**Clase 1 imperativo (Estructuras de Datos vistas: arreglos – listas, concepto de Ordenación, método de inserción)**

Un arreglo es una estructura de datos compuesta que permite acceder a cada componente por una variable índice, que da la posición de la componente dentro de la estructura de datos. La estructura arreglo se almacena en posiciones contiguas de memoria.

Características: homogénea – estática – indexada

Manejan: dimensión física: la cantidad máxima de elementos que se pueden almacenar – dimensión lógica: la cantidad de elementos que realmente están cargados en el arreglo.

Type

nombre= array [rango] of tipo;

Operaciones:

Cargar la estructura

Agregar un elemento

Insertar un elemento

Eliminar un elemento

Recorrer la estructura

Buscar un elemento

Ordenar la estructura

Una lista es una estructura de datos lineal compuesta por nodos. Cada nodo de la lista posee el dato que almacena la lista y la dirección del siguiente nodo. Toda lista puede recorrerse a partir de su primer elemento. Los elementos no necesariamente están en posiciones contiguas de memoria.

Para almacenar un nuevo elemento debe reservarse memoria dinámica y si se desea también permite eliminar un elemento por medio de las operaciones new y dispose respectivamente.

Características: homogénea – dinámica – lineal.

Type

lista= ^**nodo**;

nodo = record

dato:tipo;

sig: lista;

end;

Operaciones:

Crear una lista vacía

Agregar un elemento adelante

Agregar un elemento atrás

Insertar un elemento

Eliminar un elemento

Recorrer la estructura

Buscar un elemento

Un algoritmo de ordenación es un proceso, algoritmo o programa por el cual un conjunto de elementos puede ser ordenado.

Si tuviera un arreglo de un millón de elementos y quisiera buscar solo 1, si el arreglo no tiene ningún orden tendré que buscar en el arreglo empezando por el primer valor y continuando hasta que encuentro lo que quiero o se me termina el arreglo.

Si ese arreglo tiene un orden, y encuentro un valor mayor al que quiero buscar, no tendré que seguir buscando, puesto que ya no hay valores menores al que estoy buscando a la derecha del arreglo.

Muchas veces, antes de buscar información, es más útil primero ordenar la estructura, si es que no fue creada de manera ordenada, y una vez que se crea la estructura y se cargan los datos aplicarle un proceso de ordenación.

Existen diferentes métodos de ordenación: selección intercambio e inserción.

Algunos son más fáciles de implementar, otros requieren menos tiempo en ejecución, otros requieren menos memoria utilizada.

¿Por qué es importante ordenar una estructura? Para futuras búsquedas. Si yo voy a generar una estructura que solo voy a agregar o quiero eliminar un elemento de una posición no hay necesidad de ordenarlo. En general cuando uno genera una estructura una de las operaciones que mas se realizan sobre la estructura es la búsqueda, y por lo tanto es importante que la estructura esté ordenada para reducir los costos de las búsquedas.

El ordenamiento por inserción (insertion sort) es una manera muy natural de ordenar un conjunto de elementos almacenados en un arreglo. Inicialmente se considera un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos. Requiere O(n²) operaciones para ordenar una lista de n elementos.

**Repetir** desde i: = 2 hasta dimL

actual: = v[i];

j: = i-1;

**Mientras** (j > 0) y (v[j] > actual)

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

j: = j – 1;

v[j+1]: = actual;

En cada vuelta, en cada valor de i, lo que se hace es guardar en una variable lo que hay en esa posición, y controlar todas las anteriores buscando la posición donde quiero insertar esta variable que me guarde. Una vez que encuentro el valor, encuentro la posición, o se me termina el arreglo, lo inserto.

Si los datos están ordenados de menor a mayor el algoritmo solo hace comparaciones, por lo tanto es de orden (n).

Si los datos están ordenados de mayor a menor el algoritmo hace todas las comparaciones y todos los intercambios, por lo tanto es de orden (n^2) comparaciones.

**Clase 2 imperativo (recursión, arboles binarios de búsqueda)**

La recursión es una metodología para resolver problemas. Permite resolver un problema P por resolución de instancias más pequeñas P1, P2, … Pn del mismo problema. El problema Pi es de la misma naturaleza que el problema original, pero en algún sentido es más simple.

