

Guía Control 3

CC10A

Esta guía no tiene por objetivo enseñar mucho, solo tener algunos algoritmos ordenados y métodos útiles para copiar si se necesitan en el control.

Métodos de Ordenamiento

Los siguientes métodos están hechos para ordenar arreglos, algunos de números (int) y otros de palabras (String), si entienden el algoritmo, solo tendrán que hacer unos pequeños cambios en las líneas para ordenar otras cosas, como por ejemplo listas enlazadas (lo cual queda propuesto).

Primero defino el siguiente método que me servirá para hacer los intercambios de valores entre dos posiciones. Este lo uso en los 4 algoritmos.

- Intercambio.

```
static public void intercambio(String[]x,int i, int j){
    String aux=x[i];
    x[i]=x[j];
    x[j]=aux; }
```

Los 4 algoritmos están definidos de manera recursiva, pero el de fuerza bruta y de burbuja pueden ser escritos también de manera iterativa.

- Fuerza Bruta

```
static void fuerzabruta(int[] r, int i, int f){
    //caso base
    if (i == f) return;

    //busco "la posición" del mayor
    int l = f;
    int pos=i;
    for(int j=i; j<= f; j++)
        if(r[j]>r[pos]) pos = j;

    // Cambio el mayor con el último. Fíjense que si el mayor
    //es justamente el ultimo, no hay problema intercambio(r, pos, f);

    fuerzabruta(r,i,f-1);
}
```

- Burbuja

```
static public void burbuja(int[] r, int i, int f)
{
    //caso base
    if(i == f) return;

    //luego se hacen las comparaciones de cada uno con
```

```

        // su sucesor. Cuidado con salirse del arreglo
        for(int j=0; j<f; j++)
            if(r[j] > r[j+1]) intercambio (r, j, j+1);

        burbuja (r,i,f-1);
    }

```

• MergeSort

```

static public void mergesort(String[]x, int ip, int iu)
{
    // caso base
    if(ip >= iu) return;

    // Para ordenarlo, dividimos el problema en dos
    int im = (ip + iu)/2;
    mergesort(x,ip,im);
    mergesort(x,im+1,iu);

    // Reunimos las 2 partes
    merge(x,ip,im,iu);
}

static public void merge(String[]x,int ip, int im, int iu)
{
    String[]aux = new String[iu+1];
    int i1=ip, i2=im+1;
    for(int i=ip; i<=iu; ++i){
        if( i1<=im && (i2>iu || x[i1].compareTo(x[i2]) < 0))
            aux[i] = x[i1++];
        else
            aux[i] = x[i2++];
    }
    for(int i=ip; i<=iu; ++i)
        x[i] = aux[i];
}

```

• QuickSort

```

static public void quicksort(String[]x, int ip, int iu)
{
    //caso base
    if(ip >= iu) return;

    //Escojo un pivote y al mismo tiempo dejo a la izquierda
    // los menores y a la derecha los mayores que el pivote
    int i = particionar(x,ip,iu);
    quicksort(x,ip,i-1);
    quicksort(x,i+1,iu);
}

static public int particionar(String[]x, int ip, int iu)
{
    String pivote = x[ip];
    int i = ip;

```

```

        for(int j=ip+1; j<=iu; ++j)
        if( x[j].compareTo(x[ip]) < 0 )
            intercambio(x,++i,j);
        intercambio(x,ip,i);
        return i;
    }

```

Para demostrar que esto es “modificable” a otras estructuras, como listas enlazadas, acá va parte del metodo QuickSort para listas

```

static public void quicksort(Lista x, int ip, int iu){
    //caso base
    if(ip >= iu) return;

    int i = particionar(x,ip,iu);
    quicksort(x,ip,i-1);
    quicksort(x,i+1,iu);
}

static public void quicksort(Lista x){
    //Necesito largo de la lista
    int cont = 0;
    Nodo aux = x.primerono;
    while(aux != null){
        cont++;
        aux = aux.sgte;
    }
    quicksort(x,0,cont);
}

static public int particionar(Lista x){
    //Necesito largo de la lista
    int cont = 0;
    Nodo aux = x.primerono;

    while(aux != null){
        cont++;
        aux = aux.sgte;
    }
    return particionar(x,0,cont-1);
}

static public int particionar(Lista x, int ip, int iu){
    Object pivote = x.buscar(ip);
    int i=ip;
    for(int j=ip+1; j<=iu; ++j)
        if( x.buscar(j).compareTo(x.buscar(ip)) < 0 )
            intercambio(x,++i,j);
    intercambio(x,ip,i);
    return i;
}

```

Nota: Quedan propuestos los métodos intercambio y buscar.

