**Algoritmos de ordenación**

Estructuras que podemos ordenar: arrays y lista (árboles después).

Arrays: recordar como declararlo

Type

Arreglo = array [rango] of tipo;

Var

v:arreglo;

Lista: recordar como declararlo

Type

Lista = ^nodo;

Nodo = record

dato:tipo;

sig:lista;

end;

Var

l:lista;

**Método de ordenación: selección**

Permite ordenar una estructura ESTÁTICA, como es un vector. Tarda n2 en ejecutarse, pero es de los más fáciles de implementar. En la vida real no es de los más utilizados.

Consiste en buscar en cada vuelta en qué posición está el mínimo y al finalizar la vuelta intercambiar el elemento encontrado con el primero no ordenado.

En *primera vuelta,* se encuentra el más chico y se lo intercambia con el primer elemento.

En *segunda vuelta,* se encuentra el siguiente más chico y se lo intercambia con el 2do.

Etc.

Se detiene en la vuelta dimL -1 porque el último elemento es obviamente el más grande.

Cómo lo implemento?

Program ordenar;

Const dimF = … {máxima longitud del arreglo}

Type

TipoElelem = … {tipo de datos del vector}

Indice = 0..dimF;

Tvector = array [1..dimF] of TipoElem;

Var

a:tVector;

dimL:integer;

Begin

cargarVector(a,dimL);

seleccion(a,dimL); {envío el arreglo, que es lo que quiero ordenar, y la

dimL así sabe cuántas veces debe ejecutar}

End.

Procedure seleccion (var v:tVector; dimLog: índice);

var

i,j,pos: índice;

item: tipoElem;

begin

for i:=1 to dimLog do begin { busca el min y guarda en pos la posicion}

pos:= i;

for j := i+1 to dimLog do

if v[j] < v[pos] then pos := j;

{ intercambia v[i] y v[pos] }

item := v[pos];

v[pos] := v[i] ;

v[i] := item ;

end;

end;

Si el arreglo viene ordenado, este algoritmo igual toma el mismo tiempo porque intercambia los lugares igual! No controla que el vector ya esté ordenado, y mientras crece el tamaño del array el tiempo de ejecución crece de manera cuadrática.

**Método de ordenación: inserción**

También tiene un orden de ejecución de n2 y es ineficiente en tanto a tiempo de ejecución – es un poco más difícil de implementar que el de selección.

También es **solo para arrays.**

En la *primera vuelta* se toma solo el primer elemento, que considera ordenado.

En la *segunda vuelta* compara elem 1 con 2, y los ordena.

En la *tercera vuelta* compara el elemento 3 y lo ubica donde vaya.

Etc.

Cómo lo implemento?

Program ordenar;

Const dimF = … {máxima longitud del arreglo}

Type

TipoElelem = … {tipo de datos del vector}

Indice = 0..dimF;

Tvector = array [1..dimF] of TipoElem;

Var

a:tVector;

dimL:integer;

Begin

cargarVector(a,dimL);

insercion(a,dimL); {envío el arreglo, que es lo que quiero ordenar, y la

dimL así sabe cuántas veces debe ejecutar}

End.

Procedure insercion(var v:tVector; dimLog: índice);

var

i,j: índice;

actual: tipoElem;

begin

for i:=2 to dimLog do begin { el primero ya está ordenado}

actual := v[i];

j:= i-1; { auxiliar }

while (j > 0) and (v[j] > actual) do begin

v[j+1] := v[j];

j:= j-1;

end;

v[j+1] := actual;

end;

end;

Si los datos están ordenados de menor a mayor el algoritmo corta la búsqueda en el while y no intercambia nada, solo hace comparaciones. Si están todos ordenados el orden va a ser n, en vez de n2 como el otro.

**No hay algoritmos de ordenación aplicados a listas! Lo que hago es agregar ordenado a la lista.**

**Cómo generar un número random:**

Primero tengo que invocar en el código la funcionalidad de Pascal:

Randomize;

Luego puedo usarla:

v.cant := Random(15) +1;

Esto genera un número del 1 al 15. Por qué el +1? Porque Random genera un número del 0 al n-1 (14 en este caso).

