Esquema principal

program ejercicioPractica;

const

dimF = ;

fin = ;

valorAlto= 999;

type

rango = 1..dimF;

cadena = String[];

registro = record

end;

lista = ^nodo; // ordenada por ...

nodo = record //SI HAY MAS DE UNA LISTA O UN ARBOL CAMBIAR NOMBRE

dato: DATO;

sig: lista;

end;

vector = array [rango] of DATO; // ordenada por ...

arbol = ^nodoArbol; /// ordenado por ...

nodoArbol = record

dato: DATO;

hi: arbol;

hd: arbol;

end;

{---------------Fin del type----------------}

procedure leerDato (var x: tipoDato);

begin

with “x” do begin

xxx := random ();

if( “xxx” <> fin) then begin

end;

end;

end;

{---------------Programa Principal----------------}

var

begin

randomize;

lalista:=nil;

end.

* **VECTORES -**

Agregar un elemento (es siempre al final)

procedure agregar (var v: vector; var dimL: integer; dato: tipoElem);

begin

if (dimL < DIMF) then begin {verifica si hay espacio en el vector}

dimL := dimL + 1; {incremento la dimL}

v[dimL] := dato; {guardo el elemento en el vector}

end;

end;

Insertar en una pos determinada

procedure insertar (var v: vector; var dimL: integer; pos: integer; dato: tipoElem);

var

i: integer;

begin

{verifico si hay espacio en el vector y la posición es válida}

if ( (dimL < DIMF) AND ((pos >= 1) AND (pos <= DIMF)) ) then begin

for i := dimL downto pos do

v [i + 1] := v [i]; {hace el corrimiento a la derecha}

v [pos] := dato; {guardo el elemento}

dimL := dimL + 1; {agrando la dimL}

end;

end;

Insertar Ordenado

function buscarPosicion (v: vector; dimL: integer; dato: tipoElem) : integer;

var

pos: integer;

begin

pos := 1;

while (pos <= dimL) AND (dato > v[pos]) do

pos := pos + 1;

buscarPosicion := pos;

end;

procedure insertar (VAR v: vector; VAR dimL: integer; pos: integer; dato: tipoElem);

var

i: integer;

begin

for i := dimL downto pos do

v[i + 1] := v[i]; {corrimiento}

v[pos] := dato; {inserto el elemento}

dimL := dimL + 1; {incremento dimL}

end;

procedure insertarOrdenado (var v: vector; var dimL: integer; var exito: boolean; dato: tipoElem);

var

pos: integer;

begin

exito := false;

if (dimL < DIMF) then begin {verifico que haya espacio}

pos := buscarPosicion (v, dimL, dato);

insertar (v, dimL, pos, dato);

exito := true;

end;

end;

Crear un vector de listas (insertando ordenado)

procedure crearVectorListas (var v: vector);

//inicializa las posiciones del vector en nil

procedure inicializarVectorListas (var v: vector);

var

i: rango;

begin

for i := 1 to DIMF do

v[i] := nil;

end;

// —--------------------------------------------------

var

i: integer;

begin

inicializarVectorListas (v); // inicializa punteros

for i := 1 to DIMF do

CrearListaOrdenada (v[i]);

end;

Crear un vector a partir de un arbol (y con un rango)

procedure cargarVector (var v: vector; var dimL: integer; a: arbol; dniA, dniB: integer; catBuscada: rango\_cat);

begin

if (a <> nil) then begin

if (a^.dato.dni <= dniA) then // rango inferior

cargarVector (v, dimL, a^.hd, dniA, dniB, catBuscada)

else begin

if (a^.dato.dni >= dniB) then // rango superior

cargarVector (v, dimL, a^.hi, dniA, dniB, catBuscada)

else begin // si llega hasta aca quiere decir que esta en rango

cargarVector (v, dimL, a^.hi, dniA, dniB, catBuscada);

if (a^.dato.categoria = catBuscada) then

agregarDato (v, dimL, a^.dato.dni, a^.dato.nroLegajo);

cargarVector (v, dimL, a^.hd, dniA, dniB, catBuscada);

end;

end;

end;

end;

Eliminar un elemento a partir de una posicion

procedure eliminar (VAR v: vector; VAR dimL: integer; pos: integer);

var

i: integer;

begin

if (pos >=1) AND (pos <= dimL) then begin {si la pos es valida entro}

for i := pos to dimL-1 do

v[i] := v[i + 1]; {hago el corrimiento}

dimL := dimL - 1;

end;

end;

Eliminar todas las ocurrencias de un vector

Procedure BorrarOcurrencias (var v: vector; var dimL:integer; elem:integer);

Procedure BorrarPosModif (var v: vector; var dimL: integer; pos: integer);

var

i: integer;

begin

for i:=pos+1 to dimL do

v[i-1] := v [i];

dimL := dimL-1;

end;

var

i: integer;

Begin

i:=1; {sirve para recorrer el vector}

while (i<=dimL) do begin

if(v[i] = elem)then {si el numero del vector es el que busco -> entonces lo borro}

BorrarPosModif(v,dimL,i) {borro el numero del vector}

else {si no borra el elemento -> avanzo}

i := i+1; {avanzo en el vector}

end;

end;

Busqueda (vector desordenado)

function buscar (v: vector; dimL: integer; elem: dato) : boolean; {Retorna si se encuentra o no}

var

pos: integer;

seEncuentra: boolean;

begin

seEncuentra := false;

pos := 1;

while ( (pos <= dimL) AND (NOT seEncuentra) ) do begin

if (v[pos] = elem) then

seEncuentra := true

else

pos := pos + 1;

end;

buscar := seEncuentra;

end;

