



FACULTAD DE CIENCIAS EN FÍSICA Y MATEMÁTICAS

UNIVERSIDAD AUTÓNOMA DE CHIAPAS

APUNTES DEL CURSO DE

PROGRAMACIÓN II

Agosto – Diciembre 2019

DR. ROBERTO ARCEO REYES

LICENCIATURA EN FÍSICA

Licenciatura	Licenciatura en Física	Modalidad	Presencial
Nombre de la unidad de competencia	Programación II	Horas semestrales	Créditos
		DT = 2 DP = 2 I = 2.5	6
Nombre de la Academia	Academia de Física	Semestre	Tercero
Perfil docente	Licenciatura en Física o Matemáticas, o bien una ingeniería afín. Desable con estudios de posgrado (maestría), preferentemente se necesita tener conocimiento de la programación de alto nivel como el Lenguaje C, Fortran, entre otros.		
Presentación	La unidad de competencia permite al estudiante el estudio de los algoritmos y estructuras de datos básicos que se utilizan en la solución de problemas que surgen en el desarrollo de aplicaciones de cómputo y capacitarlo para entender e implementar un algoritmo, en cualquier lenguaje de alto nivel, explicar su comportamiento y compararlo con otros algoritmos que resuelvan el mismo problema; asimismo, proporcionarle las herramientas para que pueda generar aplicaciones de computo no triviales.		
Proyecto integrador	Resolución de problemas, personalmente o en grupo.		
Subcompetencia 1	INTRODUCCIÓN		
Conocimientos	- Tipos de datos estructurados: arreglos, registros, uniones y apuntadores. - Definición de tipos de datos. - Asignación de memoria dinámica.		
Habilidades	Analizar los datos estructurados y el uso de la memoria dinámica.		
Subcompetencia 2	ENTRADAS Y SALIDAS POR ARCHIVOS		
Conocimientos	- Flujos. Puntero FILE. - Apertura de un archivo. Funciones de entrada/salida para archivos. - Archivos binarios en C. Funciones para acceso aleatorio.		
Habilidades	Utilizar las características típicas de entrada/salida para archivos en C, así como las funciones de acceso más utilizadas.		
Subcompetencia 3	ESTRUCTURA DE DATOS Y ALGORITMOS		
Conocimientos	- Listas: simplemente enlazadas, doblemente enlazadas y Circulares. - Pilas y colas. - Árboles: binarios y n-arios. Recorrido. - Grafos: Representación. Algoritmos de teoría de grafos.		
Habilidades	Implementar las estructuras de datos básicos y las funciones que las manipulen. Recursividad.		
Subcompetencia 4	MÉTODOS DE ORDENAMIENTO		
Conocimientos	- Métodos de búsqueda: lineal, binaria y búsqueda Hash. - Métodos de ordenamiento: algoritmo de la burbuja, ordenación por selección, ordenación por inserción, ordenación Shell, ordenación rápida (quicksort).		

LICENCIATURA EN FÍSICA

Habilidades	Implementar los distintos algoritmos de ordenamiento.
Subcompetencia 5	APLICACIONES
Conocimientos	- Comprensión de datos: RLE (run length encoding) y Lempel-ziv-walsh. - Graficación: Algoritmos de Bresenham, líneas rectas, Círculos.
Habilidades	Aplicar los algoritmos y estructuras de datos estudiados a problemas reales.
Actitudes y valores	Reflexión, responsabilidad, disciplina, integridad, ingenio, colaboración y trabajos en equipo.
Actividades de aprendizaje	- Realizar lectura de textos pertinentes a la temática a abordar: revisión de material bibliográfico y de fuentes electrónicas. - Elaborar mapas conceptuales para la organización de la información. - Resolución de problemas en clase e independientes.
Recursos materiales y didácticos	- Recursos bibliográficos. - Recursos multimedia: videos, diapositivas, entre otros. - Software especializado.
Criterios de evaluación	La evaluación de los aprendizajes se realizará a través de evidencias concretas de conocimiento, proceso y productos tales como exámenes, tareas, exposiciones, entre otros. Se desarrollará de forma continua durante el proceso de enseñanza-aprendizaje a través de los siguientes momentos: Evaluación diagnóstica: Recupera los conocimientos previos y expectativas de los estudiantes respecto al tema y facilita la incorporación de nuevos aprendizajes. Evaluación formativa: Permite valorar integralmente el desempeño del estudiante durante el desarrollo de las actividades de la materia. Evaluación sumativa: Considera la integración de todas las actividades desarrolladas por el estudiante y permite la asignación de valores para la acreditación de la materia.
Referencias	- Tucker, A. B., et al. (1995). <i>Fundamentos de Informática y su versión en inglés: Fundamentals of Computing I: Logic, Problem-solving, Programs and Computers</i>. McGraw-Hill Inc. - Gottfried, B.S. (2006). <i>Programación en C</i>. McGraw-Hill Interamericana. - Jones, B.L. (2002). <i>Sams Teach Yourself C in 21 Days</i>. Sams Publishing. - Aguilar, L.J., Martínez, I.Z. (2005). <i>Programación en C, Metodología, algoritmos y estructura de datos</i>. España: 2ª Ed. McGraw-Hill/Interamericana de España. - Grogono, P. (1992). <i>Programación en Pascal</i>. SITESA. - Wirth, N. (1992). <i>Algoritmos y estructuras de datos</i>. Prentice Hall. - Stroustrup, B. (2013). <i>The C++ Programming Language</i>. Addison-Wesley Professional, 4ta Ed.

OBJETIVO:

Al finalizar la unidad el estudiante conocerá tipos de datos estructurados y el uso de la memoria dinámica.

PROPOSITO:

El propósito de la asignatura es enseñar al estudiante algunos algoritmos y estructuras de datos básicas que se utilizan en la solución de problemas que surgen en el desarrollo de aplicaciones de cómputo y capacitarlo para entender e implementar un algoritmo, en cualquier lenguaje de alto nivel, explicar su comportamiento y compararlo con otros algoritmos que resuelven el mismo problema; asimismo proporcionarle las herramientas para que pueda generar aplicaciones de cómputo no triviales.

INDICE

1. Entradas y salidas por archivos	4
2. Funciones de entrada/salida para archivo	5
2.1 Funciones putc () y fputc ()	5
2.2 Funciones getc () y fgetc ()	5
2.3 Funciones fputs () y fgets ()	5
2.4 Funciones fprintf () y fscanf ()	6
3. Construcción de una lista.....	8
4. Pilas y Colas.....	12
5. Arboles.....	17
6. Bibliografía.....	20

Entrar a Linux {\home\estudiantes\...}
Abrir una terminal

pwd desplegar la ubicación donde nos encontramos *medir bposada*
ls visualiza la información donde nos encontramos
cd entra a la carpeta bposada *cd bposada*
cd...
cp copia información a otra ubicación
mv mueve el archivo de ubicación
rm elimina el archivo seleccionado
rmdir elimina la carpeta *rmdir bposada*
rm nombre
rm* elimina todos los archivos

Editores [pico, emacs,...]

- pico ejemplo1.c
- xterm & abre otra terminal en la misma ubicación
- pico ejemplo1.c
- Emacs ejemplo1.c
- gedit ejemplo1.c

Entradas y salidas por archivos

Modos	Significado
"r"	Abre para lectura
"w"	Abre para crear nuevo archivo (si ya existe se pierden sus datos)
"a"	Abre para añadir al final
"r+"	Abre archivo ya existente para modificar (leer/escribir)
"w+"	Crea un archivo para escribir/leer (si ya existe se pierden los datos)
"a+"	Abre el archivo para modificar (escribir/leer) al final Si no existe es como w+.

Abrir un archivo de texto y después cerrarlo.

Ejemplo1.c

```
#include<stdio.h>
FILE*pf1;
pf1=fopen ("datos.dat", "r");
fclose (pf1);
```

Especificar la ruta donde nos encontramos (es decir el archive "datos.dat")

```
pf1=fopen ("\\home\\estudiantes\\anucamendi\\datos.dat", "r");
```

FILE introducción para archivo
fclose cierra el archivo
fopen abre un archivo

Funciones de entrada/salida para archivos

Una vez abierto un archivo para escribir datos hay que grabar los datos en el archivo. La biblioteca C proporciona diversas funciones para escribir datos en el archivo a través del puntero a FILE asociado.

Las funciones de entrada y de salida de archivos tienen mucho parecido con las funciones utilizadas para entrada y salida para los flujos Stein (teclado) y stdout (pantalla): printf (), scanf (), getchar (), putchar (), gets () y puts ().

Todos tienen una versión para archivos que empiezan por la letra f, así se tiene fprintf (), fscanf (), fputs (), fgets (); (las funciones específicas de archivo empiezan por f). Con estas funciones se escribe o lee cualquier dato del programa en un archivo de texto.