Listas Ordenadas

La información guardada en cada Nodo puede ser de cualquier tipo (int, double, String, etc.) y más de una. Si cambian el tipo de información, recuerden modificar la forma de hacer las comparaciones.

- **Nodo Típico**

```
public NodoLista sgte;  
public int valor;  
public NodoLista(int x, NodoLista y)  
{  
    valor = x; sgte = y;  
}
```

- **Una forma de escribir una representación de una lista en un String**

```
public String toString()  
{  
    String resp = valor + " " ;  
    if(sgte != null) resp = resp + ";" + sgte.toString();  
    return resp;  
}
```

- **Para agregar un nuevo nodo a la lista**

```
public void agregar(NodoLista raiz, int v)  
{  
    NodoLista nuevo = new NodoLista(v,null);  
    NodoLista aux = raiz;  
  
    //De esta forma recorro la lista y me "ubico"  
    // en un nodo  
    while(aux.sgte != null){  
        aux = aux.sgte  
    }  
    aux.sgte = nuevo;  
}
```

Árbol de Búsqueda Binaria (ABB)

Los árboles de búsqueda binaria en el fondo son listas enlazadas con 2 enlaces en cada Nodo que cumplen la condición (Invariante) de que el hijo de la derecha es mayor que el padre y el de la izquierda es menor.

- **Nodo Típico**

```
public NodoArbol izq;  
public NodoArbol der;  
public int valor;  
public NodoArbol(int val, NodoArbol i, NodoArbol d)  
{  
    valor = val;  
    izq = i;  
    der = d;  
}
```

- **Una forma de escribir una representación del árbol en un String**

```
public String toString()  
{  
    String i = "";  
    String d = "";  
    if(izq != null) i = izq.toString();  
    if(der != null) d = der.toString();  
    return "" + valor + ", i=" + i + ", d=" + d;  
}
```

- **Para agregar un valor a un nodo nuevo a uno ya existente. Retorna una referencia al nodo padre.**

```
static public NodoArbol agregar(int val, NodoArbol arbol)  
{  
    if(arbol == null)  
        return new NodoArbol(val, null, null);  
  
    if(arbol.valor == val)  
        return arbol; // Ya esta  
  
    if(val < arbol.valor)  
        arbol.izq = agregar(val, arbol.izq);  
    else  
        arbol.der = agregar(val, arbol.der);  
    return arbol;  
}
```

- **Método para contar hojas (nodos sin hijos)**

```
public int hojas(NodoArbol x)  
{  
    if(x == null) return 0;  
    if(x.der == null && x.izq == null) return 1;  
    return ( hojas(x.der) + hojas(x.izq) );  
}
```

- Método para obtener la altura de un árbol (las hojas tienen altura 0);

```
public int altura(NodoArbol x)
{
    if(x == null) return 0;
    return 1 + Math.max(altura(x.der), altura(x.izq));
}
```

- Boolean para saber si un árbol es ABB o no.

```
public String sucesor(NodoArbol x)
{
    if(x == null) return null;

    if(x.der == null) return x.valor;

    Nodo aux = x.der;

    while(true){
        if(aux.izq == null) break;
        aux = aux.izq;
    }
    return aux.valor;
}
```

- Boolean para saber si un árbol es ABB o no.

```
public boolean esABB(NodoArbol x)
{
    if( x == null) return true;

    if( x.der == null && x.izq == null) return true;

    if( x.izq != null && (x.valor.compareTo(x.izq.valor)<0))
        return false;

    if( x.der != null && (x.valor.compareTo(x.der.valor)>0))
        return false;

    return esABB(x.der) && esABB(x.izq);
}
```

- Estos 2 últimos métodos son un poco más complicados. Son para intercambiar de posición a un nodo padre con uno de sus hijos, siempre manteniendo el Invariante de un ABB.

```
public void rotaIzq(NodoArbol padre)
{
    if (this.izq == null) return;

    if (padre.izq == this) padre.izq = this.izq;

    else padre.der = this.izq;

    NodoArbol aux = this.izq;
```

```
        this.izq = aux.der;
        aux.der = this;
    }
    public void rotaDer(NodoArbol padre)
    {
        if (this.der == null) return;

        if (padre.der == this) padre.der = this.der;

        else padre.izq = this.der;

        NodoArbol aux = this.der;
        this.der = aux.izq;
        aux.izq = this;
    }
}
```

Cualquier duda o comentario, manden un mail a angperez@dcc.uchile.cl