Busqueda (vector ordenado): 2 métodos

* Mejorada

Function BuscoPosElemOrd (x: integer; v: vector; dimL: integer): integer; {Retorna la posición}

var

pos : integer;

begin

pos:=1;

while (pos <= dimL) and (x > v[pos]) do

pos:=pos+1;

if ( pos > dimL ) or (x < v [pos]) then

pos:=0;

BuscoPosElemOrd:= pos;

end;

* Dicotomica

procedure busquedaDicotomica (num: integer; v: vector; var inf: integer; var sup: integer; var ok: boolean);

var

central: integer;

begin

central := (inf + sup) div 2; // actualiza el medio

if ( num = v[central] ) then begin

writeln ('lo encontre');

ok := true

end

else begin

if (inf <= sup) then begin

if (num < v[central] ) then

sup := central - 1

else

inf := central + 1;

busquedaDicotomica (num, v, inf, sup, ok);

end

else ok := false;

end;

end;

Buscar el maximo (solucion recursiva)

procedure buscarMax (var max: integer; var v: vector; i: integer);

begin

if (i <= dimF) then begin

buscarMax (max, v, i+1);

if (v[i] > max) then

max := v[i];

end

else

max := -1;

end;

Sumar elementos (solucion recursiva)

procedure sumaElementos (var suma: integer; var v: vector; i: integer);

begin

if (i <= dimF) then begin

suma := suma + v[i];

sumaElementos (suma, v, i+1);

end;

end;

Ordenar por Selección

procedure ordenarSeleccion (var v: vector; dimL: integer);

var

i, j, min: integer; {uso min en lugar de "p" xq es mas explicativo}

item: tipoElem;

begin

for i := 1 to dimL-1 do begin

min := i; { necesito guardar la pos minima actual porque es el inicio }

for j := i+1 to dimL do{para buscar el elemento mínimo en cada pasada}

if (v[j].criterio < v[min].criterio) then {verifico si hay actualizar el min}

min := j;

{ intercambia v[i] con v[min], incluso si no es necesario }

item := v[min];{guardo el elemento mínimo en una variable auxiliar}

v[min] := v[i];{en la posición actual guardo el mínimo}

v[i] := item;

end;

end;

Ordenar por Inserción

procedure ordenInsercion (var v: vector; dimL: integer);

var

i, j: integer;

actual: tipoElem;

begin

for i := 2 to dimL do begin

actual := v[i];

j := i - 1;

while (j > 0) AND (v[j].criterio > actual.criterio) do begin

v[j+1] := v[j]; { hago el corrimiento }

j := j - 1; { decremento en uno a j para que compare con todos los valores anteriores a él }

end;

v[j + 1] := actual { inserto el dato en la posición que le corresponde }

end;

end;

Calcula promedio de un vector de manera recursiva

function Calcular\_promedio\_Vector(v:vector; i: integer): real;

begin

if(i <= DIM)then

calcular\_promedio:= v[i].legajo / DIM + calcular\_promedio(v,i+1);

end;

Cargar un vector de listas a partir de una lista anterior

procedure cargarVector (var v: vectorListas; l: lista);

begin

while (l <> nil) do begin

agregarAdelante (v[l^.dato.nroCriterio], l^.dato);

l := l^.sig;

end;

end;

* **LISTAS -**

Agregar Adelante // Queda al revés de cómo se ingresan los datos

procedure agregarAdelante (var pri: lista; dato: dato);

var

aux: lista;

begin

new (aux);

aux^.dato := dato;

aux^.sig := pri;

pri := aux;

end;

Agregar Atrás // Con puntero al último - Mantiene el orden en que se ingresan los datos

procedure agregarAtras (var pri, ult: lista; dato: dato); // mi "pri" tu "L"

var

aux: lista; // mi "aux" tu "nue"

begin

new(aux);

aux^.dato := dato;

aux^.sig := nil;

if (pri = nil) then // si esta vacia

pri := aux

else // si no esta vacia

ult^.sig := aux;

ult := aux;

end;

Insertar Ordenado

procedure insertarOrdenado (var pri: lista; dato: dato);

var

aux, ant, act: lista;

begin

new(aux);

aux^.dato := dato;

act := pri; // ubico act al inicio de la lista

while (act <> nil) AND (act^.dato.criterio < dato.criterio) do begin

ant := act;

act := act^.sig; // avanzo en la lista

end;

if (act = pri) then // si está vacía o hay que insertar al inicio

pri := aux

else // el nuevo nodo va en el medio o al final de la lista

ant^.sig := aux; // actualizo puntero para insertar nuevo nodo

aux^.sig := act; // inserto nuevo nodo

end;

Crear Lista: AgregarAdelante, AgregarAtras y AgregarOrdenado (para una lista de numeros enteros).

// genera valores aleatorios (entre 0 y 15) y los agrega adelante en la lista recibida por parámetro

procedure CrearListaAgregarAdelante (var l: lista);

var

num: integer;

begin

l := nil; // inicializa puntero de la lista

num := random (16); // toma numeros de 0 al 15

while (num <> 0) do begin

writeln('numero: ', num);

agregarAdelante (lista, num);

num := random (16);

end;

end;

// genera valores aleatorios (entre 0 y 15) y los agrega atras en la lista recibida por parámetro

procedure CrearListaAgregarAtras (var pri: lista);

var

num: integer;

ult: lista;

begin

l := nil; // inicializa puntero de la lista

num := random (16); // toma numeros de 0 al 15

while (num <> 0) do begin

writeln('numero: ', num);

agregarAtras (pri, ult, num);

num := random (16);

end;

end;

// genera valores aleatorios (entre 0 y 15) y los agrega de forma ordenada en la lista recibida por parámetro

procedure CrearListaOrdenada (var l: lista);

var

num: integer;

begin

l := nil; // inicializa puntero de la lista

num := random (16); // toma numeros de 0 al 15

while (num <> 0) do begin

writeln('numero: ', num);

insertarOrdenado (l, num);

num := random (16);

end;

end;