Funciones putc () y fputc ()

Ambas funciones son idénticas; putc () está definida como macro. Escriben un carácter en el archivo asociado con el puntero a FILE. Devuelven al carácter escrito, o bien EOF si no puede ser escrito. El formato de llamada:

putc (c, puntero_archivo)
fputc (c, puntero_archivo)

siendo c el carácter a escribir.

Funciones getc () y fgetc ()

Estas dos funciones son iguales, mismo formato y misma funcionalidad; pueden considerarse que son recíprocas de putc () y fputc (), getc () y fgetc (), leen un carácter (el siguiente carácter) del archivo asociado al puntero a FILE y devuelven el carácter leído o EOF si es fin de archivo (o si ha habido error). El formato de llamada es:

getc (puntero_archivo);
fgetc (puntero_archivo);

Funciones fputs () y fgets ()

Estas funciones escriben/leen una cadena de caracteres en el archivo asociado. La función fputs () escribe una cadena de caracteres. La función devuelve EOF si no ha podido escribir la cadena, un valor no negativo si la escritura es correcta; el formato de llamada es:

fputs (cadena, puntero_archivo);

La función **fgets ()** lee una cadena de caracteres del archivo. Termina la captación de la cadena cuando lee el carácter de fin de línea, o bien cuando ha leído n-1 caracteres, siendo n un argumento entero de la función. La función devuelve un puntero a la cadena devuelta, o NULL si ha habido un error. El formato de llamada es:

fgets (cadena, n, puntero_archivo);

Ejemplo: lectura de un máximo de 80 caracteres de un archivo.

```
#define T81
char cad [T];
FILE *f;
fgets (cad, T, f);
```

Funciones fprintf () y fscanf ()

Las Funciones printf () y scanf () permiten escribir o leer variables, cualquier tipo de dato estándar a los códigos de formato (%d, %f,...) indican a c la transformación que debe de realizar con la secuencia de caracteres (conversión a entero...). La misma funcionalidad tienen fprintf () y fscanf () con los prefijos (archivos asociados) a que se aplican. Estas dos funciones tienen como primer argumento el puntero a FILE asociado al archivo de texto.

Datos. dat

→20 →30
→7 →8
Juana

Empieza a leer de esta manera

```
→ → →
→ →
```

Ejemplo 2. c

```
#include <stdio.h>
main ( )
{
FILE *pf1, *pf2;
int a, b, c, d;
pf1=fopen ("datos. dat", "r");
pf2=fopen ("salida. out", "w");

fscanf (pf1, "%d %d %d %d", &a, &b, &c, &d);
fprintf (pf2, "%d %d %d %d", a, b, c, d);
fclose (pf1);
fclose (pf2);

printf ("los datos leídos son: %d %d %d %d", a, b, c, d);
return 0;
}
```

Ejemplo 3. c

```
#include <stdio.h>
#include <proccoss.h>
#include <conio.h>

Void clear_kb(void);
Void main (void)
```

```

{
FILE *fp;
Long char count [65535];
Char filename [20];
Char file1 [20];
Do
{
printf ("\n dame el archivo de entrada (para salir teclea Ø):");
gets (file1);
if ((fp=fopen (file1, "r"))==NULL)
{ fprintf (stderr, "Error abriendo archive");
Exit (2);
}

fscanf (fp, "%ls", &count);
Clear_kb ( );
Printf ("Dame el archive de salida:");
gets (filename);
if ((fp=fopen (filename, "a+"))==NULL)
{
fprintf (stderr, "\n Error abriendo archivo.");
exit (1);
}

{
fprintf (fp, "%ls", count);
fprintf (stdout, "ls", count);
}

{ while (file1 !=count);
fclose (fp);
return;
}

}

Void clear_kb (void)
{
char junk [80];
gets (junk);
}

```

El operador (→) de selección de un miembro.

Si p es un puntero a una estructura y m es un miembro de la estructura, entonces p→m accede al miembro m de la estructura apuntada por p.

El simbolo "→" se considera como un operador simple. Se denomina operador de selección de miembro o también operador de selección de componente.

$p \rightarrow m$ significa lo mismo que $(*p) = m$
utilizando el operador de selección $\rightarrow$ se puede imprimir los datos del primer
nodo de la lista.
`printf ("%lf", ptr_cabeza->dato);`

Construcción de una lista

Un algoritmo para la creación de una lista enlazada entraña, los siguientes
pasos:

Paso 1: declarar el tipo de dato y el puntero de cabeza ó primero.

Paso 2: asignar memoria para un elemento del tipo definido anteriormente
utilizando alguna de las funciones de asignación de memoria (`malloc`, `calloc`,
`realloc`) y un cast para la conversión de `void*` al tipo puntero a nodo; la
dirección del nuevo elemento es `pte_nuevo`.

Paso 3: crear interactivamente el primer elemento (cabeza) y los elementos
sucesivos de una lista enlazada simplemente.

Paso 4: repetir hasta que no haya más entrada para el elemento.

Ejemplo: crear una lista enlazada de elementos que almacenen datos de tipo
entero.

Un elemento de la lista se puede definir con la ayuda de la estructura siguiente:

Struct Elemento

```
{  
int dato;  
struct Elemento *siguiente;  
}  
typedef struct Elemento Nodo;
```

En la estructura Elemento hay dos miembros, `dato` que contiene el valor del
elemento de la lista y `siguiente` que es un puntero al siguiente nodo. El
siguiente paso para construir la lista es declarar la variable `primero` que
apuntará al primer elemento de la lista:

`Nodo *primero=NULL`

El puntero `primero` (se llama cabeza) se ha inicializado a un valor nulo, la lista
está vacía.

Crear un elemento de la lista (reservar memoria) y asignar la dirección de la
memoria reservada al puntero `primero`:

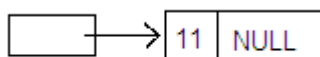
`Primero= (Nodo*) malloc (size of (Nodo));`

Si `sizeof`- tamaño de cada nodo de la lista.

`malloc`- devuelve un puntero genérico (`void*`), se convierte a `Nodo*`

Asignar un valor al campo `dato`:

`primero->dato=11;`
`primero->siguiente=NULL;`



primero

El puntero primero a punta al nuevo elemeno, 11> el nuevo valor es nulo por no haber un nodo siguiente. La operación de crear un nodo se puede hacer en una función a la que se pasa el valor del campo dato y del campo siguiente. La función devuelve un puntero al nodo. Creado:

Nodo* Crear nodo (int x, Nodo* enlace)

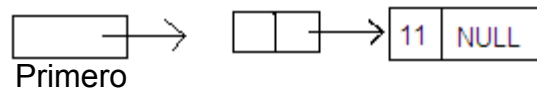
```
{Nodo *p;  
p=(Nodo*) malloc (size of (Nodo));  
p → dato=x;  
p → siguiente=enlace;  
return p;  
}
```

La llamada a la funcion Crear nodo () para crear el primer nodo de la lista:

Primero= Crear nodo (11, Nodo);

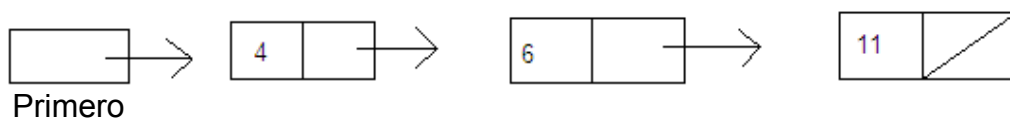
Si a hora se desea añadir un nuevo elemento con un valor (decir 6) y situarlo en el priemr lugar de la lista se escribe:

Primero= crear nodo (6, primero);



Por ultimo para obtener una lista compuesta de 4, 6, 11 se habar de ejecutar:

Primero =creando (4, primero);



INSERTAR UN ELEMENTO EN UNA LISTA

Varía con la posición en que se desea insertar. La operación de insertar puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento ultimo).
- Antes de un elemento especificado.
- Después de un elemento especificado.

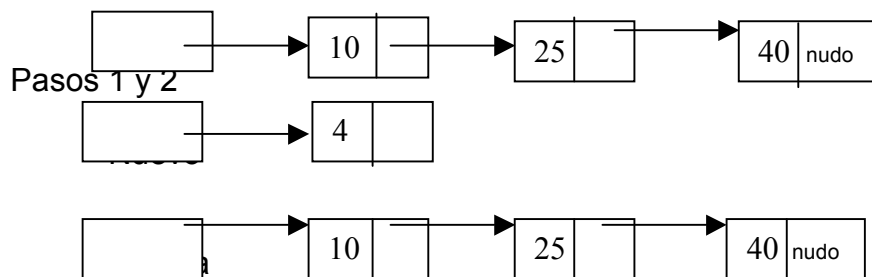
Insertar un nuevo elemento en la cabeza de una lista normalmente se insertan nuevos datos al final de una estructura de datos, es más fácil y más eficiente insertar un elemento nuevo en la cabeza de la lista.