Características:

El problema es siempre el mismo, pero debe ir achicándose.

Siempre tienen al menos un caso base, en el cual el código a implementar no es recursivo (a veces no debe escribirse código en el caso base, pero siempre existe).

El caso base es el que determina el final de la solución recursiva y siempre lo se resolver.

**Imprimir lista recursivamente:**

Procedure imprimir (L:lista);

Begin

IF (L <> nil) do

begin

write (L^.dato);

L:= L^.sig;

imprimir (L);

end;

End;

Podemos tener procedimientos o funciones recursivas. Me doy cuenta que es una solución recursiva cuando encuentro que dentro de la implementación de un procedimiento se llama a si mismo.

En memoria, se carga el programa principal con sus variables, y cuando se llama a una solución recursiva, se carga en memoria con sus variables y empieza a ejecutarse. Al llamarse a si mismo, se vuelve a cargar en memoria otra instancia de la solución recursiva y comienza a ejecutarse. La primera sigue cargada. Al llamar de vuelta sucede lo mismo, se carga en memoria la solución recursiva pero como es el caso base termina su ejecución. Se libera la memoria y vuelve a quien lo llamo, la instancia anterior de la solución recursiva. Así la memoria se va liberando hasta llegar al llamado original que devuelve la ejecución al programa principal. Como vemos, una solución recursiva ocupa mas espacio en memoria que una solución iterativa. La ventaja que presentan es que a veces existen soluciones que son mucho mas fáciles y claras de escribir de manera recursiva, que de manera iterativa.

Si inicializo un parámetro dentro de un proceso recursivo ese parámetro se va a inicializar cada ves que se llame al proceso en las instancias recursivas.

Cuando tengo parámetros, si los tengo por referencia, cuando se vuelve de la instancia anterior esos valores vuelven modificados, y si los tengo por valor no modifican al valor que tenían en la instancia anterior.

**Ej potencia recursiva en procedimiento:**

Procedure potencia (num,n:integer; var pot:integer);

Begin

if (n = 0) then pot:= 1

else

begin

pot:= pot \* num;

potencia (num, (n-1), pot);

end;

End;

**Ej potencia recursiva en función:**

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

Begin

if (n = 0) then potencia:= 1

else

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

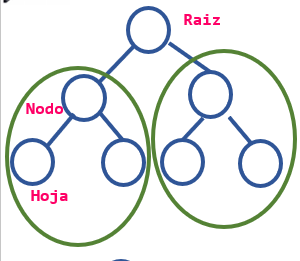
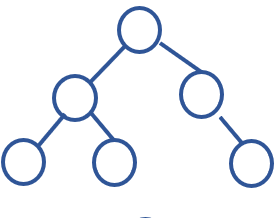
end;

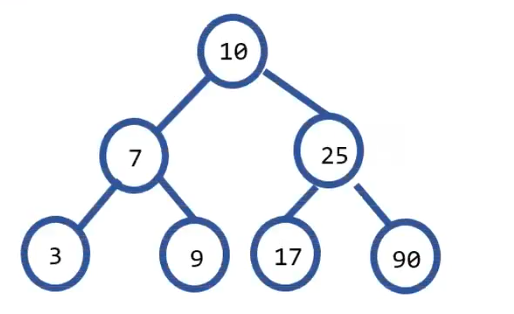
End;

**Arbol binario de busqueda.**

Un árbol binario de búsqueda es una estructura de datos jerárquica. Está formada por nodos, donde cada nodo tiene a lo sumo dos hijos y mantienen un orden.

Homogénea – Dinámica – No lineal, sino jerárquica, porque marca una jerarquía – Acceso secuencial: siempre arranco desde la raíz y de ahí bajo.

Los datos del árbol mantienen un criterio. En este caso cada hijo del lado izquierdo del árbol tiene un valor mas chico que su padre.

Este criterio de ordenación nos permite hacer búsquedas muy eficientes dentro de los árboles. Las búsquedas dentro de una estructura de tipo árbol son de orden logaritmo de n (l(n). La búsqueda dentro de ellos es muy eficiente, de los métodos que vimos, la mas eficiente.

