DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

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

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$\langle dato : T, izq : Nodo, der : Nodo \rangle$

```
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 ∨_L (esAB(r.der) ∧ esAB(r.izq))
    }
    pred InvRep (ab: ArbolBinario⟨T⟩ ) {
        sinCiclos(ab.raiz) ∧ esAB(ab.raiz)
    }
    proc altura (in ab: ArbolBinario⟨T⟩ ) : int
        Complejidad: O(n) ← n = 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, recalcule, si es necesario, las complejidads 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) ∧ 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)

        res.raiz := new Nodo < T >;
        res.size := 0;
        return res;

    proc esta (in ab: ABB⟨T⟩ , in t: T) : Bool
        Mejor caso: Θ(1)
        Peor caso: Θ(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: Θ(1)
        Peor caso: Θ(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: Θ(1)
        Peor caso: Θ(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⟨T⟩ , in t: T)
    Mejor caso: Θ(1)
    Peor caso: Θ(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⟨T⟩ ) : Array < T >
    Mejor caso: Θ(1)
    Peor caso: Θ(n)

    var cola: ColaSobreLista < T >;
    cola := colaVacia();
```

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

Si se insertar valores repetidos, según el enunciado, estos se insertaran en el subarbol 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)

        res.elems := nuevoABB();
        return res;
```

```
proc pertenece (in c: ConjuntoABB⟨T⟩ , in e: T) : Bool
        Mejor caso: Θ(1)
        Peor caso: Θ(n)

    return c.elems.esta(e);
```

```
proc agregar (inout c: ConjuntoABB⟨T⟩ , in e: T)
        Mejor caso: Θ(1)
        Peor caso: Θ(n)

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

```
proc sacar (inout c: ConjuntoABB⟨T⟩ , in e: T)
        Mejor caso: Θ(1)
        Peor caso: Θ(n)

    c.elems.eliminar(e);
```

```
proc unir (inout c1: ConjuntoABB⟨T⟩ , in c2: ConjuntoABB⟨T⟩ )
        Mejor caso: Θ(1)
        Peor caso: Θ(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⟨T⟩ , in c2: ConjuntoABB⟨T⟩ )
        Mejor caso: Θ(1)
        Peor caso: Θ(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⟨T⟩ , in c2: ConjuntoABB⟨T⟩ )
        Mejor caso: Θ(1)
        Peor caso: Θ(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⟨T⟩ , in e: T)
        Mejor caso: Θ(1)
        Peor caso: Θ(n)

    c.elems.insertar(e);

proc tamano (in c: ConjuntoABB⟨T⟩ ) : int
        Complejidad: Θ(1)

    return c.elems.cantidadNodos();
```

}

El diccionario sobre ABB tiene basicamente los mismos procs que el conjunto solo que en vez de usar ABB<T> usamos ABB<struct⟨$clave : K, valor : V$⟩> con K un tipo comparable.

1.4. Ejercicio 4

HACER!

1.5. Ejercicio 5

Escriba un algoritmo que verifique si un árbol binario cumple con la propiedad de max-heap

```
proc esMaxHeap (in ab: ArbolBinario) : Bool
    return esMaxHeapAux(ab.raiz)

proc esMaxHeapAux (in r: Nodo<T>) : Bool
    var esMayorQueHijos: bool

    if r == null then
        return true;
    endif

    esMayorQueHijos := (r.izq == null || r.dato >= r.izq.dato) &&
                       (r.der == null || r.dato >= r.der.dato);

    return esMayorQueHijos && esMaxHeapAux(r.izq) && esMaxHeapAux(r.der)
```

1.6. Ejericio 6

Implemente el TAD ColaDePrioridad<T> utilizando heaps (implementados con arreglos).

1. Escriba en castellano y en lógica el inverp
2. Escriba los algoritmos para las operaciones encolar, desencolarMax y cambiarPrioridad. Justifique la complejidad de cada operación.