El proceso de inserción se puede resumir en este algoritmo:

1. Asignar un nuevo nodo apuntado por nuevo que es una variable puntero local que apunta al nuevo nodo que se va insertar en la lista.
2. Situar el nuevo elemento en el campo dato del nuevo nodo.
3. Hacer que el campo enlace siguiente del nuevo nodo apunte a la cabeza (primer nodo) de la lista original.
4. Hacer que la cabeza (puntero cabeza) apunte al nuevo nodo que se ha creado.

Ejemplo:

Una lista enlazada contiene 3 elementos 10, 25 y 40; insertar un nuevo elemento "4" en la cabeza de la lista.



Código C

```

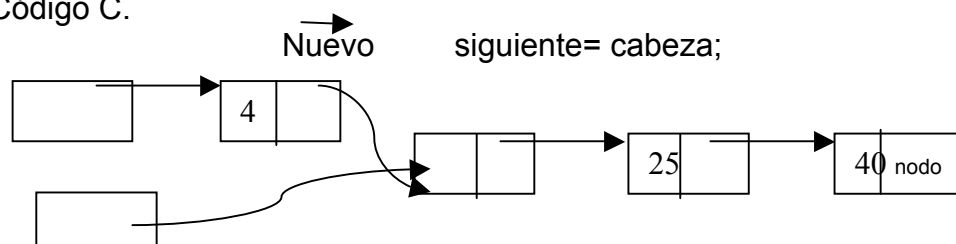
typedef int Item;
typedef struct tipo-nodo;
{
    Item dato;
    struct tipo-nodo* siguiente;
}
Nodo; /* Declaración del tipo Nodo */
Nodo* nuevo;
nuevo= (Nodo*) malloc (sizeof (Nodo)); /* se asigna un nuevo nodo */
nuevo-> dato= entrada;

```

Paso 3:

El campo enlace (siguiente) del nuevo nodo apunta a la cabeza actual de la lista.

Código C.

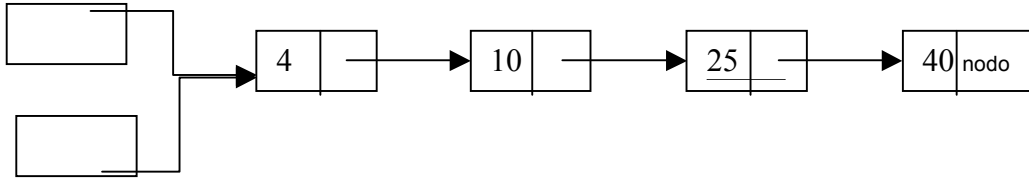


Paso 4

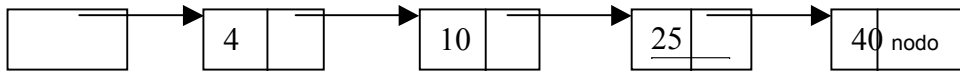
Se cambia el puntero de cabeza para apuntar al nuevo nodo creado, es decir, el puntero de cabeza apunta hacia el mismo sitio que apunte nuevo.

Código C.

cabeza= nuevo



En este momento la función de insertar un elemento en la lista termina su ejecución, la variable local nuevo desaparece y solo permanece el puntero de cabeza que apunta a la nueva lista enlazada.



El código fuente de la función.

Insertar cabeza lista:

Void Insertar CabezaLista (Nodo** cabeza, Item entrada)

```
{
    Nodo* nuevo;
    nuevo->dato= entrada; /* Pone elemento en nuevo */
    nuevo->siguiente= *cabeza; /* Enlaza nuevo nodo al frente de la lista */
    *cabeza= nuevo; /* Mueve puntero cabeza y apunta al nuevo nodo */
}
```

Ejercicio:

Crear una lista de dos números aleatorios e insertar los nuevos nodos por la cabeza de la lista. Una vez creada la lista, se han de recorrer los nodos para mostrar los números pares. La función InsertarCabeza Lista () añade un nodo a la lista, siempre como nodo cabeza. El primer argumento es un puntero a puntero porque tiene que modificar la variable cabeza, que es a su vez un puntero a nodo.

La función Nuevo Nodo () reserva memorias para un nodo, asigna el campo dato y devuelve la dirección del nodo creado.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define Mx99
#define randomize [Srand (time (NULL) ) ]
#define random (num) (rand ( ) % (num) )
typedef int Item;
typedef struct Elemento
{
    Item dato;
    Struct Elemento * siguiente;
}
Item d;
```

```

Nodo * Cabeza, * Ptr;
Int K;
Cabeza= NULL; /* Inicializa cabeza a lista vacia */
randomize; /* El ciclo se termina cuando se genera el numero aleatorio */
for (d= random (Mx); d;)
{
    Insertar Cabeza Lista ( & cabeza, d);
    d= random (Mx);
}
/* Ahora se recorre la lista para escribir los pares */
for ( K=0 , Ptr= cabeza; Ptr;)
{
    f (Ptr→ dato % 2 == 0)
    {
        printf ( "%d", Ptr→ dato );
        K++;
        printf ("%c", ( K % 12?'': '\n' )); /* Cada K datos salta de linea */
    }
    Ptr= Ptr→ siguiente;
}
printf ( "\n\n");
}
Void Insertar Cabeza Lista ( Nodo ** Cabeza, Item entrada)

```

```

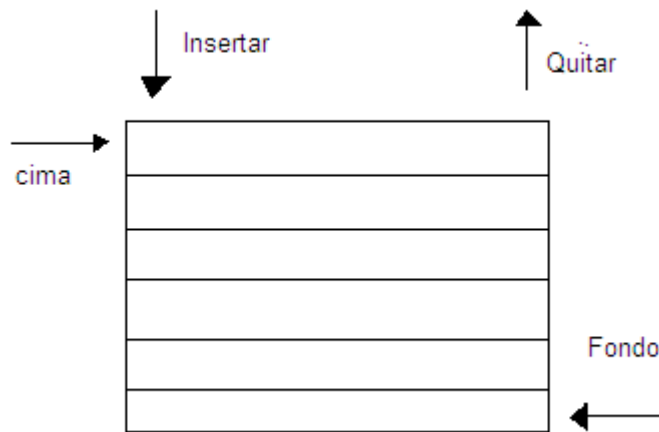
{
    Nodo * nuevo;
    nuevo= NuevoNodo (entrada); /* Enlaza nuevo nodo al frente de la lista */
    nuevo→ siguiente= Cabeza;
    *Cabeza= nuevo; /* Mueve puntero cabeza y apunta al nuevo nodo */
}
Nodo * NuevoNodo (Item X)
{
    Nodo * a;
    a= (Nodo *) malloc ( sizeof ( Nodo ) ); /* Asigna nuevo nodo */
    a→ dato= X;
    a→ siguiente= NULL;
    return a;
}

```

PILAS Y COLAS

Una pila (stack) es una colección ordenada de elementos a los que solo se puede acceder por un único lugar o extremo de la pila. Los elementos de la pila se añaden o quitan (borran) de la misma solo por su parte superior (cima) de la pila.

Una pila puede estar vacía (no tiene elementos) o llena (en caso de tener tamaño fijo).



La operación Insertar (push) sitúa un elemento de la pila y quitar (pop) elimina o quita el elemento de la pila.

Especificaciones de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes (no todas ellas se implementan al definir una pila):

- **Tipo de dato:** Dato que se almacena en la pila.
- **Insertar (push):** Insertar un dato en la pila.
- **Quitar (pop):** Sacar (quitar) un dato de la pila.
- **Pila vacía:** Comprobar si la pila no tiene elementos.
- **Pila llena:** Comprobar si la pila esta llena de elementos.
- **Limpiar pila:** Quitar todos sus elementos y dejar la pila vacía.
- **Tamaño de pila:** Número de elementos máximo que puede contener la pila.
- **Cima:** obtiene el elemento cima de la pila.

El tipo pila implementado con arrays

Una pila se puede implementar mediante arrays o mediante listas enlazadas.

Una implementación estática se realiza utilizando un array de tamaño fijo y una implementación dinámica mediante una lista enlazada.

En C para definir una pila con arrays se utiliza una estructura pila incluyen una lista (array) y un índice o puntero a la cima de la pila; además una constante con el máximo número de elementos. El tipo pila junto al conjunto de operaciones de la pila se puede encerrar en un archivo de inclusión (pila.h).

Insertar (push)

- 1.- Verificar si la pila no está llena.
- 2.- Incrementa en 1 el puntero de la pila.
- 3.- Almacenar elemento en la posición del puntero de la pila.

Quitar (pop)

- 1.- Si la pila no está vacía.
- 2.- Leer el elemento de la posición del puntero de la pila.
- 3.- Decrementar en 1 el puntero de la pila.

Especificaciones del tipo de pila

La declaración de una pila incluye los datos y operaciones ya citados anteriormente:

1. Datos de la pila (tipo TipoData, que es conveniente definirlo mediante typedef)
2. Verificar que la pila no está llena antes de intentar insertar o poner (<<push>>) un elemento en la pila; verificar que una pila no esta vacía antes de intentar sacar (<<pop>>) un elemento de la pila.
3. Pila llena devuelve 1 (verdadero) si la pila está vacío y Ø (falso) en caso contrario.
4. Pila llena devuelve 1 (verdadero) si la pila está llena y Ø(falso) en caso contrario. Estas funciones se utilizan para verificar las operaciones.
5. Limpiar pila. Se limpia o vacía la pila dejándola sin elementos y disponible para otras tareas.
6. Cima, devuelve el valor situado en la cima de la pila, pero no se Decrementar el puntero de la pila, ya que la pila queda intacta.

Declaración

/*archivo pilaría.h*/

#include<stdio.h>

#include<stdlib.h>

#define MaxTamaPila 100

Typedef struct
{TipoDato listapila[Max TamaPila];
int cima;
} Pila;

/*Operaciones sobre la Pila*/

*Void CrearPila(Pila *P);*
*Void Insertar (Pila*P, const TipoDato elemental);*
*TipoDato Quitar (Pila*P);*
*Void Limpiar Pila (Pila*P);*

/*Operación de acceso a Pila*/

TipoDato Cima(Pila P);

/*verificación estado de la Pila*/

Int Pila Vacía (Pila P);
Int Pila Llena (Pila P);

Antes de incluir pilarray.h debe de declararse el TipoDato. Así si se quiere una pila de enteros:

Typedef int TipoDato;
#include "pilarray.h"

En el caso de que la pila fuera de números complejos:

Typedef struct
{flota x,y;