Búsqueda de un elemento (lista desordenada)

function buscar (pri: lista; x: tipoElem) : tipoDevolucion;

var

found: boolean;

begin

found := false;

while (pri <> nil) AND (not found) do begin

if (x = pri^.dato.criterio) then

found := true

else

pri := pri^.sig;

end;

buscar := found;

end;

Búsqueda de un elemento (lista ordenada)

function buscar (pri: lista; x: tipoElem) : tipoDevolucion;

var

found: boolean;

begin

found := false;

while (pri <> nil) AND (not found) do begin // mientras no llgue al final y no encuentre el elemento

if (x = pri^.dato.criterio) then // si lo encontre actualizo booleano

found := true;

else

pri := pri^.sig; // sino avanzo

end;

buscar := found;

end;

Búsqueda de un elemento (de forma recursiva)

function buscar (l: lista; x: integer) : boolean;

begin

if (l = nil) then

buscar := false

else

if (l^.dato = x) then

buscar := true

else

buscar := buscar (l^.sig, x);

end;

Eliminar un elemento (lista desordenada)

procedure borrar (var pri: lista; var ok: boolean; elemABorrar: tipoElem);

var

ant, act: lista;

begin

ok := false;

act := pri;

// avanzamos mientras no lleguemos al final y no encontremos el elemento a eliminar

// Si el elemento a borrar SEGURO EXISTE, no hace falta preguntar act <> nil

while (act <> nil) AND (act^.dato.criterio <> elemABorrar) do begin

ant := act;

act := act^.sig;

end; // si sale del while es porque ya lo encontró seguro

if (act <> nil) then begin // si la lista no esta vacia (si el elem seguro existe esta linea no va)

ok := true;

if (act = pri) then // si hay que eliminar el primer nodo

pri := act^.sig

else

ant^.sig := act^.sig; // si es en el medio o en el último nodo se reasigna el siguiente de ant al siguiente del que voy a eliminar para no perder nodos

dispose (act);

end;

end;

Eliminar todas las ocurrencias de un valor en una lista sin orden

procedure borrarOcurrencias (elem: tipoElem; var pri: lista);

var

ant, act: lista;

begin

ant := pri;

act := pri;

while (act <> nil) do begin

if (act^.dato = elem) then

borrarElemento (pri, ant, act)

else begin

ant := act;

act := act^.sig;

end;

end;

procedure borrarElemento (var pri, ant, act: lista); // act trae la pos sobre la que hay que eliminar

var

aux: lista;

begin

aux := act;

act := act^.sig;

if (aux = pri) then begin

pri := act;

ant := act;

end

else

ant^.sig := act;

dispose (aux);

end;

Buscar el elemento mínimo en una lista, de forma recursiva

// devuelve el minimo valor de una lista

function minimo (l: lista) : integer;

var

num\_min: integer;

begin

if (l <> nil) then begin // caso base: l = nil

num\_min:= minimo (l^.sig); // llama a la recursion

if (l^.dato > num\_min)then // actualiza minimo

minimo := l^.dato

else

minimo := num\_min; // si no es menor, mantiene el del backtracking

end

else

minimo := 999; // si llega al final o la lista esta vacia, inicializa el minimo

end;

Buscar el elemento máximo en una lista, de forma recursiva

// devuelve el máximo valor de una lista

function maximo (l: lista) : integer;

var

num\_max: integer;

begin

if (l <> nil) then begin // caso base: l = nil

num\_max := maximo (l^.sig); // llama a la recursion

if (l^.dato > num\_max)then // actualiza maximo

maximo := l^.dato

else

maximo := num\_max; // si no es menor, mantiene el del backtracking

end

else

maximo := -1; // si llega al final o la lista esta vacia, inicializa el maximo

end;

