
  var conj: Array < TuplaEnt >
  var indices: tupla<Array < int >, Array < int >, Array < int >>
  pred sinRepetidos (a: Array < T >) {
     $(\forall i, j : \text{int}) (0 \leq i < j < \text{length}(a) \rightarrow_L a[i] \neq a[j])$ 
  }
  pred indiceValido (a: Array < TuplaEnt >, indice: Array < int >, comp: int) {
     $\text{length}(a) = \text{length}(\text{indice}) \wedge$ 
     $(\forall i : \text{int}) (i \in \text{indice} \iff 0 \leq i < \text{length}(a)) \wedge_L$ 
     $(\forall i : \text{int}) (0 \leq i < \text{length}(\text{indice}) - 1 \rightarrow_L a[\text{indice}[i]]_{\text{comp}} \leq a[\text{indice}[i+1]]_{\text{comp}})$ 
  }
  pred InvRep (c: Conjunto<TuplaEnt>) {
     $\text{sinRepetidos}(c.\text{conj}) \wedge$ 
     $(\forall \text{comp} : \text{int}) (0 \leq \text{comp} < \text{length}(c.\text{indices}) \rightarrow_L \text{indiceValido}(c.\text{conj}, \text{indices}[\text{comp}], \text{comp}))$ 
  }
  pred predAbs (c1: Conjunto<TuplaEnt>, c2: Conjunto<tupla< $\mathbb{Z}$ ,  $\mathbb{Z}$ ,  $\mathbb{Z}$ >>) {
     $(\forall e : \mathbb{Z}) (e \in c2.\text{elems} \iff (\exists i : \text{int}) (0 \leq i < \text{length}(c1.\text{conj}) \wedge_L c1.\text{conj}[i] = e))$ 
  }
}

```

```
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

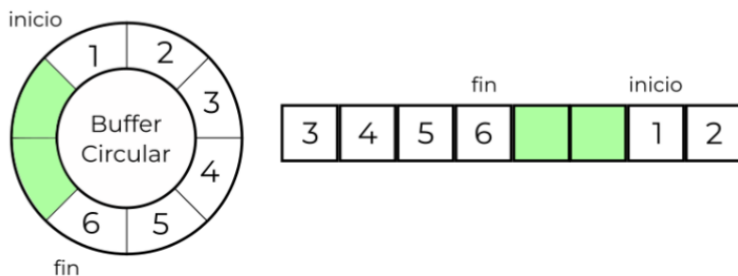
        i := i + 1;
    endwhile

    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 máximo 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!