```

}TipoDato;
#include "pilarray.h"

```

Ejemplo:

Escribir un programa que manipule una Pila de enteros, con el tipo definido anteriormente e introduzca un dato de tipo entero.

El programa crea una pila de números enteros, insertar en la pila un dato leído del teclado y visualiza el elemento cima.

```

Typedef int TipoDato;
#include "pilarray.h"
Void main()
{
    Pila P;                                /*crear una pila vacía*/
    Int x;
    Crear Pila (&P);
    Scanf ("%d", &x);                        /*insertar x en la pila P*/
    Insertar (&P, x);
    Printf ("%d\n", cima (P));             /*Visualiza el ultimo elemento*/

    /*Elimina el elemento cima (x) y deja la pila vacía*/

    If (!PilaVacía (P))
        aux=Quitar (&P);
        printf ("%d\n", aux);
        Limpiar Pila (&P);                   /*limpiar la pila, queda vacía*/
    }

    /*Archivo paldromo.c*/
    Typedef char TipoDato;
    #include "pilarray.h"
    #include <ctype.h>
    Int main ( )
    {
        Char palabra [100], ch;
        Pila P;
        Int j, palmo;
        Crear Pila (&P);

        /*lee la palabra*/

        Do {
            Puts (\Palabra a comprobar si es palindromo");
            For (j=0, (ch=get char ( ) ) !='\n');
            }
            Palabra [j]= '\0';

            /*comprueba si es palindrome*/
            Palmo=1
            For (j=0;palmo &&! PilaVacía (P)) ;

```

```

{
Palmo=palabra [j++]=Quitar (&P);
}
Limpiar Pila (&P);
If (palmo)
Printf ("\\n la palabra %5 es un palindromo \\n", palabra);
Else
Printf ("\\n la palabra %5 es un palindromo \\n", palabra);
Printf ("\\n ¿otra palabra?:");
Scanf ("%c %+c", &ch);
} while (tolo wer (ch)=='s');

```

COLAS

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan.

Desde el punto de vista de estructura de datos, una cola es similar a una pila, en donde los datos se almacenan de un modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola. Las acciones que están permitidas en una cola son:

- Creación de una cola vacía.
- Verificación de que una cola está vacía.
- Añadir un dato al final de una cola.
- Eliminación de los datos de la cabeza de la cola.

Operaciones de insertar y quitar en una cola.

Insertar x

x						
---	--	--	--	--	--	--

Frente final

Insertar y

x	y					
---	---	--	--	--	--	--

Frente final

Insertar z

x	y	z				
---	---	---	--	--	--	--

Frente final

Quitar x

y	z					
---	---	--	--	--	--	--

Frente final

Quitar y

z						
---	--	--	--	--	--	--

Frente final

La declaración del tipo de dato cola y los prototipos de las operaciones de la cola se almacenan en un archivo de cabecera “colaarray.h”.

```
#include <stdio.h>
#include <stdlib.h>
#define Max TamQ 100
Typedef struct
{
    Int frente;
    Int final;
    TipoDato lista Q [Max Tam Q];
} cola;

/*Operaciones del tipo de datos cola*/
/*Operaciones de modificación de la cola*/

Void crear cola (Cola*Q, TipoDato elemento);
TipoDato Eliminar Q (Cola*Q);
Void Borrar Cola (cola*Q);

/*Acceso a la cola*/
TipoDato Frente Q (Cola Q);

/*métodos de verificación del estado de la cola*/

Int longitud Q (cola Q);
Int Q vacía (cola Q);
Int Q llena (cola Q);
```

ÁRBOLES

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí, a través de ramas.

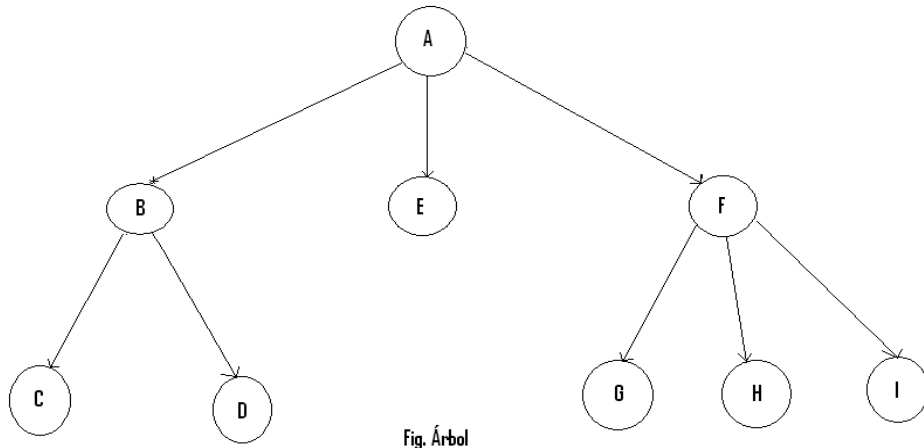
Un árbol consta de un conjunto finito de elementos, denominados nodos y un conjunto finito de líneas dirigidas, denominadas ramas, que conectan los nodos. El número de ramas asociado con un nodo es el grado del nodo.

Un árbol T es un conjunto finito de uno o más nodos tales que:

1.- Hay un nodo diseñado especialmente llamado raíz.

2.- los nodos restantes se dividen en m mayor o igual a Φ conjuntos disjuntos, tales que $T_1, \dots, T_m$, en donde cada uno de estos conjuntos es un árbol. A $T_1, \dots, T_m$ se les denomina subárboles de la raíz.

Si un árbol no está vacío, entonces el primer nodo se llama raíz. obsérvese en la definición que el árbol ha sido definido de modo recursivo, ya que los subárboles se definen como árboles.



El nivel de un nodo es su distancia a la raíz. La raíz tiene una distancia cero de sí misma, por lo que se dice que la raíz está en el nivel Φ . Los hijos d la raíz están en el nivel 1, sus hijos en el nivel 2 y así sucesivamente.

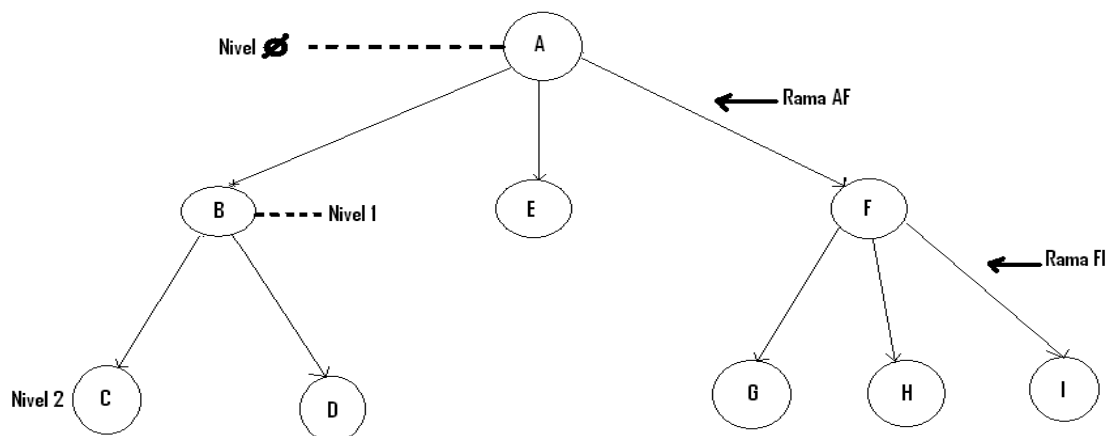
Las flechas que conectan a un nodo a otro se llaman *arcos* o *ramas*.

La *longitud* de un camino es el número de arcos que contiene, o sencillamente, el número de nodos -1.

Un árbol se divide en subárboles. Un subárbol es cualquier estructura conectada por debajo de la raíz.

ÁRBOLES BINARIOS

Un árbol binario es un árbol en el que ningún nodo puede tener más de dos subárboles.



En un árbol binario cada nodo puede tener cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho.