Módulo ColaDePrioridadHeap<T> implementa ColaDePrioridad<T> {
var elems: $Vector < \text{struct}\langle dato : T, pri : \mathbb{R}\rangle >$
InvRep: Para todo i en rango de $elems$

- Si $2i + 1$ está en rango de $elems$ entonces $elems[i].pri \geq elems[2i + 1].pri$
- Si $2i + 2$ está en rango de $elems$ entonces $elems[i].pri \geq elems[2i + 2].pri$

pred InvRep (c: $ColaDePrioridadHeap\langle T\rangle$) {
$(\forall i : \text{int})\ (0 \leq i < c.elems.s.length \rightarrow_L$
$(2i + 1 < c.elems.s.length \rightarrow_L c.elems[i].pri \geq c.elems[2i + 1].pri) \wedge$
$(2i + 2 < c.elems.s.length \rightarrow_L c.elems[i].pri \geq c.elems[2i + 2].pri))$
}

proc swap (inout v: Vector<struct$\langle dato : T, pri : \mathbb{R}\rangle$>, in i1: int, in i2: int)

```
var aux: Struct< dato: T, pri: float >;
aux := v[i1];
v[i1] := v[i2];
v[i2] := aux;
```

proc siftDown (inout v: Vector<struct$\langle dato : T, pri : \mathbb{R}\rangle$>, in i: int)

```
var tieneHijoIzq: bool;
var tieneHijoIzq: bool;
tieneHijoIzq := 2*i + 1 < v.longitud();
tieneHijoDer := 2*i + 2 < v.longitud();

if tieneHijoIzq && tieneHijoDer && (v[i].pri < v[2*i + 1].pri || v[i].pri < v[2*i + 2].pri)
    then

    if v[2*i + 1].pri > v[2*i + 2].pri then
        swap(v, i, 2*i + 1);
        siftDown(v, 2*i + 1);
    else
        swap(v, i, 2*i + 2);
        siftDown(v, 2*i + 2);
    endif
else if tieneHijoIzq && v[i].pri < v[2*i + 1].pri then
    swap(v, i, 2*i + 1);
    siftDown(v, 2*1 + 1);
else if tieneHijoDer && v[i].pri < v[2*i + 2].pri then
    swap(v, i, 2*i + 2);
    siftDown(v, 2*i + 2);
endif
```

```
proc siftUp (inout v: Vector<struct⟨dato : T, pri : ℝ⟩>, in i: int)
    var indPadre: int;
    indPadre := (i − 1) / 2;
    if i != 0 && v[i].pri > v[indPadre].pri then
        swap(v, i indPadre);
        siftUp(v, indPadre);
    endif
```

```
proc encolar (inout c: ColaDePrioridadHeap⟨T⟩ , in e: T, in pri: ℝ)
```
Complejidad: $O(\log n)$

En realidad si el elemento se inserta en un nuevo nivel entonces el vector se redimensionaria, por lo que sería $O(n)$.

```
    c.elems.agregarAtras(new Struct(dato: e, pri: pri));
    siftUp(c.elems, c.elems.longitud() − 1);
```

```
proc desencolarMax (inout c: ColaDePrioridadHeap⟨T⟩ ) : T
```
Complejidad: $O(\log n)$

Eliminar un elemento del vector es $O(n)$ porque hay que mover todas las demás posiciones, sin embargo eliminar el último elemento de un vector sería $O(1)$ (?)

```
    var res: T;
    res := c.elems[0].dato;
    c.elems[0] := c.elems[c.elems.longitud() − 1];
    c.elems.eliminar(c.elems.longitud() − 1);
    siftDown(c.elems, 0);
```

```
proc cambiarPrioridad (inout c: ColaDePrioridadHeap⟨T⟩ , in e: T, in pri: ℝ)
```
Complejidad: $O(n)$

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

    while i < c.elems.longitud() && c.elems[i].dato != e do
        i := i + 1;
    endwhile

    if i < c.elems.longitud() then
        c.elems[i].pri := pri;
        siftUp(c.elems, i);
    endif
}
```

1.7. Ejercicio 7

¿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 apellido? Describa todos los cambios necesarios a la implementación del ejercicio anterior. HACER!

1.8. Ejercicio 8

Suponiendo que el metodo Array2MinHeap se encuentra dentro del modulo

