

ESTRUCTURAS DE DATOS

APUNTES PARA CLASES

(ESTO NO PRETENDE SER UN LIBRO)

*Profesor Depto Ingeniería en Computación
Facultad de Ingeniería
Universidad de Magallanes
jcanuman @kataix*

jose.canuman@umag.cl

Índice

I Referencias	1
II Introducción	2
III Estructuras de datos	4
1. Repaso de C	4
1.1. Muy básico	4
1.2. Redireccionamiento	5
1.3. Codificación y buenas prácticas	6
1.4. Arreglos	8
1.5. Funciones	9
2. Más C	12
2.1. Estructuras	12
2.2. Makefile	13
2.3. Acceso a linea de comandos	15
3. Búsqueda	16
3.1. Búsqueda secuencial(o linear search)	16
3.2. Búsqueda binaria iterativa	17
4. Hashing	18
4.1. Hash Sencillo (Módulo)	19
4.2. Hash de Suma de Caracteres	19
4.3. Hash Polinómico	19
4.4. Hash de Multiplicación	19
4.5. Hash de Jenkins (Jenkins One-at-a-time)	20
4.6. Hash SHA-1 o MD5	20
4.7. Aplicar una Función Hash a un Archivo	20
5. Ordenación	23
5.1. Intercambio	23
5.2. Selección	24
5.3. Inserción	25
5.4. Punteros	26

6. Recursividad	30
6.1. MergeSort(fusión)	31
6.2. Quicksort	32
7. Cuestiones y problemas	34
 IV TAD Lineales	 39
8. Tipo de Dato Abstracto	39
9. TAD Lineales	42
9.1. Estructuras dinámicas	42
9.2. Lista enlazada simple	42
9.3. Listas dobles enlazadas	45
9.4. Listas circulares	46
10. Pilas	47
10.1. Implementación con Lista Enlazada Simple	48
11. Colas	49
11.1. Implementación con arreglo	50
12. Hashing	52
12.1. Direcciónamiento abierto	54
12.2. Encadenamiento separado	55
13. Cuestiones y problemas	55
 V TAD no Lineales	 57
14. Árboles	57
14.1. Árbol general	57
14.2. Binarios	58
14.3. Heap	59
14.4. Árbol Binario de Búsqueda (ABB)	62
14.5. AVL	64
14.6. Árbol 2-3	67
14.7. Árbol B	68
15. Trie	68
16. Construcción del Trie	69
16.1. Paso 1: Insertar "trie"	69
16.2. Paso 2: Insertar "trigger"	69
16.3. Paso 3: Insertar "trick"	69
16.4. Paso 4: Insertar "trip"	69

17. Suffix Tree	70
17.1. Ejemplo de Uso:	70
18. Construcción del Suffix Tree para "banana\$"	70
18.1. Paso 1: Identificar los Sufijos	70
18.2. Paso 2: Construcción del Árbol	71
19. Suffix Array	71
19.1. Paso 1: Generar los Sufijos	71
19.2. Paso 2: Ordenar los Sufijos en Orden Lexicográfico	72
19.3. Paso 3: Construir el Suffix Array	72
20. Grafos	73
20.1. Representación	74
20.2. Caminos mínimos	76
20.3. Árbol cobertor mínimo	78
21. Cuestiones y problemas	79
VI Referencias y Recursos	81
22. Referencias	81
22.1. Figuras	81
22.2. Páginas	81

Referencias



Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Algorithms by Robert Sedgewick and Kevin Wayne

Data Structures and Algorithms by Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, John Hopcroft, Ullman D. Jeffrey

The Art of Computer Programming by Donald Knuth

Data Structures and Algorithm Analysis in C by Mark Allen Weiss

Introduction to Algorithms: A Creative Approach by Udi Manber

Handbook of Algorithms and Data Structures by Gaston H. Gonnet, Ricardo Baeza-Yates

Fundamentos de algoritmia by Brassard, G.; Bratley, P.

GeeksforGeeks
Handbook Baeza-Gonnet
Ejemplos MA Weiss

Introducción

¿Porqué estudiar Estructuras de datos y algoritmos?

Estructuras de datos es uno de los curso fundamentales de Ciencias de la Computación. Si bien es cierto que las estructuras de datos y los algoritmos son difíciles de dominar, no es una hazaña imposible.