```
#include<stdio.h>
#include<stdlib.h>
typedef char TipoDato;
#include "pilaarray.h"
```

```
void main()
{
```

```
ArbolBinario raiz, a1, a2;
Pila pila;
```

```
nuevoArbol(&a1, NULL,"Maria", NULL);
nuevoArbol(&a2, NULL,"Rodrigo", NULL);
nuevoArbol(&raiz,a1, "Esperanza" ,a2);
Insertar(&pila, raiz);
nuevoArbol(&a1, NULL,"Anyora", NULL);
nuevoArbol(&a2, NULL,"Abel", NULL);
nuevoArbol(&raiz,a1,"M Jesus", a2);
Insertar(&pila, raiz);
a2= Quitar(&pila);
a1= Quitar(&pila);
```

```
nuevoArbol(&raiz,a1,"Esperanza", a2)
```

```
}
```

```
void nuevoArbol(ArbolBinario *raiz,ArbolBinario ramaIzqda,TipoElemento
x, ArbolBinario ramaDrcha)
```

```
{
    *raiz = crearNodo(x);
    (*raiz) -> izdo = ramaIzqda;
    (*raiz) -> dcho = ramaDrcha;
}
```

```
ArbolBinario crearNodo(TipoElemento x)
```

```
{
    ArbolBinario a;
    a = (ArbolBinario) malloc(sizeof(Nodo));
    a -> dato=x;
    a -> dcho=a -> izdo=NULL;
    return a;
}
```

```

/**
 * Árboles binarios en C
 * Operaciones de:
 * Inserción
 * Recorrido inorden, postorden y preorden
 * Uso de malloc
 *
 * @author parzibyte
 * @see https://parzibyte.me/blog
 */
#include <stdio.h>
#include <stdlib.h>

struct Nodo {
    int dato;
    struct Nodo *izquierda;
    struct Nodo *derecha;
};

struct Nodo *nuevoNodo(int dato) {
    // Solicitar memoria
    size_t tamanoNodo = sizeof(struct Nodo);
    struct Nodo *nodo = (struct Nodo *) malloc(tamanoNodo);
    // Asignar el dato e iniciar hojas
    nodo->dato = dato;
    nodo->izquierda = nodo->derecha = NULL;
    return nodo;
}

void insertar(struct Nodo *nodo, int dato) {
    // ¿Izquierda o derecha?
    // Si es mayor va a la derecha
    if (dato > nodo->dato) {
        // Tienes espacio a la derecha?
        if (nodo->derecha == NULL) {
            nodo->derecha = nuevoNodo(dato);
        } else {
            // Si la derecha ya está ocupada, recursividad ;
            insertar(nodo->derecha, dato);
        }
    } else {
        // Si no, a la izquierda
        if (nodo->izquierda == NULL) {
            nodo->izquierda = nuevoNodo(dato);
        } else {
            // Si la izquierda ya está ocupada, recursividad ;
            insertar(nodo->izquierda, dato);
        }
    }
}

```

```
}
```

```
void preorden(struct Nodo *nodo) {  
    if (nodo != NULL) {  
        printf("%d,", nodo->dato);  
        preorden(nodo->izquierda);  
        preorden(nodo->derecha);  
    }  
}
```

```
void inorden(struct Nodo *nodo) {  
    if (nodo != NULL) {  
        inorden(nodo->izquierda);  
        printf("%d,", nodo->dato);  
        inorden(nodo->derecha);  
    }  
}
```

```
void postorden(struct Nodo *nodo) {  
    if (nodo != NULL) {  
        postorden(nodo->izquierda);  
        postorden(nodo->derecha);  
        printf("%d,", nodo->dato);  
    }  
}
```

```
int main(void) {  
    struct Nodo *raiz = nuevoNodo(28);  
    insertar(raiz, 11);  
    insertar(raiz, 96);  
    insertar(raiz, 21);  
    insertar(raiz, 6);  
    insertar(raiz, 97);  
    insertar(raiz, 1);  
    insertar(raiz, 30);  
    insertar(raiz, 10);  
    insertar(raiz, 2);  
    printf("\nImprimiendo preorden\n");  
    preorden(raiz);  
    printf("\nImprimiendo inorden\n");  
    inorden(raiz);  
    printf("\nImprimiendo postorden\n");  
    postorden(raiz);  
}
```

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct nodo {
    int info;
    struct nodo *izq;
    struct nodo *der;
};

struct nodo *raiz=NULL;

int existe(int x)
{
    struct nodo *reco = raiz;
    while (reco != NULL)
    {
        if (x == reco->info)
            return 1;
        else
            if (x>reco->info)
                reco = reco->der;
            else
                reco = reco->izq;
    }
    return 0;
}

void insertar(int x)
{
    if (!existe(x))
    {
        struct nodo *nuevo;
        nuevo = malloc(sizeof(struct nodo));
        nuevo->info = x;
        nuevo->izq = NULL;
        nuevo->der = NULL;
        if (raiz == NULL)
            raiz = nuevo;
        else
        {
            struct nodo *anterior, *reco;
            anterior = NULL;
            reco = raiz;
            while (reco != NULL)
            {
                anterior = reco;
                if (x < reco->info)
                    reco = reco->izq;
                else

```

```

        reco = reco->der;
    }
    if (x < anterior->info)
        anterior->izq = nuevo;
    else
        anterior->der = nuevo;
    }
}

void cantidad(struct nodo *reco,int *cant)
{
    if (reco != NULL)
    {
        (*cant)++;
        cantidad(reco->izq, cant);
        cantidad(reco->der, cant);
    }
}

void cantidadNodosHoja(struct nodo *reco,int *cant)
{
    if (reco != NULL) {
        if (reco->izq == NULL && reco->der == NULL)
            (*cant)++;
        cantidadNodosHoja(reco->izq,cant);
        cantidadNodosHoja(reco->der,cant);
    }
}

void imprimirEntreConNivel(struct nodo *reco, int nivel)
{
    if (reco != NULL) {
        imprimirEntreConNivel(reco->izq, nivel + 1);
        printf("%i(%i) - ", reco->info, nivel);
        imprimirEntreConNivel(reco->der, nivel + 1);
    }
}

void retornarAltura(struct nodo *reco, int nivel,int *altura)
{
    if (reco != NULL)
    {
        retornarAltura(reco->izq, nivel + 1,altura);
        if (nivel>*altura)
            *altura = nivel;
        retornarAltura(reco->der, nivel + 1,altura);
    }
}

```

```

void mayorValor()
{
    if (raiz != NULL)
    {
        struct nodo *reco = raiz;
        while (reco->der != NULL)
            reco = reco->der;
        printf("Mayor valor del arbol:%i\n", reco->info);
    }
}

```

```

void borrarMenor()
{
    if (raiz != NULL) {
        struct nodo *bor;
        if (raiz->izq == NULL)
        {
            bor = raiz;
            raiz = raiz->der;
            free(bor);
        }
        else {
            struct nodo *atras = raiz;
            struct nodo *reco = raiz->izq;
            while (reco->izq != NULL)
            {
                atras = reco;
                reco = reco->izq;
            }
            atras->izq = reco->der;
            free(reco);
        }
    }
}

```

```

void imprimirEntre(struct nodo *reco)
{
    if (reco != NULL)
    {
        imprimirEntre(reco->izq);
        printf("%i - ", reco->info);
        imprimirEntre(reco->der);
    }
}

```

```

void borrar(struct nodo *reco)
{
    if (reco != NULL)
    {

```

```

        borrar(reco->izq);
        borrar(reco->der);
        free(reco);
    }
}

int main()
{
    int cant;
    int altura=0;
    insertar(100);
    insertar(50);
    insertar(25);
    insertar(75);
    insertar(150);
    printf("Impresion entreorden: ");
    imprimirEntre(raiz);
    printf("\n");
    cant=0;
    cantidad(raiz,&cant);
    printf("Cantidad de nodos del arbol:%i\n", cant);
    cant=0;
    cantidadNodosHoja(raiz,&cant);
    printf("Cantidad de nodos hoja:%i\n", cant);
    printf("Impresion en entre orden junto al nivel del nodo:");
    imprimirEntreConNivel(raiz,1);
    printf("\n");
    altura=0;
    retornarAltura(raiz,1,&altura);
    printf("Artura del arbol:%i\n",altura);
    mayorValor();
    borrarMenor();
    printf("Luego de borrar el menor:");
    imprimirEntre(raiz);
    borrar(raiz);
    getch();
    return 0;
}

```


Algoritmos de Ordenación y Búsqueda

La ordenación o clasificación de datos (sort en inglés) es una operación consistente en disponer un conjunto -estructura- de datos en algún determinado orden con respecto a uno de los campos de elementos del conjunto.

