



Guia 7

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 7	3
1.1. Ejercicio 1	3
1.2. Ejercicio 2	4
1.3. Ejercicio 3	7

1. Guía 7

1.1. Ejercicio 1

Implementamos un **Árbol Binario** (AB)

- Escriba en castellano el invariante de representación para este módulo
- Escriba en lógica el invrep usando preds recursivos
- Escriba los algoritmos para los siguientes procs y, de ser posible, calcule su complejidad
 - altura(in ab: ArbolBinario<T>): int
 - cantidadHojas(in ab: ArbolBinario<T>): int
 - está(in ab: ArbolBinario<T>, in t: T): bool
 - cantidadApariciones(in ab: ArbolBinario<T>, in t: T): int

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

Módulo ArbolBinario<T> implementa Arbol Binario<T> {
var raiz: Nodo<T>

InvRep: No tiene ciclos y la raiz es null o el subarbol derecho y el izquierdo son AB

pred esAB (r: Nodo<T>) {
r = null \vee_L (esAB(r.der) \wedge esAB(r.izq))
}

pred InvRep (ab: ArbolBinario<T>) {
sinCiclos(ab.raiz) \wedge esAB(ab.raiz)
}

proc altura (in ab: ArbolBinario<T>) : int
Complejidad: $O(n) \leftarrow n = \text{cantNodos}$
return altura(ab.raiz);

proc alturaAux (in r: Nodo<T>) : int
if r = null then
return 0;
endif
return 1 + max(alturaAux(r.izq), alturaAux(r.der));

proc cantidadHojas (in ab: ArbolBinario<T>) : int
Complejidad: $O(n)$
return cantidadHojasAux(ab.raiz);

proc cantidadHojasAux (in r: Nodo<T>) : int
if r = null then
return 0;
else if r.izq = null && r.der = null then
return 1;
endif
return cantidadHojasAux(r.izq) + cantidadHojasAux(r.der);

```

proc esta (in ab: ArbolBinario<T> ) : Bool
    Complejidad:  $O(n)$ 
    return estaAux(r.raiz, e);

proc estaAux (in r: Nodo<T>, in e: T) : Bool
    if r == null then
        return false;
    else if r.dato == e then
        return true;
    else
        return estaAux(r.izq, e) || estaAux(r.der, e);
    endif

proc cantidadApariciones (in ab: ArbolBinario<T> , in e: T) : int
    Complejidad:  $O(n)$ 
    return cantidadAparicionesAux(ab.raiz, e);

proc cantidadAparicionesAux (in r: Nodo<T>, in e: T) : int
    var cant: int;
    cant := 0;

    if r == null then
        return 0;
    else if r.dato == e then
        cant := 1;
    endif

    return cant + cantidadAparicionesAux(r.izq, e) + cantidadAparicionesAux(r.der, e);
}

```

1.2. Ejercicio 2

Un **Árbol Binario de Búsqueda** (ABB) es un árbol binario que cumple que para cualquier nodo N, todos los elementos del árbol a la izquierda son menores o iguales al valor del nodo y todos los elementos del árbol a la derecha son mayores al valor del nodo, es decir

```

pred esABB (a: Nodo<T>) {
    a = null ∨ (
        (∀e : T) (e ∈ elems(a.izq) → e ≤ a.dato) ∧ (∀e : T) (e ∈ elems(a.der) → e > a.dato) ∧
        esABB(a.izq) ∧ esABB(a.der)
    )
}
aux elems (a: Nodo<T>) : conj<T> = IfThenElse(a = null, ∅, {a.dato} ∪ elems(a.izq) ∪ elems(a.der));

```

- Implemene los algoritmos para los siguientes procs y calcule su complejidad en mejor y peor caso
 1. está(in ab: ABB<T>, in t: T): bool
 2. cantidadApariciones(in ab: ABB<T>, in t: T): int
 3. insertar(inout ab: ABB<T>, in t: T)
 4. eliminar(inout ab: ABB<T>, in t: T)
 5. inOrder(in ab: ABB<T>): Array<T>
- Asumiendo que el árbol está balanceado, recalculé, si es necesario, las complejidades en peor caso de los algoritmos del ítem anterior