* **ÁRBOLES -**
* Program arboles;
* Type
* // Arbol de enteros
* arbol= ^nodoA;
* nodoA = Record
* dato: integer;
* HI: arbol;
* HD: arbol;
* End;
* // Lista de Arboles PARA IMPRIMIRPORNIVEL
* listaNivel = ^nodoN;
* nodoN = record
* info: arbol;
* sig: listaNivel;
* end;
* {------------------------------ FIN DEL TYPE ------------------------------}
* // Insertar - Inserta un dato en un arbol binario de busqueda. No guadra repeticiones.
* procedure Insertar (var a: arbol; dato: integer);
* begin
* if (a = nil) then begin
* new(a);
* a^.dato:= dato;
* a^.HI:= nil;
* a^.HD:= nil;
* end
* else
* if (a^.dato > dato) then
* Insertar (a^.HI, dato)
* else if (a^.dato < dato) then
* Insertar (a^.HD, dato);
* end;
* procedure CrearABB (var a: arbol);
* var
* num: integer;
* begin
* a := nil; // inicializa puntero del arbol (la raiz)
* writeln ('ingrese un numero entero');
* readln (num);
* while (num <> 0) do begin
* Insertar (a, num); // inserta el dato en el arbol
* writeln ('ingrese un numero entero');
* readln (num);
* end;
* {
* num := random (101); // toma numeros de 0 al 100
* while (num <> 0) do begin
* writeln('numero: ', num); // imprime el numero random
* Insertar (a, num); // inserta el dato en el arbol
* num := random (101); // lee otro numero
* end;
* }
* end;
* {-------------------MODULOS PARA IMPRIMIRPORNIVEL------------------------------}
* // CONTARELEMENTOS - Devuelve la cantidad de elementos de una lista l
* function ContarElementos (l: listaNivel): integer;
* var
* c: integer;
* begin
* c:= 0;
* While (l <> nil) do begin
* c:= c+1;
* l:= l^.sig;
* end;
* contarElementos := c;
* end;
* // AGREGARATRAS - Agrega un elemento atras en l
* Procedure AgregarAtras (var l, ult: listaNivel; a:arbol);
* var
* nue:listaNivel;
* begin
* new (nue);
* nue^.info := a;
* nue^.sig := nil;
* if (l = nil) then
* l := nue
* else
* ult^.sig := nue;
* ult := nue;
* end;
* // IMPRIMIRPORNIVEL - Muestra los datos del arbol "a" por niveles
* Procedure ImprimirPorNivel (a: arbol);
* var
* l, aux, ult: listaNivel;
* nivel, cant, i: integer;
* begin
* l := nil;
* if (a <> nil) then begin // si el arbol tiene elementos
* nivel := 0;
* agregarAtras (l, ult, a); // agrega en la lista "l" el arbol "a"
* while (l <> nil) do begin
* nivel := nivel + 1;
* cant := contarElementos(l);
* write ('Nivel ', nivel, ': ');
* for i := 1 to cant do begin // tantas repeticiones como elementos haya en la lista
* write (l^.info^.dato, ' - '); // imprime el dato en concreto
* if (l^.info^.HI <> nil) then agregarAtras (l, ult, l^.info^.HI); // si tiene hijo izquierdo lo agrega a la lista
* if (l^.info^.HD <> nil) then agregarAtras (l, ult, l^.info^.HD); // si tiene hijo derecho lo agrega a la lista
* aux := l; // usa un puntero auxiliar para poder eliminar el dato que ya se imprimio
* l := l^.sig; // avanza la lista para procesar el proximo elemento
* dispose (aux); // elimina el dato que ya se imprimio
* end;
* writeln;
* end;
* end;
* end;
* {-------------------FIN MODULOS PARA IMPRIMIRPORNIVEL-------------------------}
* // imprime los elementos de un arbol en orden ascendente (si fuese descendente solo hay que intercambiar las lineas 132 y 134)
* procedure enOrden (a: arbol);
* begin
* if (a <> nil) then begin
* enOrden (a^.hi);
* write (a^.dato, ' - ') ;
* enOrden (a^.hd);
* end;
* end;
* // imprime los elementos de un arbol devolviendo la raiz seguido por todos los hijos izquierdos debajo, luego se dirige hacia su nodo derecho y repite el proceso
* procedure preOrden (a: arbol);
* begin
* if (a <> nil) then begin
* write (a^.dato, ' - ');
* preOrden (a^.hi);
* preOrden (a^.hd);
* end;
* end;
* // imprime los elementos de un arbol devolviendo primero los hijos y después el padre.
* procedure posOrden (a: arbol);
* begin
* if (a <> nil) then begin
* posOrden (a^.hi);
* posOrden (a^.hd);
* write (a^.dato, ' - ');
* end;
* end;
* // busca un elemento en un arbol. Si se encuentra devuelve un puntero al nodo donde se encuentra dicho valor; en caso de no encontrarlo retorna nil.
* function Buscar (a: arbol; dato: integer) : arbol;
* begin
* if (a = nil) then
* Buscar := nil
* else
* if (dato = a^.dato) then
* Buscar := a // si se encontro el dato se guarda el puntero donde se encuentra
* else
* if (dato < a^.dato) then
* Buscar := Buscar (a^.HI, dato) // si el dato es mas chico que donde estoy parada busco en el lado izquierdo
* else
* Buscar := Buscar (a^.HD, dato); // si el dato es mas grande que donde estoy parada busco en el lado derecho
* end;
* // devuelve el valor minimo de un arbol (en caso de llegar vacio retorna -1)
* function VerMin (a: arbol) : integer;
* begin
* if (a = nil) then // si el arbol no tiene elementos
* VerMin := -1 -- porque lo pide la consigna
* else
* if (a^.HI = nil) then // el caso base es que el nodo de la izquerda sea nil, eso significa que estoy parada en el minimo
* // porque el elemento minimo en un ABB tiene que ser (x convencion) el que se encuentre mas a la izquierda
* VerMin := a^.dato // se guarda el dato y deja de llamar a la recursion
* else
* VerMin := VerMin (a^.HI);
* end;
* // devuelve el valor maximo de un arbol (en caso de llegar vacio retorna -1)
* function VerMax (a: arbol) : integer;
* begin
* if (a = nil) then // si el arbol no tiene elementos
* VerMax := -1
* else
* if (a^.HD = nil) then // el caso base es que el nodo de la derecha sea nil, eso significa que estoy parada en el maximo
* // porque el elemento maximo en un ABB tiene que ser (x convencion) el que se encuentre mas a la derecha
* VerMax := a^.dato
* else
* VerMax := VerMax (a^.HD);
* end;
* \*function buscarMax (a: arbol) : arbol;
* begin
* if (a = nil) then
* buscarMax := nil
* else
* if (a^.hd = nil) then
* buscarMax := a
* else
* buscarMax := buscarMax (a^.hd);
* end;
* {PROGRAMA PRINCIPAL}
* var
* {pun, } a: arbol;
* elemABuscar, min, max: integer;
* begin
* randomize;
* CrearABB (a); // crea un arboles binario de busqueda de valores enteros
* writeln;
* writeln ('ABB por nivel:');
* ImprimirPorNivel (a); // imprime el ABB por nivel
* writeln;
* writeln ('Imprime el arbol usando enOrden');
* enOrden (a);
* writeln;
* writeln ('Imprime el arbol usando preOrden');
* preOrden (a);
* writeln;
* writeln ('Imprime el arbol usando posOrden');
* posOrden (a);
* writeln;
* writeln;
* writeln ('Ingrese un numero entero para buscar');
* readln (elemABuscar);
* // pun := Buscar (a, elemABuscar); // guarda donde se encontro el elemento (o nil si no se encontro)
* if (Buscar (a, elemABuscar) <> nil) then
* writeln ('El elemento se encontro')
* else
* writeln ('El elemento no se hallaba en el ABB');
* writeln;
* writeln;
* min := VerMin (a);
* if (min <> -1) then
* writeln ('El elemento minimo del arbol es: ', min)
* else writeln ('El arbol estaba vacio');
* writeln;
* writeln;
* max := VerMax (a);
* if (max <> -1) then
* writeln ('El elemento maximo del arbol es: ', max)
* else writeln ('El arbol estaba vacio');
* end.