Los métodos de ordenación se suelen dividir en dos grandes grupos:

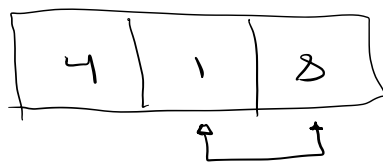
- Directos burbuja, selección, inserción.
- indirectos (avanzados) shell, ordenación rápida, ordenación por mezcla, radix sort.

Ordenación por burbuja

El método de ordenación por burbuja es el más conocido y popular entre estudiantes y aprendices de programación, por su facilidad de comprender y programar; por el contrario, es el menos eficiente y por ello, se aprende su técnica pero suele no utilizarse.

- Algoritmo de la burbuja
tenemos un arreglo (lista) con " n "

elementos, la ordenación por burbuja requiere hasta " $n-1$ " pasadas. En cada pasada se comparan e intercambian elementos.



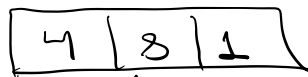
quiere

8	4	1
---	---	---

 como hace a "mano"

Paso 1:

Intercambio el "1" con el "8"

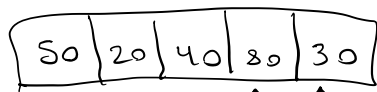
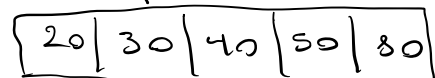


Paso 2: Intercambio el "4" por el "8"

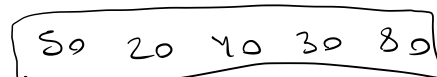


Hicimos 2 pasos y tenemos 3 elementos de ordenados ($3-1$) pasos para tener el arreglo descendo.

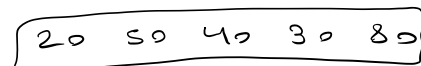
Ejemplo: Tenemos un arreglo de 5 elementos ($A = 50, 20, 40, 80, 30$); queremos el arreglo final en la forma:



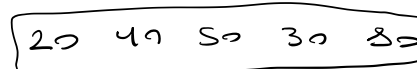
Paso 1: "30" con el "80" :



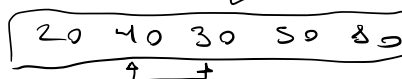
Paso 2: "20" con el "50"



Paso 3: "50" con el "40"



Paso 4: "50" con el "30"



Paso 5: "40" con el "30"

↘ 20 30 40 50 30 ↙

Hicimos 5 pasos para tener el arreglo desordenado. Esta técnica se denomina "ordenación por burbuja" o "ordenación por hundimiento" debido a que los valores más pequeños "burbujan" gradualmente (suben) hacia la cima o parte superior del arreglo de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del arreglo.

• Codificación del algoritmo de la burbuja
La función `ordBurbuja()` implementa el algoritmo de ordenación de la burbuja.

```
void ordBurbuja ( long a[], int n )  
{  
    int interruptor = 1;  
    int pasada;   
    for ( pasada = 0; pasada < n-1 && interruptor;   
          pasada++ )  
        /* bucle externo controla la cantidad de pasadas */  
        interruptor = 0;
```

```

for (j=0; j < n-1; j++)
    if (a[j] > a[j+1])
    {
        /* elementos desordenados, es necesario intercambiarlos */
        long aux;
        interruptor = 1;
        aux = a[j];
        a[j] = a[j+1];
        a[j+1] = aux;
    }
}

```

/* Ordenación por burbuja: arreglo de n elementos
 Se realizan una serie de pases, mientras
 indiceIntercambio > 0 */

```

void ordBurbuja2 (long a[], int n)
{
    int i, j;
    int indiceIntercambio;
    /* i es el índice del último elemento de la
    sublista */
}

```

$i = n - 1;$

/ el proceso continua hasta que no haya intercambios */*

while ($i > 0$)

{

 indiceIntercambio = 0;

/ explorar la sublista $a[0]$ hasta $a[i]$ */*

 for ($j = 0; j < i; j++$)

/ intercambiar pareja y actualizar indiceIntercambio */*

 if ($a[j+1] < a[j]$)

 {

 long aux = $a[j]$;

$a[j] = a[j+1]$;

$a[j+1] = aux$;

 indiceIntercambio = j ;

 }

/ i se pone al valor del indice del ultimo intercambio */*

$i = \text{indiceIntercambio};$

 }

}

Ejemplo 8.8

El programa siguiente ordena una lista de números reales y a continuación los imprime.

```
#include <stdio.h>
/* prototipos */
void imprimir(float a[], int n);
void intercambio(float* x, float* y);
void ordenar (float a[], int n);

int main()

    float a[10]={25.5,34.1,27.6,15.24, 3.27, 5.14, 6.21,7.57,4.61, 5.4};
    imprimir(a,10);
    ordenar (a,10);
    imprimir(a,10);
    return 0;
}

void imprimir(float a[], int n)
{
    int i = 0;
    for (; i < n-1; i++) {
        printf("%f,%c",a[i],((i+1)%10==0 ? '\n' : ' '));
    }
    printf("%f \n",a[n-1]);
}

void intercambio(float* x, float* y)
{
    float aux;
    aux = *x;
    *x = *y;
    *y = aux;
}

/* ordenar burbuja */
void ordenar (float a[], int n)
{
    int i,j;
    for (i = n-1; i>0; i--)
        for (j = 0; j < i; j++)
            if (a[j] > a[j+1])
                intercambio(&a[j],&a[j+1]);
}
```

8.7. BÚSQUEDA EN LISTAS

Los *arrays* (listas y tablas) son uno de los medios principales por los cuales se almacenan los datos en programas C. Debido a esta causa, existen operaciones fundamentales cuyo tratamiento es imprescindible conocer. Estas operaciones esenciales son: la *búsqueda* de elementos y la ordenación o clasificación de las listas.

La *búsqueda* de un elemento dado en un array (lista o tabla) es una aplicación muy usual en el desarrollo de programas en C. Dos algoritmos típicos **que** realizan esta tarea son la ***búsqueda secuencial***

```

#include <stdio.h>

/* prototipos */

void imprimir(float a[], int n);
void intercambio(float* x, float* y);
void ordenar (float a[], int n);

int main()
{

float a[10]={25.5, 34.1, 27.6, 15.24, 3.27, 5.14, 6.21, 7.57, 4.61,
5.4};

imprimir(a,10);
ordenar(a,10);
imprimir(a,10);
return 0;

}
void imprimir(float a[], int n)
{
    int i=0;

    for(; i<n-1; i++) {
        printf("%f, %c",a[i], ((i+1)%10==0 ? '\n' : ' '));
    }
    printf("%f \n ",a[n-1]);
}
void intercambio(float* x, float* y)
{
    float aux;
    aux= *x;
    *x= *y;
    *y= aux;
}

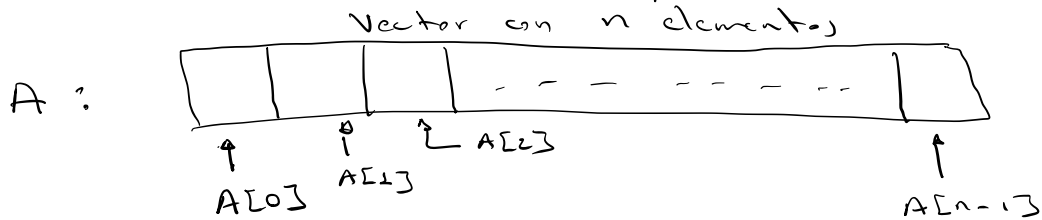
/* ordenador burbuja */

void ordenar (float a[], int n)
{
    int i,j;
    for(i=n-1; i>0; i--)
        for(j=0; j<i; j++)
            if(a[j]>a[j+1])
                intercambio(&a[j],&a[j+1]);
}

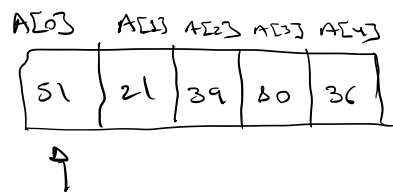
```

"Ordenación por selección"

Consideremos el algoritmo para ordenar un arreglo "A" de enteros en orden ascendente, esto es del número más pequeño al mayor.