**Programa arboles declaración;**

Type

arbol = ^nodo;

nodo = record

dato: tipo;

HI: arbol;

HD: arbol;

end;

**Creacion:**

Procedure crear (var A:árbol; num:integer);

Begin

if (A = nil) then

begin

new(A);

A^.dato:= num; A^.HI:= nil; A^.HD:= nil;

end

else

if (num < A^.dato) then crear(A^HI,num)

else crear(A^.HD,num)

End;

**Imprimir un arbol**

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

**begin**

**if** ( a<> nil ) **then begin**

write (a^.dato);

preOrden (a^.HI);

preOrden (a^.HD);

**end;**

**end;**

**Imprimir arbol (lo ultimo que se imprime es la raiz)**

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

**begin**

**if** ( a<> nil ) **then begin**

posOrden (a^.HI);

posOrden (a^.HD);

write (a^.dato);

**end;**

**end;**

**Imprimir arbol (los datos se imprimen ordenados)**

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

**begin**

**if** ( a<> nil ) **then begin**

enOrden (a^.HI);

write (a^.dato);

enOrden (a^.HD);

**end;**

**end;**

Como busco en un árbol

Esta operación retorna un puntero al nodo en el árbol A que tiene valor x o Nil si no existe. Se presentan una solución recursiva y una solución iterativa. La versión iterativa de la operación Buscar devuelve un valor booleano que indica si el dato se encuentra o no en el árbol.

**Las funciones pueden retornar punteros porque es un tipo de dato simple.**

Function Buscar (a:arbol; x:elemento): arbol;

begin

if (a=nil) then Buscar:=nil

else if (x= a^.dato) then Buscar:=a

else

if (x < a^.dato) then Buscar:=Buscar(a^.hi ,x)

else Buscar:=Buscar(a^.hd ,x)

end;

La ventaja con respecto a la búsqueda en vectores y listas es que voy descartando ramas en cada una de las búsquedas

**Clase 3**

Merge, merge de listas de arreglos, merge acumuladores.

La operación de merge consiste en generar una nueva estructura de datos (arreglos, listas) ordenada a partir de la mezcla de dos o mas estructuras de datos previamente ordenadas. Las estructuras que se combinan guardan el mismo orden lógico interno (por ejemplo datos ordenados alfabéticamente).

Es decir, que a partir de dos o mas estructuras que están ordenadas generar una nueva estructura que también va a estar ordenada por el mismo criterio y va a combinar los elementos de las dos estructuras previamente ordenadas.

Primero deberé buscar, en las estructuras que me dan, el valor mínimo, y luego agregare ese mínimo a mi nueva estructura, es decir, necesitare 2 procesos diferentes.

Procedure merge (E1,E2:lista; var Enuevo:lista);

Begin

Enuevo:= nil;

minimo (E1,E2,min);

while (min <> ‘ZZZ’) do

begin

agregarAtras (Enuevo,min);

minimo (E1,E2,min);

end;

End;

Procedure minimo(var E1,E2:lista; var min:string); paso los estantes por referencia porque debo avanzar en la estructura para que la próxima comparación no sea la misma.

Begin

min := ‘ZZZ’;

if (e1 <> nil) and (e2 <> nil)then

if (e1^.dato <= e2 ^.dato ) then

begin

min:= e1^.dato;

e1:= e1 ^.sig;

end;

else begin

min:= e2 ^.dato;

e2:= e2 ^.sig;

end

else (e1 <> nil) and (e2 = nil) then begin

min:= e1^.dato;

e1:= e1 ^.sig;

end

else (e1 = nil) and (e2 <> nil) then begin

min:= e2 ^.dato;

e2:= e2 ^.sig;

end;

end;

Estos procedimientos funcionan solo cuando tengo 2 punteros, a continuación tendré un arreglo que almacena los punteros iniciales de mis estructuras.

Merge acumulador

Aquí el procedimiento merge será el que cambie, teniendo un corte de control para verificar que los datos sean iguales y acumular.

procedure mergeConCorte (var v:vector; l:lista2);

var

e,actual:entradas;

total:integer;

begin

l:=nil;

minimo(v,e);

while (e.cod<>9999) do

begin

actual:=e;

total:=0;

while (e.cod<>9999) and (e.cod = actual.cod) do

begin

total:= total+1;

minimo(v,e);

end;

agregarAdelante(l,actual,total);

end;

end;