- ¿Qué pasa en un ABB cuando se insertan valores repetidos? Proponga una modificación del módulo que resuelva este problema

```

Módulo ABB<T> implementa Arbol Binario De Busqueda<T> {
  var raiz: Nodo<T>
  var size: int
  pred InvRep (ab: ABB<T> ) {
    esABB(ab.raiz)  $\wedge$  ab.size = cantNodos(ab.raiz)
  }
  aux cantNodos (r: Nodo<T>) : int = IfThenElse(r = null, 0, 1+cantNodos(r.izq)+cantNodos(r.der));
  proc nuevoABB () : ABB<T>
    Complejidad:  $O(1)$ 
    var ab: ABB< T >;
    ab := new ABB< T >;
    ab.raiz := new Nodo< T >;
    ab.size := 0;
    return ab;

  proc esta (in ab: ABB<T> , in t: T) : Bool
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n)$ 
    return estaAux(ab.raiz, t);

  proc estaAux (in r: Nodo<T>, in t: T) : Bool
    if r.dato == t then
      return true;
    else if t > r.dato then
      return estaAux(r.der, t);
    else
      return estaAux(r.izq, t);
    endif

  proc cantidadApariciones (in ab: ABB<T> , in t: T) : int
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n)$ 
    return cantidadAparicionesAux(ab.raiz, t);

  proc cantidadAparicionesAux (in r: Nodo<T>, in t: T) : int
    if r == null then
      return 0;
    else if r.dato == t then
      return 1 + cantidadAparicionesAux(r.izq, t);
    else if r.dato > t then
      return cantidadAparicionesAux(r.der, t);
    else
      return cantidadAparicionesAux(r.izq, t);
    endif

  proc insertar (inout ab: ABB<T> , in t: T)
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n)$ 
    ab.raiz := insertarAux(ab.raiz, t);
    ab.size := ab.size + 1;

```

```
proc insertarAux (inout r: Nodo<T>, in t: T) : Nodo<T>
```

```

  if r == null then
    r := new Nodo < T >;
    r.dato := t;
  else if t <= r.dato then
    r.izq := insertarAux(r.izq, t);
  else
    r.der := insertarAux(r.der, t);
  endif

  return r;

```

```
proc eliminar (inout ab:  $ABB\langle T \rangle$ , in t: T)
```

Mejor caso: $\Theta(1)$

Peor caso: $\Theta(n)$

```

  if ab.esta(t) then
    ab.raiz := eliminarAux(ab.raiz, t);
    ab.size := ab.size - 1;
  endif

```

```
proc eliminarAux (inout r: Nodo<T>, in t: T) : Nodo<T>
```

```

  if r == null then
    return null
  else if r.dato == t then
    if r.izq != null && r.der != null then
      r.dato = minimo(r.der);
      r.der = eliminarAux(r.der, r.dato);
    else if r.izq != null then
      return r.izq;
    else
      return r.der;
    endif
  else if t > r.dato then
    r.der := eliminarAux(r.der, t);
  else
    r.izq := eliminarAux(r.izq, t);
  endif

  return r;

```

```
proc minimo (in r: Nodo<T>) : T
```

```

  while r.izq != null do
    r := r.izq;
  endwhile

  return r.dato;

```

```
proc inOrder (in ab:  $ABB\langle T \rangle$ ) : Array < T >
```

Mejor caso: $\Theta(1)$

Peor caso: $\Theta(n)$

```

  var cola: ColaSobreLista < T >;
  cola := colaVacía();