Type de árboles

type

arbol = ^nodo; //como Arbol es un puntero es un tipo de dato SIMPLE

nodo = record

dato: tipo; //esto hay que cambiarlo cuando lo usemos segun el tipo de dato

HI: arbol;

HD: arbol;

end;

Crear un árbol\*\*pupiii\*\*

procedure crearArbol (var a: arbol);

procedure leerTipoElem ();

begin

end;

procedure agregarNodo (var a: arbol; elem: tipoElem);

begin

if (a = nil) then begin// caso base: que el árbol sea nil (está vacío o llegué a una hoja)

new (a);  
 a^.dato := elem;

a^.hi := nil; // inicializa los hijos en nil

a^.hd := nil;

end

else begin

if (elem.criterio < a^.dato.criterio) then // veo dónde va el nuevo nodo (izq o der)

agregarNodo (a^.hi, elem)

else

agregarNodo (a^.hd, elem)// si el valor que ingreso es mayor o igual va a ir a la derecha

end;

end;

var

elem: tipoElem;

begin

// aca se tiene que leer el registro de tipoElemn

while (elem <> fin) do begin

agregarNodo (a, elem);

// aca se vuelve a leer el registro de tipoElem

end;

end;

Sumar elementos

function sumarElem (a: arbol) : integer;

begin

if ( a <> nil ) then

sumarElem := a^.dato + sumarElem (a^.hi) + sumarElem (a^.hd)

else  
 sumarElem := 0;

end;

Acumular cant ´para codigos pares

function suma(a:arbol);

begin

if(a <> nil) then

suma:=0

else

suma:=a^.cant+suma(a^.HI)+suma(a^.HD)

else

suma:= suma(a^.HI)+suma(a^.HD)

end;

Buscar un elemento

// Buscar - Busca un codigo de participante en un arbol y devuelve si lo encontro o no

function Buscar (a: arbol; dato: integer) : boolean;

begin

if (a = nil) then // si no hay mas elementos o no se encontro

Buscar := false

else

if (dato = a^.dato.codParticipante) then

Buscar := true // si se encontro el dato se pone la funcion en true y se deja de llamar a la recursion

else

if (dato < a^.dato.codParticipante) then

Buscar := Buscar (a^.hi, dato) // si el dato es mas chico que donde estoy parada busco en el lado izquierdo

else

Buscar := Buscar (a^.hd, dato); // si el dato es mas grande que donde estoy parada busco en el lado derecho

end;

procedure buscar (a:arbol; x: integer; var ok:boolean);

begin

if (a=nil)then

ok:=false

else

if (a^.dato=x)then

ok:=true

else begin

buscar(a^.HI,x,ok);

if (not ok) then

buscar(a^.HD,x,ok)

end;

end;

Buscar minimo

// devuelve el valor minimo de un arbol (en caso de llegar vacio retorna -1)

function VerMin (a: arbol) : integer;

begin

if (a = nil) then // si el arbol no tiene elementos

VerMin := -1 → porque lo pide la consigna

else

if (a^.HI = nil) then // el caso base es que el nodo de la izquerda sea nil, eso significa que estoy parada en el minimo

// porque el elemento minimo en un ABB tiene que ser (x convencion) el que se encuentre mas a la izquierda

VerMin := a^.dato // se guarda el dato y deja de llamar a la recursion

else

VerMin := VerMin (a^.HI);

end;

Buscar maximo

\*// devuelve el valor maximo de un arbol (en caso de llegar vacio retorna -1)

function buscarMax (a: arbol) : integer;

begin

if (a = nil) then // si el arbol no tiene elementos

buscarMax := -1

else

if (a^.HD = nil) then // el caso base es que el nodo de la derecha sea nil, eso significa que estoy parada en el maximo

// porque el elemento maximo en un ABB tiene que ser (x convencion) el que se encuentre mas a la derecha

buscarMax := a^.dato // se guarda el dato y deja de llamar a la recursion

else

buscarMax := buscarMax (a^.HD);

end;

// misma solucion pero en un proceso

procedure buscarMax (a: arbol; var max: integer);

begin

if (a = nil) then

max := -1

else begin

if (a^.hd = nil) then

max := a^.dato

else

buscarMax (a^.hd, max);

end;

end;

Imprimir entre valores en un Arbol

procedure imprimirEntreValores(a:arbol);

begin

if(a<>nil)then

if(a^.dato.legajo > valorInf) and (a^.dato.legajo < valorSup)then

write(a^.dato.nombre);

imprimirEntreValores(a^.HI)

imprimirEntreValores(a^.HD)

else

if(a^.dato.legajo < 2803)then

imprimirEntreValores(a^.HD)

else

if(a^.dato.legajo > 6982) then

imprimirEntreValores(a^.HI)

end;

Calcular cuantos elementos tiene una lista

function contarElementos (l: listaNivel) : integer;

var

c: integer;

begin

c := 0;

while (l <> nil) do begin

c := c +1;

l := l^.sig;

end;

contarElementos := c;

end;

Cuenta cantidad de veces que aparece un elemento en el arbol

VERSION 1:

procedure calcularCantidad (a : arbol; mark : String; var cont : integer);

begin

if (a <> nil) then begin

if (mark = a^.dato.marca) then

cont := cont + 1;

calcularCantidad (a^.hi, mark, cont);

calcularCantidad (a^.hd, mark, cont);

end;

end;