**Recursión**

Consiste en reducir el problema en instancias más pequeñas del mismo problema hasta llegar al “base”, que se puede resolver de forma trivial.

¿Es eficiente? Depende del punto de vista. El módulo se llama a si mismo por ende se carga en memoria tantas veces como se re-llame. El método iterativo sólo lo carga una vez. Es una manera alternativa donde la resolución es más clara, no necesariamente más fácil o eficiente.

Un procedimiento recursivo puede tener uno o más casos bases, no suelen ser más de dos.

Por ejemplo para imprimir una lista:

Procedure imprimir (pri:lista);

Begin

If (pri <> nil) do begin

write(pri^.dato);

pri:=pri^.sig;

**imprimir(pri);**

end;

End;

Para calcular una potencia:

Function potencia (x,n:integer):integer;

Begin

If (n=0) then potencia:=1

else if (n=1) then potencia := x

else

potencia:= x\*potencia(x,n-1);

end;

End;

Si quiero imprimir números de N a 1:

Procedure imprimirRecursivos(n:integer);

begin

if (n > 1) then

imprimirRecursivos(n-1);

writeln(N);

end;

Si por ejemplo ingreso n=3:

Entra. n > 1? Si, entonces entra de nuevo con n = 2.

n > 1? Si, entonces entra de nuevo con n = 1

n > 1? No! Entonces no ejecuta la recursión de nuevo, pasa a la sgte instrucción: imprime 1.

Vuelve a la instancia anterior de recursión. Como n se pasó por valor y no por referencia, acá vale 2. Imprime 2.

Vuelve a la instancia anterior, n vale 3. Imprime 3.

Termina el programa, imprimió: 1 2 3.

Cómo evaluar un ejercicio con recursivo para trabajarlo:

Por ejemplo, si tengo que imprimir un vector. ¿Cuál es el caso base? Uno puede ser vector vacío. Ahí no tengo que hacer nada, así que puedo trabajar con un if dimL > 0, y si es falso no hace nada. Entro al if:

Procedure imprimirVectorRecursivo (v:vector; dimL:integer);

begin

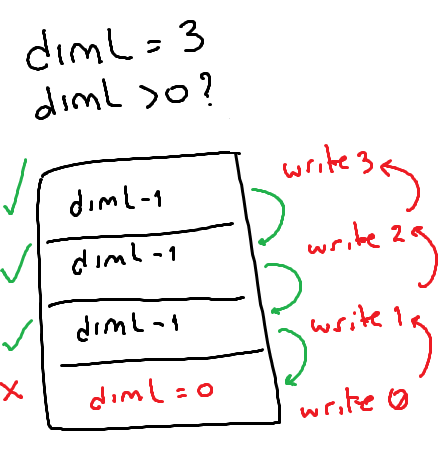
if (dimL > 0) then begin

imprimirVectorRecursivo(v, dimL-1);

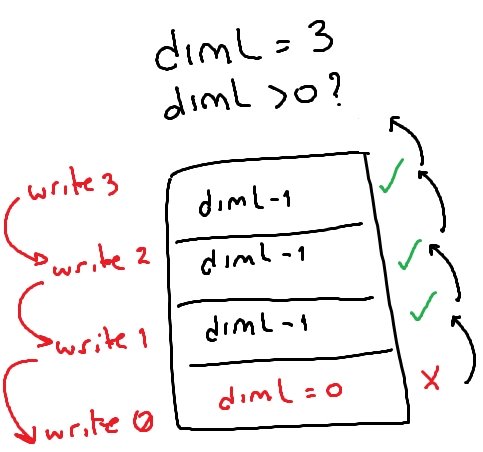
writeln(v[dimL]);

end;

end;



Si lo hago al reves, con writeln(v[dimL]) primero y la llamada a la recursión segundo, imprimo en orden inverso, y el programa “desarma” para arriba las iteraciones ya en memoria pero no ejecuta nada más:



**Árboles**

Son estructuras de datos jerárquicas, donde cada nodo puede tener hasta dos hijos (en el caso de árboles binarios como los que vemos). El nodo principal se denomina raíz, y los nodos que no tienen hijos se denominan hojas.