```

```
inOrderAux(cola, ab.raiz);
```

```
return colaAArray(cola, ab.size);
```

```
proc inOrderAux (inout c: ColaSobreLista<T>, in r: Nodo<T>)
```

```
  if r == null then
```

```
    return;
```

```
  endif
```

```
  inOrderAux(c, r.izq);
```

```
  c.encolar(r.dato);
```

```
  inOrderAux(c, r.der);
```

```
proc colaAArray (inout c: ColaSobreLista<T>, in size: int) : Array < T >
```

```
  var res: Array < T >;
```

```
  var i: int;
```

```
  res := new Array < T >(size);
```

```
  i := 0;
```

```
  while !c.vacia() do
```

```
    res[i] := c.desencolar();
```

```
    i := i + 1;
```

```
  endwhile
```

```
  return res;
```

```
proc cantidadNodos (in ab: ABB<T>) : int
```

```
  return ab.size;
```

```
}
```

Si el árbol está balanceado entonces la complejidad en el peor caso de los algoritmos está, insertar y eliminar pasa a ser $\Theta(\log n)$

Si se insertan valores repetidos, según el enunciado, estos se insertarán en el subárbol izquierdo del nodo con ese mismo valor (no se cual sería el problema)

1.3. Ejercicio 3

Implementar los siguientes TADs sobre ABB. Calcule las complejidades de los procs en mejor y peor caso

1. Conjunto<T>
2. Diccionario<K, V>
3. ColaDePrioridad<T>

Recalcule, si es necesario, las complejidades en peor caso de los algoritmos de los TADs considerando que se implementan sobre AVL en vez de ABB.

```
Módulo ConjuntoABB<T> implementa Conjunto<T> {
```

```
  var elems: ABB<T>
```

```
  proc conjVacio () : ConjuntoABB<T>
```

```
    Complejidad:  $O(1)$ 
```

```
  var conj: ConjuntoABB < T >;
```

```
  conj := new ConjuntoABB < T >;
```

```
  conj.elems := nuevoABB();
```

```
  return conj;
```

```

proc pertenece (in c: ConjuntoABB $\langle T \rangle$  , in e: T) : Bool
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n)$ 

    return c.elems.esta(e);

proc agregar (inout c: ConjuntoABB $\langle T \rangle$  , in e: T)
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n)$ 

    if !c.elems.pertenece(e) then
        c.elems.insertar(e);
    endif

proc sacar (inout c: ConjuntoABB $\langle T \rangle$  , in e: T)
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n)$ 

    c.elems.eliminar(e);

proc unir (inout c1: ConjuntoABB $\langle T \rangle$  , in c2: ConjuntoABB $\langle T \rangle$  )
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(nm)$ 

    var elems: Array < T >;
    var i: int;
    elems := c2.elems.inOrder();
    i := 0;

    while i < elems.length() do
        c1.elems.agregar(elems[i]);
        i := i + 1;
    endwhile

proc restar (inout c1: ConjuntoABB $\langle T \rangle$  , in c2: ConjuntoABB $\langle T \rangle$  )
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(nm)$ 

    var elems: Array < T >;
    var i: int;
    elems := c2.elems.inOrder();
    i := 0;

    while i < elems.length() do
        c1.elems.eliminar(elems[i]);
        i := i + 1;
    endwhile

```



```

proc intersecar (inout c1: ConjuntoABB $\langle T \rangle$  , in c2: ConjuntoABB $\langle T \rangle$  )
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n^2 + nm)$ 

    var elems: Array < T >;
    var i: int;
    elems := c1.elems.inOrder();
    i := 0;

    while i < elems.length() do
        if !c2.elems.pertenece(elems[i]) then
            c1.elems.eliminar(elems[i]);
        endif
        i := i + 1;
    endwhile

proc agregarRapido (inout c: ConjuntoABB $\langle T \rangle$  , in e: T)
    Mejor caso:  $\Theta(1)$ 
    Peor caso:  $\Theta(n)$ 

    c.elems.insertar(e);

proc tamano (in c: ConjuntoABB $\langle T \rangle$  ) : int
    Complejidad:  $\Theta(1)$ 

    return c.elems.cantidadNodos();

}

```

El diccionario sobre ABB tiene basicamente los mismos procs que el conjunto solo que en vez de usar $\text{ABB}\langle T \rangle$ usamos $\text{ABB}\langle \text{struct}\langle \text{clave} : K, \text{valor} : V \rangle \rangle$ con K un tipo comparable.