- En muchas campañas de contratación de grandes(y nuevas) empresas ponen a prueba las habilidades de resolución de problemas usan las estructuras de datos y algoritmos(DSA) adecuados.
- El aprendizaje de estructuras de datos ayuda a escribir código optimizado. Escribir código optimizado es extremadamente útil ya que permite limitar el uso de recursos en términos de tiempo y espacio.
- Aprendizaje de habilidades avanzadas, todos ellos de importancia inmensa en el mundo real:
 - Manejo de la complejidad
 - Uso sistemático de la memoria
 - Capacidad de reutilización
 - Abstracción

Los científicos informáticos de hoy deben estar capacitados para tener una comprensión profunda de la principios detrás del diseño eficiente del programa, porque sus experiencias de vida ordinaria a menudo no se aplican al diseñar programas de computador.

Cuando te apegas a la informática básica, cuando te apegas a las estructuras de datos y diseño de algoritmos básicos, allí el entrevistador está haciendo lo correcto para evaluar esencialmente las habilidades de resolución de problemas del candidato.

— Gayle Laakman McDowell(Cracking the Code Interview)



De que trata el curso:

Programación y resolución de problemas, con aplicaciones.

Algoritmo: método para resolver un problema.

Estructura de datos: método para almacenar información.

Pero también:

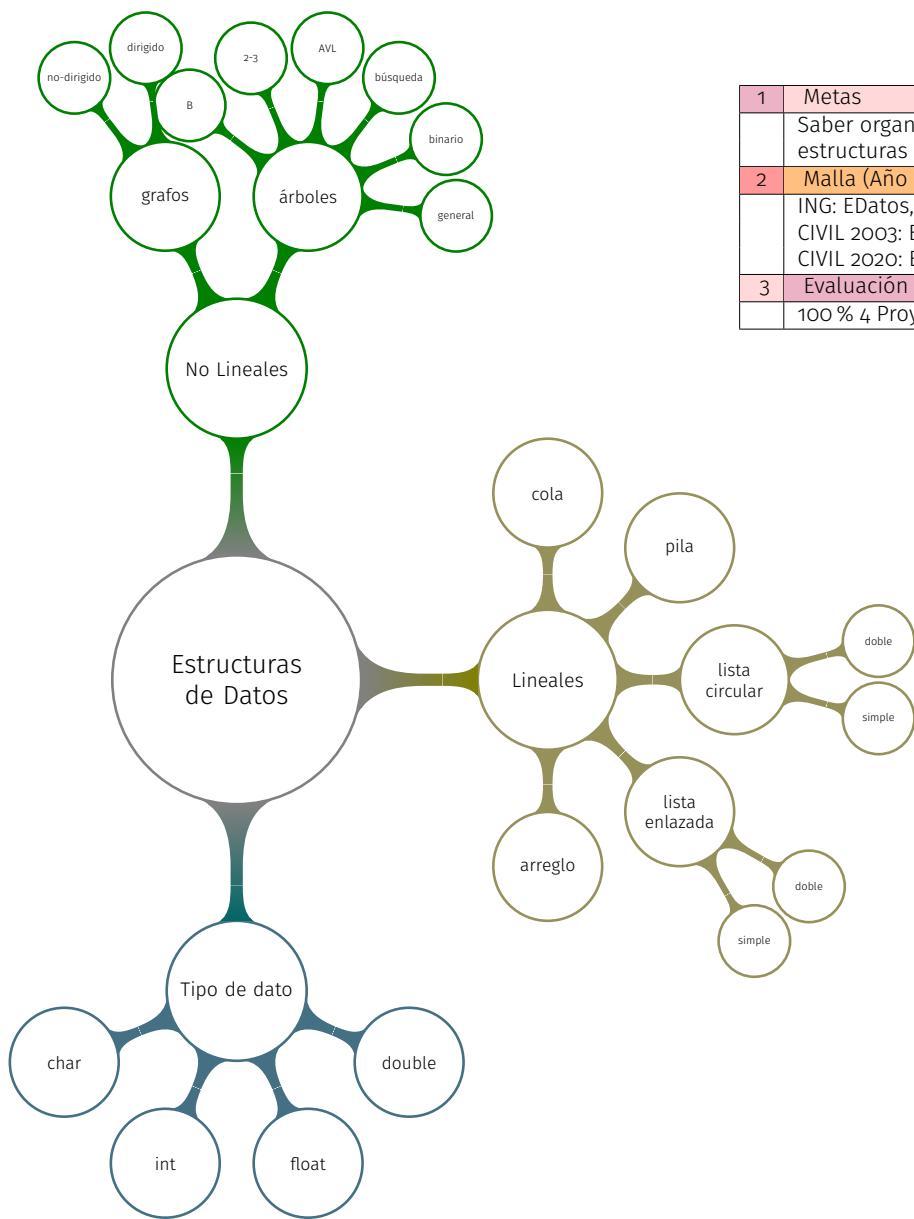
Reutilizar código.