VERSION 2:

function cantElementos (a:arbol):integer;

begin

if (a = nil) then cantElementos := 0

else begin

if ((50000 > a^.dato.monto) and (a^.dato.monto>10000)) then

cantElementos := 1 + cantElementos(a^.HI) + cantElementos(a^.HD)

else

cantElementos := cantElementos(a^.HI) + cantElementos(a^.HD);

end;

end;

Recorrido acotado (en este ejemplo se usa para elementos que esten dentro de un rango)

// recorridoAcotado - Recorre el arbol teniendo en cuenta un limite inferior y uno superior

procedure recorridoAcotado (a: arbol; inf, sup: integer; var cant: integer);

begin

if (a <> nil) then

if (a^.dato.dni >= inf) then

if (a^.dato.dni <= sup) then begin

cant := cant + 1; // +1 si encontro un elemento en rango

recorridoAcotado (a^.hi, inf, sup, cant);

recorridoAcotado (a^.hd, inf, sup, cant);

end

else

recorridoAcotado (a^.hi, inf, sup, cant)

else

recorridoAcotado (a^.hd, inf, sup, cant);

end;

* **MERGE -**

Codigo base

Procedure merge (vListas: vector; var nuevo: listaNueva);

procedure agregarAtras (var pri, ult: lista; dato: ? );

var

aux: lista;

begin

new (aux);

aux^.dato := dato;

aux^.sig := nil;

if (pri = nil) then // si esta vacia

pri := aux

else // si no esta vacia

ult^.sig := aux;

ult := aux;

end;

Procedure minimo (var v: vector; var min: ? );

Var

i, pos: integer;

Begin

pos := -1;

min.criterio := 999;

for i := 1 to DIMF do

if (v[i] <> nil) and (v[i]^.dato.criterio <= min.criterio) then begin

min := v[i]^.dato; // actualiza min

pos := i; // guarda la posicion de la lista

end;

if (pos <> -1) then // si encontre un minimo, avanzo

v[pos] := v[pos]^.sig;

End;

Var

min: ?;

ult: lista;

Begin

nuevo := nil;

minimo (vListas, min);

while (min.criterio <> 999) do begin

agregarAtras (nuevo, ult, min);

minimo (vListas, min);

end;

End;

Merge Acumulador (lista)

procedure mergeAcumulador (var listaNueva: listaPorCodigo; vListas: vectorVentas);

procedure agregarAtras (var pri, ult: listaPorCodigo; dato: ventasPorCodigo);

var

aux: listaPorCodigo;

begin

new (aux);

aux^.dato := dato;

aux^.sig := nil;

if (pri = nil) then // si esta vacia

pri := aux

else // si no esta vacia

ult^.sig := aux;

ult := aux;

end;

procedure minimo (var v: vectorVentas; var min: venta);

var

i, pos: integer;

begin

pos := -1;

min.cod\_producto := 999;

for i := 1 to DIMF do

if (v[i] <> nil) and (v[i]^.dato.cod\_producto <= min.cod\_producto) then begin

min := v[i]^.dato; // actualiza min

pos := i; // guarda la posicion de la lista

end;

if (pos <> -1) then // si encuentra un minimo, avanza

v[pos] := v[pos]^.sig;

end;

var

min: venta;

ventasXCodigo: ventasPorCodigo;

ult: listaPorCodigo;

productoActual: integer;

begin

listaNueva := nil;

minimo (vListas, min); // se devuelve el primer minimo

while (min.cod\_producto <> 999) do begin

productoActual := min.cod\_producto; // corte de control

ventasXCodigo.total\_ventas := 0; // inicializa contador

while (min.cod\_producto = productoActual) do begin// mientras siga viniendo el mismo producto

ventasXCodigo.total\_ventas := ventasXCodigo.total\_ventas + min.cant\_ventas;

minimo (vListas, min);

end;

ventasXCodigo.cod\_producto := productoActual; // carga el otro campo del registro acumulador

agregarAtras (listaNueva, ult, ventasXCodigo); // agrega el nuevo registro CARGADO a la lista (merge acumulador)

end;

end;

Merge Acumulador (arbol)

procedure mergeAcumulador (var a: arbol; v: vectorListas);

procedure agregarNodo (var a: arbol; elem: infoCriterio);

begin

if (a = nil) then begin // caso base: que el árbol sea nil (está vacío o llegué a una hoja)

new (a);

a^.dato := elem;

a^.hi := nil; // inicializa los hijos en nil

a^.hd := nil;

end

else begin

if (elem.criterio < a^.dato.criterio) then // veo dónde va el nuevo nodo (izq o der)

agregarNodo (a^.hi, elem)

else

agregarNodo (a^.hd, elem) // si el valor que ingreso es mayor o igual va a ir a la derecha

end;

end;

procedure minimo (var v: vectorListas; var min: registroContenidoEnElVector);

var

i, pos: integer;

begin

pos := -1;

min.criterio := 999;

for i := 1 to DIMF do // busca el minimo en todas las posiciones (listas) del vector

if (v[i] <> nil) and (v[i]^.dato.criterio <= min.criterio) then begin

min := v[i]^.dato; // actualiza min

pos := i; // guarda la posicion de la lista

end;

if (pos <> -1) then // si encontre un minimo, avanzo

v[pos] := v[pos]^.sig;

end;

var

min: registroContenidoEnElVector;

criterioActual: integer;

begin

minimo (v, min); // se devuelve el primer minimo

while (min.criterio <> 999) do begin

criterioActual := min.criterio;

while (min.criterio <> 999) and (min.criterio = criterioActual) do begin// mientras siga viniendo el mismo dato

... {hace lo que pide la consigna (contador, maximo, etc}

minimo (vectorListas, min);

end;

info.criterio := criterioActual; // se completa el registro a guardar

agregarNodo (a, info);

end;

end;