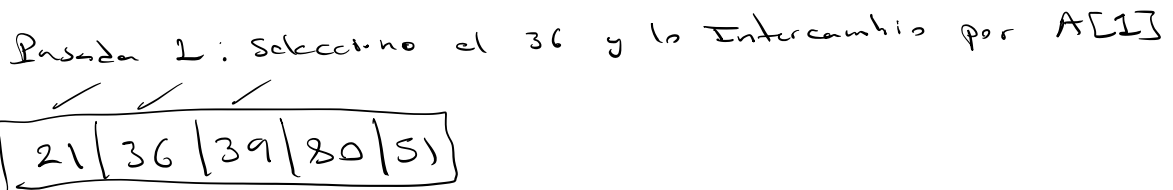
El elemento del vector $A[0]$ es el más pequeño y el último elemento $A[n-1]$ es el más grande.



Paso 0: $\emptyset$



Pasada 0: Seleccionar 21
Intercambiar 21 y $A[0]$



Pasada 2: Selecciono el 51 y lo intercambio por $A[3]$.



- Algoritmo de selección

1. Seleccionar el elemento más pequeño de la lista A . Intercambiarlo con el primer elemento, $A[0]$, tenemos la primer entrada con el elemento más pequeño.
2. Considerar las posiciones de la lista $A[1]$, $A[2]$, $A[3]$ -- y seleccionamos el elemento segundo más pequeño y lo intercambiamos por $A[1]$ y;
3. lo seguimos haciendo así sucesivamente hasta completar la lista.

- Codificación del algoritmo de selección

La función `ordSeleccion()` ordena una lista o vector de números reales de "n" elementos.

* Ordenar un arreglo de "n" elementos de tipo `double` utilizando el algoritmo de ordenación por selección.

```
void ordSeleccion(double a[], int n)
```

```
{
```

```
    int indiceMenor, i, j;
```

```
/* Ordenar a[0] hasta a[n-1] en cada pasada */
```

```

for (i = 0 ; i < n - 1 ; i++)
/* comienzo de la exploración en índice i */
    indiceMenor = i ;
/* se explora la sublista a[i+1] ... a[n-1] */
    for (j = i + 1 ; j < n ; j++)
        if (a[j] < a[indiceMenor])
            indiceMenor = j ;
/* sitúa el elemento más pequeño en a[i] */
    if (i != indiceMenor)
    {
        double aux = a[i] ;
        a[i] = a[indiceMenor] ;
        a[indiceMenor] = aux ;
    }
}
}

```

```

#include <stdio.h>

/* prototipos */

void imprimir(float a[], int n);
void intercambio(float* x, float* y);
void ordBurbuja(float a[], int n);
void ordSeleccion(float a[], int n);

int main()
{
    float a[10]={25.5, 34.1, 27.6, 15.24, 3.27, 5.14, 6.21, 7.57, 4.61,
5.4};

    printf("EL vector escrito es: \n");
    imprimir(a,10);
    ordSeleccion(a,10);
    printf("EL vector en orden ascendente es: \n");
    imprimir(a,10);
    return 0;
}

void imprimir(float a[], int n)
{
    int i=0;

    for(; i<n-1; i++) {
        printf("%.2f, %c",a[i], ((i+1)%10==0 ? '\n' : ' '));
    }
    printf("%.2f \n ",a[n-1]);
}

void intercambio(float* x, float* y)
{
    float aux;
    aux= *x;
    *x= *y;
    *y= aux;
}

/* ordenador burbuja */

void ordBurbuja(float a[], int n)
{
    int i,j;
    for(i=n-1; i>0; i--)
        for(j=0; j<i; j++)
            if(a[j]>a[j+1])
                intercambio(&a[j],&a[j+1]);
}

```

```

}

void ordSeleccion(float a[], int n)
{
    int indiceMenor, i, j;

    /*Ordenador a[0] hasta a[n-1] en cada pasada*/
    for(i=0; i<n-1; i++)
    {
        /*comienzo de la  en indice i*/
        indiceMenor=i;

        /* j explora la sublista a[i+1]...a[n-1]*/
        for(j=i+1;j<n;j++)
            if(a[j]<a[indiceMenor])
                indiceMenor=j;

        /* situa el elemento mas pequeñño en a[i]*/
        if(i!=indiceMenor)
        {
            float aux= a[i];
            a[i]=a[indiceMenor];
            a[indiceMenor]=aux;
        }
    }
}

```

Ordenación por Inserción

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista o archivo que ya está ordenado. Así en el caso de la lista de enteros: $A = 50, 20, 40, 80, 30$.

50 → posición $A[0]$
• Comienzo con 50

Procesar 20: 20 50
• Se inserta 20 en la posición 0;
• 50 se mueve a la posición 1. "A[1]".

Procesar 40: 20 40 50
• Se inserta 40 en la posición 1;
• Se mueve el 50 a la posición 2.

Procesar 80: 20 40 50 80
• El elemento 80 está bien ordenado

Procesar 30: 20 30 40 50 80
• Se inserta el 30 en la posición 1.
• Se desplaza la sublista a la derecha.

• El método de ordenación por inserción.

Algoritmo de ordenación por inserción

• Contempla los siguientes pasos:

1. El primer elemento $A[0]$ se considera ordenado.
2. Se inserta $A[i]$ en la posición correcta delante o detrás de $A[0]$ (sea mayor o menor).
3. Por cada bucle o iteración (desde $i = 1$ hasta $n-1$) se explora la sublista $A[i-1] \dots A[0]$ buscando la posición correcta de inserción.
4. Insertar el elemento en la posición correcta.

* Codificación del algoritmo de inserción
El método `ordInsercion()` tiene dos argumentos, el arreglo $a[]$ y el número de elementos, "n".

```
void ordInsercion(int a[], int n)
{
    int i, j;
    int aux;
    for (i = 1; i < n; i++)
    {
```

/* índice j explora la sublista $a[i-1] \dots a[0]$

buscando la posición correcta del elemento destino,
lo asigna a $a[j]$ +/

$j = i$;

$aux = a[i]$;

+/- localiza el punto de inserción explorando
hacia abajo +/

while ($j > 0$ && $aux < a[j-1]$)

{

+/- desplazar elementos hacia arriba para hacer espacio +/

$a[j] = a[j-1]$;

$j--$;

}

$a[j] = aux$;

}

}

```

#include <stdio.h>

/* prototipos */

void imprimir(float a[], int n);
void intercambio(float* x, float* y);
void ordBurbuja(float a[], int n);
void ordSeleccion(float a[], int n);
void ordInserccion(float a[], int n);

int main()
{
    float a[10]={25.5, 34.1, 27.6, 15.24, 3.27, 5.14, 6.21, 7.57, 4.61,
    5.4};

    printf("EL vector escrito es: \n");
    imprimir(a,10);
    ordInserccion(a,10);
    printf("EL vector en orden ascendente es: \n");
    imprimir(a,10);
    return 0;
}

void imprimir(float a[], int n)
{
    int i=0;

    for(; i<n-1; i++) {
        printf("%.2f, %c",a[i], ((i+1)%10==0 ? '\n' : ' '));
    }
    printf("%.2f \n ",a[n-1]);
}

void intercambio(float* x, float* y)
{
    float aux;
    aux= *x;
    *x= *y;
    *y= aux;
}

/* ordenador burbuja */

void ordBurbuja(float a[], int n)
{
    int i,j;
    for(i=n-1; i>0; i--)
        for(j=0; j<i; j++)
            if(a[j]>a[j+1])
                intercambio(&a[j],&a[j+1]);
}

```



```

}

void ordSeleccion(float a[], int n)
{
    int indiceMenor, i, j;

    /*Ordenador a[0] hasta a[n-1] en cada pasada*/

    for(i=0; i<n-1; i++)
    {

        /*comienzo de la en indice i*/

        indiceMenor=i;

        /* j explora la sublista a[i+1]...a[n-1]*/

        for(j=i+1;j<n;j++)

            if(a[j]<a[indiceMenor])

                indiceMenor=j;

        /* situa el elemento mas pequeño en a[i]*/

        if(i!=indiceMenor)

            {
                float aux= a[i];
                a[i]=a[indiceMenor];
                a[indiceMenor]=aux;
            }
    }
}

void ordInserccion(float a[], int n)
{
    int i,j;
    float aux;
    for (i=1; i<n;i++){
        /*indice j la sublista a[i-1]...a[0]
        buscando la posicion correcta de elementos
        lo asigna a a[j]*/
        j=i;
        aux=a[i];
        /*se localiza el punto de inserccion explorando hacia
abajo*/
        while (j>0 && aux<a[j-1]){
            /*desplazando elementos hacia arriba para