Entender un problema.

Usar lenguaje técnico.

Aplicar cultura informática.

Diferenciar los conceptos de la implementación.



1	Metas
	Saber organizar datos, conocer y saber las operaciones básicas de las estructuras de datos lineales y no lineales.
2	Malla (Año 2/Semestre 2)
	ING: EDatos, Req: Programación II CIVIL 2003: EDatos, Req: Tecnología Informática CIVIL 2020: EDatos, Req: Proramación Estructurada
3	Evaluación
	100 % 4 Proyectos, obligatorios

Estructuras de datos

En el sentido más general, una estructura de datos es cualquier representación de datos y sus operaciones asociadas. Incluso un número entero o de punto flotante almacenado en el computador puede verse como una estructura de datos simple. Más comúnmente, las personas usan el término "estructura de datos" para referirse a una organización o estructuración para una colección de datos o elementos. Una lista ordenada de enteros almacenados en una matriz es un ejemplo de tal estructuración. Al desarrollar un programa o una aplicación, muchos desarrolladores se encuentran más interesados en el tipo de algoritmo utilizado más que en el tipo de estructura de datos implementada. Sin embargo, la elección de la estructura de datos utilizada para un algoritmo en particular es siempre de suma importancia. Cada estructura de datos tiene propiedades únicas y está construida para adaptarse a varios tipos de aplicaciones.

Algoritmos + Estructuras de datos = Programas

– Niklaus Wirth



De hecho, afirmaré que la diferencia entre un mal programador y uno bueno es si considera más importante su código o sus estructuras de datos.

– Linus Torvalds



SECCIÓN 1

Repasso de C

SUBSECCIÓN 1.1

Muy básico

La única forma de aprender un nuevo lenguaje de programación es escribir programas en él.

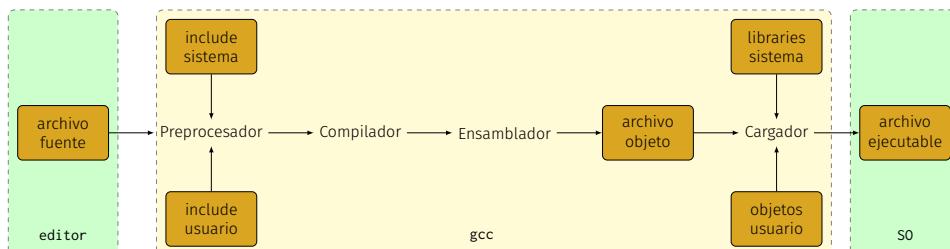
– Dennis Ritchie



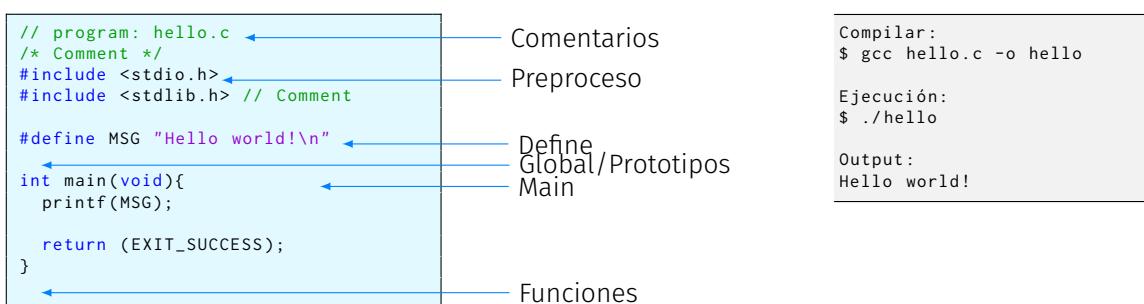
Compilación y Estructura de un programa C

¿Cómo el código fuente se convierte en un programa real que puede ejecutarse en un computador? El código fuente consta de un conjunto de archivos y directorios (carpetas) que contienen código. Este código fuente generalmente debe convertirse a una

forma que el computador puede entender. Este proceso se llama compilación. El programa que realiza esta conversión se llama compilador.