Imprimir

vector lista arbol

procedure imprimir\_vector(v: vector);

var

i: integer;

begin

for i := 1 to DIM do

writeln(v[i].nombre, ' ', v[i].legajo, ' ');

readln; // toque enter para imprimir el siguiente

end;

procedure imprimirVectorListas (v: vector);

procedure imprimirLista (l: lista);

begin

if (l = nil) then writeln ('La lista esta vacia')

else begin

while (l <> nil) do begin

writeln (' lo que se imprima: ',l^.dato.campo,' - ');

writeln (' lo que se imprima: ',l^.dato.campo,' - ');

writeln;

l := l^.sig;

end;

end;

var

i: integer;

begin

for i := 1 to cant\_? do begin

writeln;

writeln;

writeln (' lo que se imprima: ', i );

imprimirLista (v[i]);

readln;

end;

end;

// imprime los valores contenidos en una lista en el orden en que se guardados

procedure imprimirListaEnOrden (l: lista); // DE FORMA RECURSIVA

begin

if (l <> nil) then begin

writeln('Nro: ', l^.dato);

imprimirListaEnOrden (l^.sig);

end;

end;

procedure imprimirListaInvertida (l: lista); // DE FORMA RECURSIVA

begin

if (l <> nil) then begin

imprimirListaInvertida (l^.sig);

write(l^.dato, ' - ');

end;

end;

\*procedure enOrden (a: arbol);

begin

if ( a <> nil ) then begin

enOrden (a^.hi);

write (a^.dato) ;

enOrden (a^.hd);

end;

end;

\*procedure imprimir (a: arbol);

begin

if (a <> nil) then begin

imprimir (a^.hd);

writeln (a^.dato);

imprimir (a^hi);

end;

end;

\*procedure posOrden (a: arbol);

begin

if (a <> nil) then begin

posOrden (a^.hi);

posOrden (a^.hd);

write (a^.dato);

end;

end;

\*procedure preOrden (a: arbol);

begin

if (a <> nil) then begin

write (a^.dato);

preOrden (a^.hi);

preOrden (a^.hd);

end;

end;

\*Procedure ImprimirPorNivel (a: arbol);

var

l, aux, ult: listaNivel;

nivel, cant, i: integer;

begin

l := nil;

if (a <> nil) then begin // si el arbol tiene elementos

nivel := 0;

agregarAtras (l, ult, a); // agrega en la lista "l" el arbol "a"

while (l <> nil) do begin

nivel := nivel + 1;

cant := contarElementos(l);

write ('Nivel ', nivel, ': ');

for i := 1 to cant do begin // tantas repeticiones como elementos haya en la lista

write (l^.info^.dato, ' - '); // imprime el dato en concreto

if (l^.info^.HI <> nil) then agregarAtras (l, ult, l^.info^.HI); // si tiene hijo izquierdo lo agrega a la lista

if (l^.info^.HD <> nil) then agregarAtras (l, ult, l^.info^.HD); // si tiene hijo derecho lo agrega a la lista

aux := l; // usa un puntero auxiliar para poder eliminar el dato que ya se imprimio

l := l^.sig; // avanza la lista para procesar el proximo elemento

dispose (aux); // elimina el dato que ya se imprimio

end;

writeln;

end;

end;

end;

procedure imprimirEntreValores(a:arbol;num1,num2:integer); //permite imprimir entre dos valores que doy

begin

if(a<>nil)then

if(a^.dato.legajo > 2803) and (a^.dato.legajo < 6982)then

write(a^.dato.nombre);

imprimirEntreValores(a^.HI)

imprimirEntreValores(a^.HD)

else

if(a^.dato.legajo < 2803)then

imprimirEntreValores(a^.HD)

else

if(a^.dato.legajo > 6982) then

imprimirEntreValores(a^.HI)

end;

Modulos Recursivos

Implementar un módulo recursivo que reciba la estructura generada en a) y retorne el monto de la compra de mayor valor para un código de cliente que se recibe como parámetro, o -1 si el código de cliente no existe.

procedure puntoB (var found: boolean; var montoMax: real; a: arbol; codigo: integer);

begin

if ((a <> nil) and (not found)) then begin

if (a^.dato.codCliente = codigo) then begin

found := true;

montoMax := a^.dato.montoMax;

end

else

if (a^.dato.codCliente < codigo) then

puntoB (found, montoMax, a^.hd, codigo)

else

puntoB (found, montoMax, a^.hi, codigo);

end;

if (a = nil) and (not found) then // si no hay mas nodos por procesar y no se encontro: no existe

montoMax := -1;

end;

{MISMO MODULO PERO COMO FUNCION}

Implementar un módulo recursivo que reciba la estructura generada en a) y retorne la cantidad

de ciudades cuya cantidad de dosis entregadas es menor a un valor que se recibe como parámetro.

procedure puntoB (var cantDosis: integer; a: arbol; cant: integer);

begin

if (a <> nil) then

if (a^.dato.totalDosis < cant) then begin

cantDosis := cantDosis + 1;

puntoB (cantDosis, a^.hd, cant); // llama recursivamente al hijo derecho osea:

end; // Primero va a procesar los mayores

puntoB (cantDosis, a^.hi, cant); // llama recursivamente al hijo izquierdo

end;

Implemente un módulo recursivo que devuelva el mínimo valor de la lista.

function minimo(l:lista; min:integer):integer; //viene inicializado en 999

begin

if(l = nil)then

minimo := min

else begin

if(l^.dato < min) then

min := l^.dato;

minimo := minimo(l^.sig,min);

end;

end;

Implemente un módulo recursivo que devuelva el máximo valor de la lista.

function buscarMax (l: lista; max: integer) : integer;

begin

if (l = nil) then // si esta vacia

buscarMax := max // la funcion toma el valor del maximo

else begin

if (l^.dato > max) then

max := l^.dato; //actualiza el maximo

buscarMax (l^.sig, max); // compara el maximo actualizado con el siguiente nodo

end;

end;

Implemente un módulo recursivo que devuelva verdadero si un valor determinado

se encuentra en la lista o falso en caso contrario.