Es una estructura de datos **COMPUESTA, DINÁMICA, HOMOGÉNEA** y **NO LINEAL.**

Es jerárquica, representa una jerarquía: cada nodo (excepto el raíz) tiene su padre.

Es dinámica porque podemos agregar o eliminar nodos.

Es homogénea porque todos los elementos son del mismo tipo.

Un árbol vacío se representa con el valor nil.

Un nuevo dato siempre se inserta como una hoja.

Cómo declaro un árbol: como una lista.

Program arboles;

Type

arbol = ^nodo;

nodo = record

dato: tipoElemento;

HI: arbol;

HD: arbol;

end;

Var

a:arbol;

Begin

…

End.

Por ejemplo:

Program arbolGenealogico;

Type

arbol: ^nodo;

nodo = record

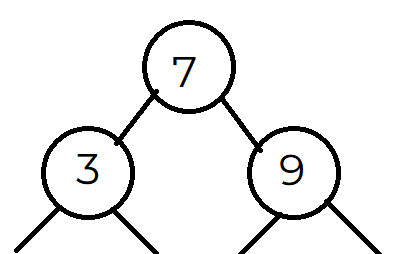
dato:string;

HI:arbol;

HD:arbol;

end;

Usamos los **árboles binaries de búsqueda**: significa que en casa caso en particular hay una condición que se respeta – los hijos mayores a cualquier nodo en que yo me pare estará a la derecha, y los más chicos a la izquierda. Por ejemplo:



¿Por qué usamos un árbol? Porque la búsqueda es mucho más eficiente. La búsqueda es de orden logarítmico – vamos descartando ramas del árbol que sabemos que no se van a recorrer. La lista, por el contrario, es lineal.

¿Cómo cargo valores a un árbol?

Program arbol;

Type

    arbol = ^nodo;

    nodo = record

        dato: **integer**;

        HI:arbol;

        HD:arbol;

    end;

Var

    a:arbol;

    num:**integer**;

Begin

    a:=nil *{inicializo el árbol}*

    read(num);

    while(num <> 50) do

        begin

            agregar(a,num);

            read(num);

        end;

End.

Y el procedimiento para agregarlo de manera recursiva:

**Procedure** agregar (var a:arbol;num:**integer**); *{el arbol x ref pq la primera vez modifico la raiz}*

begin

    if (a = nil) then begin *{si el árbol está vacío...}*

        new(a);

        a^.dato := num;

        a^.HI := nil; *{los hijos van en nil}*

        a^.HD := nil;

    else

        if(num <= a^.dato) then agregar(a^.HI,num) *{va a la izq}*

        else agregar(a^.HD,num) *{va a la derecha}*

    end;

end;

Agregar de manera iterativa sería mucho más complejo.

**¿Cómo recorro un árbol?**

Siempre comienzo el recorrido desde el nodo raíz. Si quiero por ejemplo modificar todos los nodos, sigo por su hijo izquierdo, y luego el hijo izquierdo de este – si no lo tiene, volvemos al padre y vamos al derecho; de ahí al izquierdo, etc.

¿Cuál es el caso base? Llegar al nodo = nil.

¿Cuántos llamados recursivos se hacen en cada nodo? Uno por hijo.

¿Cómo se implementa?

Program arboles;

Type

    arbol = ^nodo;

    nodo = record

        dato: **integer**;

        HI: arbol;

        HD: arbol;

    end;

Var

    a: arbol;

    num: **integer**;

Begin

    a:= nil; *{inicializo el árbol}*

    read(num); *{cargo el dato}*

    while (num <> 50) do begin

        agregar(a,num);

        read(num);

    end;

   enOrden(a);

End.

Hay otros algoritmos de recorrido:

- Preorden: se visita la raiz, luego el subarbol izquierdo y luego el derecho. Es útil para copiar un árbol porque la raiz siempre se procesa primero

- Postorden: se visita el árbol izquierdo, luego el derecho, luego la raiz. Es útil para eliminar el árbol porque los hijos se procesan antes que los padres.