Un programa en lenguaje C contiene una o más funciones, donde una función se define como un grupo de sentencias que realizan una tarea bien definida. Las declaraciones en una función se escriben en una secuencia lógica para realizar una tarea específica. La función `main()` es la función más importante y forma parte de todos los programas en C. Más bien, la ejecución de un programa en C comienza con esta función.



SUBSECCIÓN 1.2

Redireccionamiento

Filosofía detrás de Unix

Escribir programas que hagan una cosa y la hagan bien.

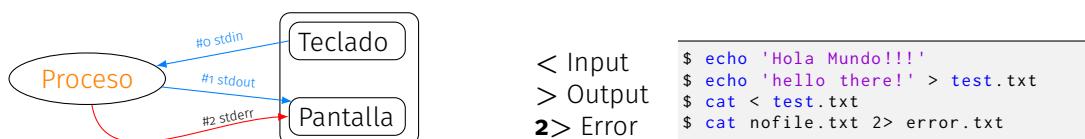
Escribir programas sin salida adicional, no insista en la entrada interactiva.

Escribir programas para manejar flujos de texto, porque esa es una interfaz universal.

Unix también adoptó la filosofía de "peor es mejor".

peor es mejor

Se construyen funciones atómicas que se enfocan en una cosa simple, y luego juntelas para hacer cosas complicadas. No hay variables globales para realizar un seguimiento. Quizás como resultado directo, el diseño de Unix se enfoca en dos componentes principales: Procesos y Archivos. Todo en Unix es un proceso o un archivo. Nada más.



Ejemplo 1

```
#include <stdio.h>
int main() {
    int a;

    fscanf(stdin,"%d",&a);
    fprintf(stdout,"valor de a: %d\n",a);
    fprintf(stderr,"este es un error\n");

    return 0;
}
```

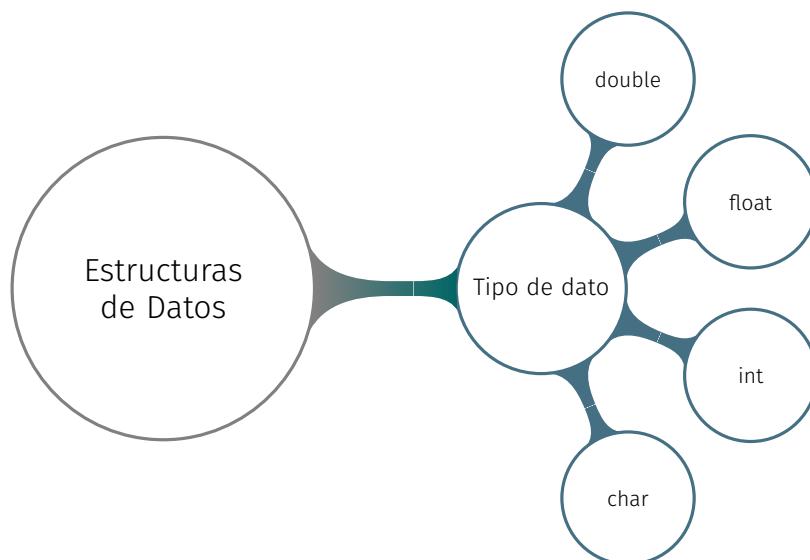
SUBSECCIÓN 1.3

Codificación y buenas prácticas

- ① Siga rigurosamente las reglas más recientes en la documentación del compilador de C estándar.
- ② Utilice nombres de variables lógicas para evitar confusiones.
- ③ El uso adecuado de secuencias de escape como \t o \n mejora la legibilidad de su código.
- ④ Hacer uso de funciones siempre que el código parezca demasiado largo.
- ⑤ No use notaciones abreviadas en exceso si no está muy familiarizado con ellas. A veces, se vuelve difícil de comprender.
- ⑥ El uso de comentarios en C es una muy buena práctica. Te ayuda a entender mejor tu código. Documentación rápida
- ⑦ Revise todo su código antes de la compilación.
- ⑧ Guarde siempre su programa antes de la compilación y tenga cuidado al dar declaraciones en ciclo infinito.
- ⑨ Tenga cuidado al realizar operaciones matemáticas indeterminadas mientras programa como la división de un número por cero.
- ⑩ Al inicializar vectores, evite especificar el tamaño.
- ⑪ Nunca deje punteros sin inicializar.
- ⑫ Use Makefile y en lo posible use indent.