```
proc Array2MinHeap (in arr: Array<T>) : MinHeap<T>
        var i: int;
        i := 0;

        res.elems := arr.copy();

        while i < (arr.length() + 1) / 2 do
            siftDown(res.elems, i);
            i := i + 1;
        endwhile

        return res;

proc sortArr (inout arr: Array<T>)
        var h: MinHeap< T >;
        var i: int;

        h := Array2MinHeap(arr);
        i := 0;

        while i < arr.length() do
            arr[i] := h.desencolarMin();
            i := i + 1;
        endwhile
```

1.9. Ejercicio 9

HACER!

1.10. Ejercicio 10

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

- Describa en castellano el invrep
- 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?

Para la implementación del Trie me parecio adecuado hacer una implementación mas tipo diccionario ya que es facilmente adaptable a una implementaciónde conjunto, lo cual no es así al revés.

T es un tipo iterable de E

Nodo<E, K>es Struct<

alfabeto: Array<Nodo<E, K>>,

valor: K

>

Módulo TrieArr<T, K> implementa Trie<T> {

var raiz: Nodo<E, K>

var sizeAlfabeto: int

InvRep

- Cada nodo tiene un único padre
- La raíz no tiene padre
- Las hojas tiene un valor definido
- La raíz no tiene valor
- Cada nodo tiene alfabeto definido
- El tamaño de alfabeto de cada nodo es sizeAlfabeto

proc buscar (in t: $TrieArr\langle T, K\rangle$, in e: T) : K

Complejidad: $O(|e|)$

```
var actual: Nodo < E, K >;
var i: int;
actual := t.raiz;
i := 0;

while i < e.length() && actual.alfabeto[e[i]] != null do
    actual := actual.alfabeto[e[i]];
    i := i + 1;
endwhile

if i == e.length() && actual.valor != null then
    return actual.valor;
endif
```

proc agregar (inout t: $TrieArr\langle T, K\rangle$, in e: T, in k: K)

Complejidad: $O(|e|)$

```
var actual: Nodo < E, K >;
var i: int;
actual := t.raiz;
i := 0;

while i < e.length() do
    if actual.alfabeto[e[i]] == null then
        var n: Nodo < E, K >;
        n := new Nodo < E, K >;
        n.alfabeto := new Array < E >(t.sizeAlfabeto);
        actual.alfabeto[e[i]] := n;
        actual := n;
    else
        actual := actual.alfabeto[e[i]];
    endif

    i := i + 1;
endwhile

actual.valor := k;
```

}

Nodo<E, K>es Struct<
hijo: Nodo<E, K>,
hermano: Nodo<E, K>,
valor: K,
valorAlf: E
>

`Módulo TrieList<T> implementa Trie<T>` {

var raiz: Nodo<E, K>

InvRep:

- Cada nodo tiene valorAlf definido
- Cada nodo tiene un solo padre excepto la raíz
- Si un nodo no tiene hijo entonces tiene un valor definido
- Todos los valorAlf son distintos entre hermanos

`proc buscar` (in t: $TrieList\langle T\rangle$, in e: T) : K

Complejidad: $O(n)$ ← n cantidad de claves
En el peor de los casos todos las claves empiezan con una letra distinta y la palabra se encuentra en el último hermano

```
var actual: Nodo < E, K >;
var i: int;
actual := t.raiz;
i := 0;

while i < e.length() && actual != null do
    if actual.valorAlf != e[i] then
        actual := actual.hermano;
    else
        if i != e.length() - 1 then
            actual := actual.hijo;
        endif
        i := i + 1;
    endif
endwhile

return actual.valor;
```

`proc agregar` (inout t: $TrieList\langle T\rangle$, in e: T, in k: K)

HACER!

Creo que me conviene cambiar la representación para que la raíz sea un nodo que tengo como hijo una letra, y no que la raíz sea la primera letra

}

1.11. Ejercicio 11

Utilizando una estrucutra de Trie para almacenar palabras, escriba los 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. buscarIntervalo: Dadas dos palabras $p1$ y $p2$, devolver todas las palabras que se encuentren entre $p1$ y $p2$ en orden lexicográfico, ordenadas.

HACER!