// devuelve verdadero si un valor determinado se encuentra en la lista o falso en caso contrario

function buscarElem (l: lista; elemABuscar: integer) : boolean;

begin

if (l = nil) then // caso base: si hay elementos o llegamos al final de la lista

buscarElem := false

else // si todavia hay elementos en la lista

if (l^.dato = elemABuscar) then

buscarElem := true // si lo encontro se debe terminar la busqueda

else // si no se encuentra el elemento hay que seguir buscandolo

buscarElem := buscarElem (l^.sig, elemABuscar); // se invoca de nuevo la funcion con el siguiente elemento de la lista

end;

Dado un dni, modificar la obra social de dicho paciente a una recibida. Considere que el

paciente puede no existir.

procedure cambiarObraSocial (var exito: boolean; var a: arbol; dniABuscar: integer; nuevaObraSocial: rango);

begin

if (a <> nil) and (not exito) then

if (a^.dato.dni = dniABuscar) then begin

exito := true;

a^.dato.obraSocial := nuevaObraSocial; // cambia la obra social

end

else begin

if (a^.dato.dni < dniABuscar) then

cambiarObraSocial (exito, a^.hd, dniABuscar, nuevaObraSocial);

cambiarObraSocial (exito, a^.hi, dniABuscar, nuevaObraSocial);

end;

end;

Implementar un módulo recursivo que reciba la información generada en “a” y retorne el

promedio de los números de legajo.

function promedioLegajos (v: vector; dimL, i: integer): real;

begin

if (i <= dimL) then

promedioLegajos := (v[i].nroLegajo / diml) + promedioLegajos (v, diml, i+1);

end;

Random:

Para enteros

variable := random(x); actua como un readln(x);  
“x” va a tomar valores desde el 0 (incluido) hasta (x-1).

writeln(variable); para ver lo que se ingreso con el random.

Usar “randomize” en el pp para que los datos varien (se genera una semilla)

Ej: variable = random(9)+2010 // hace un random entre 2010 y 2018

Para hacer una especie de random para strings (lo que hace es convertir el valor del numero ingresado para a.dni en un string).

Ej: registo.nombre := concat ( 'Nombre - ', IntToStr (registro.dni));

Declarar abajo de program pero arriba de const:

uses

sysutils;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Otra forma:**procedure randomString (tamanio: integer; var palabra: string);

var

str, Result: String;

begin

str:='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';

Result:='';

repeat

Result := Result + str [ Random (Length (str) )+1 ];

until (Length (Result) = tamanio);

palabra := Result;

end;

{ ej: randomString(10,nombre); //en la variable nombre va a guardar un string de 10 letras random }

MERGE CHECHU

Procedure mergeAcumulador (todos:estantes; var Enuevo:lista);

Begin

Enuevo:= nil;

minimo (todos,minNombre, monto);

while (minNombre <> ‘ZZZ’) do

begin

actual:= minNombre;

montoTotal:=0;

while ((minNombre <> ‘ZZZ’) and (minNombre = actual) )do

begin

montoTotal:= montoTotal + monto;

minimo (todos,minNombre,monto);

end;

agregarAtras(Enuevo,actual,montoTotal);

end;

End;

Procedure minimo (var todos:estantes; var nomMin:string; var monto:real);

Var

indiceMin,i:integer;

Begin

nomMin:= ‘ZZZ’; indiceMin:= -1;

for i:= 1 to cantE do

if (todos[i] <> nil) then

if (todos[i]^.dato.nombre <= nomMin) then begin

indiceMin:= i;

nomMin:= todos[i]^.dato.nombre;

end;

if (indiceMin <> -1) then

begin

//nomMin:= todos[indiceMin]^.dato.nombre;

monto:= todos[indiceMin] ^.dato.monto;

todos[indiceMin]:= todos[indiceMin]^sig;

end;

End;

Busqueda dicotomica

Procedure buscarCliente (v: vector; pri, ult: integer; dato: integer) : integer; // dato = dni cliente

var

pos: integer;

begin

if (pri > ult) then // caso base: solo va a ser mayor cuando no se encuentre el dato

buscarCliente := -1

else begin

pos := (pri + ult) DIV 2; // actualizo el medio

if (v[pos].dni = dato) then // si se encontro

buscarCliente := pos

else if (vClientes[pos].dni > dato) then

buscarCliente := buscarCliente (v, pri, pos-1, dato)

else

buscarCliente := buscarCliente (v, pos+1, ult, dato);

end;

{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

busquedaDicotomica - Busca un codigo de identificacion de oficina y retorna si esta en el vector. }

Procedure busquedaDicotomica (v: vector; pri, ult: integer; dato: integer; var pos: rango);

begin

if (pri > ult) then // caso base: solo va a ser mayor cuando no se encuentre el dato

writeln('El codigo buscado no existe.')

else begin

pos := (pri + ult) DIV 2; // actualizo el medio

if (v[pos].codId = dato) then // si se encontro

writeln('Se encontro el elemento. El DNI del propietario es: ', v[pos].DNI\_propietario)

else begin // si no encontre lo que busco me fijo para donde me tengo que mover

if (v[pos].codId < dato) then

pri := pos + 1 // me corro para la derecha

else { (v[pos].campo > dato) }

ult := pos - 1; // me corro para la izquierda

busquedaDicotomica (v, pri, ult, dato, pos); // solo hace esto cuando no se encontro el dato que buscaba

end

end;

end;