**Procedure** preOrden(a:arbol);

begin

    if (a <> nil) then begin

        write(a^.dato); *{acción}*

        preOrden(a^.HI); *{sigo yendo a la izq mientras no sea nil}*

        preOrden(a^.HD); *{voy a la derecha sino}*

    end;

end;

**Procedure** postOrden(a:arbol);

begin

    if (a <> nil) then begin

        postOrden(a^.HI); *{sigo yendo a la izq mientras no sea nil}*

        postOrden(a^.HD); *{voy a la derecha sino}*

        write(a^.dato); *{acción}*

    end;

end;

¿Qué pasa si quiero buscar un valor en un árbol?

Como los ABB *siempre* *están ordenados,* podemos usar esto de manera eficiente para descartar ramas enteras – buscar por distinto un valor como si fuera una lista ordenada funciona pero no es eficiente y está mal.

Si estoy buscando un valor arranco por la raíz: si mi valor es más chico, descarto toda la rama derecha. Y así sucesivamente. Si llego a nil, el valor no existe, sino es porque lo encontré.

**Opción 1: devuelvo un boolean si encontré el valor.**

Programa arbolesEnteros;

Type

    arbol = ^nodo;

    nodo = record

        dato: **integer**;

        HI: arbol;

        HD: arbol;

    end;

**Function** buscar(a:arbol; x:**integer**):**boolean**;

begin

    if (a=nil) then buscar:=false; *{caso base}*

    else

        if (a^.dato = x) then buscar:= true; *{lo encontré?}*

        else

            if (x > a^.dato) then buscar:= buscar(a^.HD,x); *{busco a  la derecha}*

            else buscar := buscar(a^.HI, x) *{busco a la izquierda}*

end;

Var

a: arbol;

min: **integer**;

ok: **boolean**;

x: **integer**;

Begin

    cargarArbol(a);

    read(x);

    ok:=false;

    if(a <> nil) then

        ok:=buscar(a,x);

    write(ok);

End.

Puedo alternativamente devolver el nodo en lugar de un boolean haciendo el output de la función de tipo arbol, y devolviendo el nodo entero.

Programa arbolesEnteros;

Type

    arbol = ^nodo;

    nodo = record

        dato: **integer**;

        HI: arbol;

        HD: arbol;

    end;

**Function** buscarNodo(a:arbol;x:**integer**):arbol;

Begin

    if(a=nil) then buscarNodo:= nil;

    else

        if (a^.dato = x) then buscarNodo := a;

        else

            if (x > a^.dato) then

                buscarNodo := buscarNodo(a^.HD, x)

            else

                buscarNodo := buscarNodo(a^.HI, x)

end;

Var

a: arbol;

bus:arbol;

x:**integer**;

Begin

    cargarArbol(a);

    read(x);

    bus:=buscarNodo(a,x);

    if(bus <> nil) then write('Encontrado');

End.

Además podríamos querer hacer otras operaciones, como:

* Buscar valores dentro de un rango;
* Obtener el valor máximo de un ABB; {este siempre va a ser el valor más a la derecha}
* Obtener el valor mínimo de un ABB; {siempre el valor más a la izquierda}
* Contar la cantidad de nodos;
* Calcular el nivel de un ABB;
* Calcular la altura de un ABB.

**Máximos y Minimos**

¿Cómo encontramos el valor más pequeño en el árbol, o el nodo completo? Tengo que recorrer el árbol solo por la izquierda hasta encontrar el lugar donde HI^.HI = nil, es decir el valor “más a la izquierda”.

Programa arbolesEnteros;

Type

    arbol = ^nodo;

    nodo = record

        dato: **integer**;

        HI: arbol;

        HD: arbol;

    end;

Var

    a:arbol; min:arbol;

Begin

    cargarArboles(a);

    min := minimoNodo(a);

    if (min <> nil) then write (min^.dato);

End.

**Function** minimoNodo(a:arbol):arbol;

begin

    if (a = nil) then minimoNodo := nil

    else

        if (a^.HI = nil) then minimoNodo:=a;

        else minimonodo:=minimoNodo(a^.HI);

end;

El máximo se calcula de forma parecida, pero recorriendo por los hijos derechos.

**Búsqueda en un árbol**