```

```
hacer espacio*/
                a[j]=a[j-1];
                j--;
            }
            a[j]=aux;
        }
    }
```

void ordenacionShell(double a[], int n)

{

int intervalo, i, j, k;

intervalo = n / 2;

while (intervalo > 0)

{

for (i = intervalo; i < n; i++)

{

j = i - intervalo;

while (j >= 0)

{

k = j + intervalo;

if (a[j] <= a[k])

j = -1;

/* así termina el bucle, par ordenado */

else

{

double temp;

temp = a[j];

a[j] = a[k];

a[k] = temp;

j -= intervalo;

}

}

}

intervalo = intervalo / 2;

}

}

Fig. 6.2. Codificación del algoritmo quicksort

La función `quicksort()` refleja el algoritmo citado anteriormente; a esta función se la llama pasando como argumento el array `a[]` y los índices que la delimitan 0 y `n-1` (índice inferior y superior). Llamada a la función:

```
quicksort(a, 0, n-1);
```

Y la codificación recursiva de la función:

```
void quicksort(double a[], int primero, int ultimo)
```

```
{
```

```
    int i, j, central;
```

```
    double pivote;
```

```
    central = (primero + ultimo)/2;
```

```
    pivote = a[central];
```



```
i = primero;
j = ultimo;
do {
    while (a[i] < pivote) i++;
    while (a[j] > pivote) j--;
    if (i <= j)
    {
        double tmp;
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
        i++;
        j--;
        /* intercambia a[i] con a[j] */
    }
} while (i <= j);
if (primero < j)
    quicksort(a, primero, j);
if (i < ultimo)
    quicksort(a, i, ultimo);
}
```

/* mismo proceso con sublista izquierda */

/* mismo proceso con sublista derecha */

```

#include <stdio.h>

/* prototipos */

void imprimir(float a[], int n);
void intercambio(float* x, float* y);
void ordBurbuja(float a[], int n);
void ordSeleccion(float a[], int n);
void ordInserccion(float a[], int n);
void ordenacionShell(float a[], int n);
void quicksort(float a[], int primero, int ultimo);

int main()
{
    int n;
    n=10;

    float a[]={25.5, 34.1, 27.6, 15.24, 3.27, 5.14, 6.21, 7.57, 4.61,
5.4};

    printf("\n EL vector escrito es: \n");
    imprimir(a,n);
    //ordBurbuja(a,n);
    //ordSeleccion(a,n);
    //ordInserccion(a,n);
    //ordenacionShell(a,n);
    quicksort(a,0,n-1);
    printf("\n EL vector en orden ascendente es: \n");
    imprimir(a,n);
    return 0;
}

void imprimir(float a[], int n)
{
    int i=0;

    for(; i<n-1; i++) {
        printf("%.2f, %c",a[i], ((i+1)%10==0 ? '\n' : ' '));
    }
    printf("%.2f \n ",a[n-1]);
}

void intercambio(float* x, float* y)
{
    float aux;
    aux= *x;
    *x= *y;
    *y= aux;
}

```

```

/* ordenador burbuja */

void ordBurbuja(float a[], int n)
{
    int i,j;
    for(i=n-1; i>0; i--)
        for(j=0; j<i; j++)
            if(a[j]>a[j+1])
                intercambio(&a[j],&a[j+1]);
}

void ordSeleccion(float a[], int n)
{
    int indiceMenor, i, j;

    /*Ordenador a[0] hasta a[n-1] en cada pasada*/
    for(i=0; i<n-1; i++)
    {
        /*comienzo de la en indice i*/
        indiceMenor=i;

        /* j explora la sublista a[i+1]...a[n-1]*/
        for(j=i+1;j<n;j++)
            if(a[j]<a[indiceMenor])
                indiceMenor=j;

        /* situa el elemento mas pequeñño en a[i]*/
        if(i!=indiceMenor)
        {
            float aux= a[i];
            a[i]=a[indiceMenor];
            a[indiceMenor]=aux;
        }
    }
}

void ordInserccion(float a[], int n)
{
    int i,j;
    float aux;
    for (i=1; i<n;i++){

```

```

        /*indice j la sublista a[i-1]...a[0]
        buscando la posicion correcta de elementos
        lo asigna a a[j]*/
        j=i;
        aux=a[i];
        /*se localiza el punto de inserccion explorando hacia
abajo*/
        while (j>0 && aux<a[j-1]){
            /*desplazando elementos hacia arriba para
hacer espacio*/
                a[j]=a[j-1];
                j--;
            }
            a[j]=aux;
        }
    }
}

```

```

void ordenacionShell(float a[], int n)
{
    int intervalo, i, j, k;

    intervalo = n/2;
    while (intervalo >0)
    {
        for(i=intervalo; i<n; i++)
        {
            j = i - intervalo;
            while (j>=0)
            {
                k = j + intervalo;
                if(a[j]<=a[k])
                    j = -1;
                else
                {
                    float temp;
                    temp = a[j];
                    a[j] = a[k];
                    a[k] = temp;
                    j -= intervalo;
                }
            }
        }
        intervalo = intervalo / 2;
    }
}

```

```

void quicksort(float a[], int primero, int ultimo)
{
    int i, j, central;

```



```

float pivote;

central = (primero + ultimo)/2;
pivote = a [central];

i = primero;
j = ultimo;

do{
    while(a[i] < pivote) i++;
    while(a[j] > pivote) j--;

    if (i <= j)
    {
        float tmp;
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
        i++;
        j--;
    }
}while (i <= j);

if (primero < j)
    quicksort(a, primero, j);

if (i < ultimo)
    quicksort(a, i, ultimo);
}

```

Gráfica 3D

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>

main()
{
    clrscr();

    char opcion;
    float x,y,z,ax,ay,az,tx,ty,tz,gx,gy,cx,cy;
    int m,n,a;
    m=9; n=2;
    x=0;y=0;z=0;ax=0;ay=0;az=0;tx=0;ty=0;tz=0;gx=0;gy=0;cx=0;cy=0;
    printf("\tPresiona x, y o z para manipular el grafico, s para terminar.")
;
    getch();
    initgraph(&m,&n,"c:\\tc\\bgi");
    do{
        for(x=-3;x<=3;x+=10.0/640.0)
        {
            y=cos(x);
            tx=cos(az)*(x*cos(ay)+(y*sin(ax)+z*cos(ax))*sin(ay))-sin(az)*(y*cos(ax)
)-z*sin(ax));
            ty=sin(az)*(x*cos(ay)+(y*sin(ax)+z*cos(ax))*sin(ay))+cos(az)*(y*cos(ax)
)-z*sin(ax));
            tz=-x*sin(ay)+(y*sin(ax)+z*cos(ax))*cos(ay);
            gx=ty-tx; gy=tz-tx;
            cx=(320/10)*gx+320;
            cy=240-(240/10)*gy;
            putpixel(cx,cy,WHITE);
        }
        // opcion=getch();
        switch(opcion=getch())
        {
            case 'x': ax=ax+0.5;
            break;
            case 'y': ay=ay+0.5;
            break;
            case 'z': az=az+0.5;
            break;
            case 's': a=1;
            break;
            default: printf("No existe tal cosa.");
            break;
        }
    }while(a != 1 );
    closegraph();
    return ;
}
```

BIBLIOGRAFIA

- Peter Grogono. Programación en Pascal, SITESA, 1987
- Niklaus Wirth. Algoritmos y estructuras de datos, Prentice Hall
- Kernighan, W. & Mark, Nelson. El lenguaje de programación C, Prentice Hall; 2ª. Ed. 1991
- D. M. Ritchie. The Data Compression Book, M. and T. Books.
- Bjarne Stroustrup. The C++ Programming Language (2ª, Ed.), Addison Wesley
- Bell, Gleary and Witten. Text Compression, Prentice Hall.
- Robert Sedgewick. Algorithms, Addison Wesley
- J. D. Foley. Computer Graphics: Practice, Addison Wesley
- Schneider, Buel. Advanced Programming and Problems Solving Turbo C++ 3.0, Wiley
- Sara Baase. Computer Algorithms: Introduction to Design and Analysis, Addison Wesley.
- User'S Whit Pascal, Guide Borland. Turbo Pascal 7.0 User'S Guide, Borland
- Estructuras de Datos y Algoritmos: Aho.
- Hopcroft, Ullman; Addison Wesley. Turbo Pascal 7.0, Languaje Guide, Borland

EVALUACIÓN:

- 1º. examen escrito, Unidades 1 y 2, con un peso del 30%
- 2º. examen escrito, Unidades 3, con un peso del 30%
- 3º. examen escrito, Unidades 4, con un peso del 30%
- 4º. examen escrito final 10%