Buenas prácticas

Tipo de dato



```
#include <stdio.h>
int main(void){
    printf("char: %ld\n", sizeof(char));
    printf("int: %ld\n", sizeof(int));
    printf("float: %ld\n", sizeof(float));
    printf("double: %ld\n", sizeof(double));
    printf("long unsigned int: %ld\n", sizeof(long unsigned int));
    printf("unsigned int: %ld\n", sizeof(unsigned int));
    printf("long : %ld\n", sizeof(long));
    return 0;
}
```

Output:
 char: 1
 int: 4
 float: 4
 double: 8
 long unsigned int: 8
 unsigned int: 4
 long : 8

AMD Ryzen 7 5800X 8-Core Processor
 WSL2: 64 bits

(cast)

La conversión de tipos es el proceso en el que el compilador convierte automáticamente un tipo de datos en un programa a otro tipo. El método de conversión de tipos permite a los usuarios convertir los valores presentes en cualquier tipo de datos en otro tipo de datos con la ayuda del operador de conversión, como:

(nombre_tipo) expresión

```
#include <stdio.h>
int main() {
    int total = 17, values = 5;
    double average;

    average = (double) total / values;
    printf("The average of all the values available with us is : %f\n", average );
    return 0;
}
```

Ejemplo 2

La compilación y ejecución del código generaría la siguiente salida del programa:

The average of all the values available with us is : 3.400000

Debe tener en cuenta aquí que la precedencia del operador de cast es mucho mayor que la del operador de división. Cuando este proceso de conversión de tipos es implícito, significa que el compilador es capaz de realizarlo automáticamente. Por el contrario, podemos especificar el tipo de datos explícitamente usando el operador de conversión. Ambos métodos están bien, pero usar el operador de conversión siempre que sea necesario se considera una mejor práctica de programación (explícito).

Casting explícito

```
#include<stdio.h>
int main(){
    float num = 56.3;
    int p = (int)num + 50; // data type casting explicitly

    printf("Let us understand Explicit Type Casting in C\n");
    printf("The value of the digit used is: %f\n", num);
    printf("The value of the variable p is: %d\n",p);

    return 0;
}
```

Ejemplo 3

El output obtenido es el que sigue:

Let us understand Explicit Type Casting in C
 The value of the digit used is: 56.299999
 The value of the variable p is: 106

Algunas funciones de conversión de tipo de datos incorporadas en lenguaje C:

- `atof()`: se usa para convertir el tipo de datos cadena en el tipo de datos `float`.
- `atoi()`: se usa para convertir el tipo de datos cadena en el tipo de datos `int`.
- `atbol()`: se usa para convertir el tipo de datos cadena en el tipo de datos `long`.
- `itoba()`: se usa para convertir el tipo de datos `int` en el tipo de datos cadena.
- `ltoa()`: se usa para convertir el tipo de datos `long` en la cadena de tipo de datos cadena.

Ejemplo 4

```

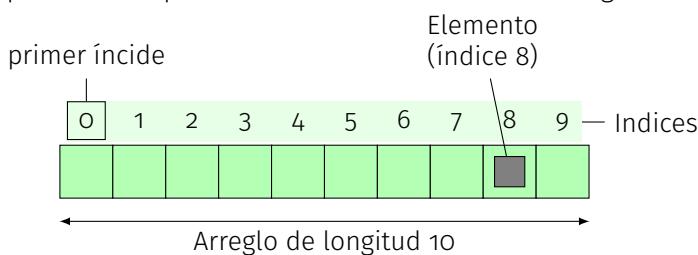
1 #include<stdio.h>
2
3 int main(){
4     int a=1;
5
6     for(int i=0; i<7;i++){
7         int p = 1 << i;
8         printf("%d", a & p ? 1:0);
9     }
10    printf("\n");
11
12    return 0;
13 }
```

Output:
1000000

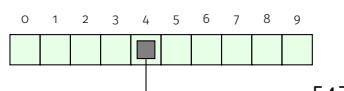
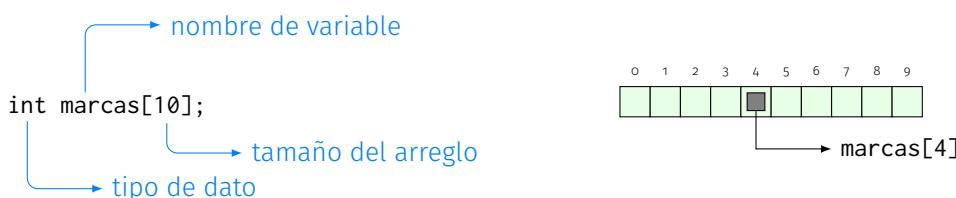
SUBSECCIÓN 1.4

Arreglos

Un arreglo es una colección de elementos que tienen el mismo tipo de datos. Los elementos se almacenan en ubicaciones de memoria consecutivas y son referenciados por un índice (también conocido como el *subíndice*). El subíndice es un número entero que se utiliza para identificar un elemento del arreglo.



Declaración



```

1 int totales[100];
2 char nombre[30];
3 float promedios[100];
```

Asignación

```
int marcas[5] = {90, 82, 78, 95, 88};
```

90	82	78	95	88
----	----	----	----	----

```
int marcas[5] = {90, 45};
```

90	45	0	0	0
----	----	---	---	---

```
int marcas[] = {90, 82, 78, 95, 88, 54};
```

90	82	78	95	88	54
----	----	----	----	----	----

```
int marcas[5] = {};
```

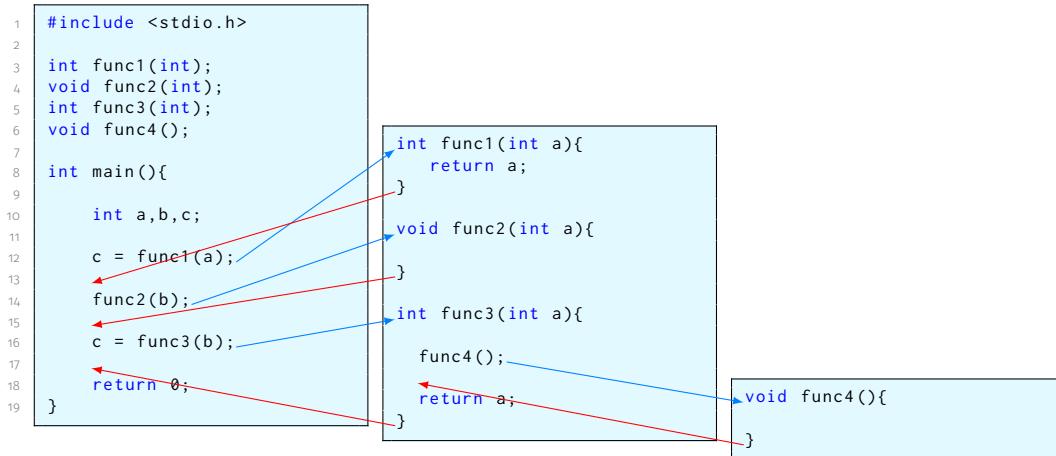
0	0	0	0	0
---	---	---	---	---

```
int i=6; int a[i]; X
```

```
1 int i, marcas[10];
2 for(i=0;i<10;i++)
3     marcas[i]=i;
4     marcas[3] = 99;
```

0	1	2	3	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0	1	2	99	4	5	6	7	8	9
0									

Funciones en funciones



Ejemplo 6

```

#include <stdio.h>
#include <math.h>
#define N 11
float calculateSD(float data[]);
int main() {
    int i;
    float data[N];
    printf("Enter %d elements: ",N);
    for (i = 0; i < N; ++i)
        scanf("%f", &data[i]);
    printf("\nStandard Deviation = %.6f",
          calculateSD(data));
    return 0;
}

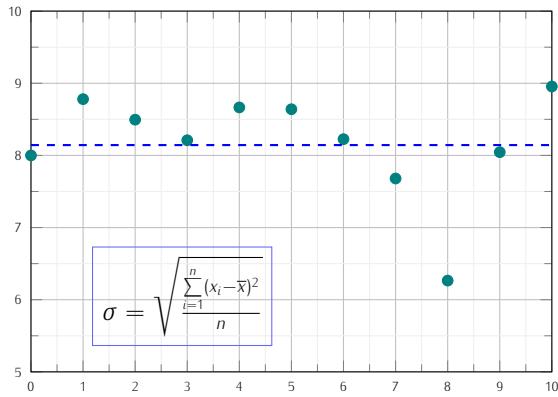
float calculateSD(float data[]) {
    float sum = 0.0, mean, SD = 0.0;
    int i;
    for (i = 0; i < N; ++i) {
        sum += data[i];
    }
    mean = sum / N;
    printf("Media = %.6f", mean);
    for (i = 0; i < N; ++i) {
        SD += pow(data[i] - mean, 2);
    }
    return sqrt(SD / N);
}

```

```

$ gcc stand.c -o stand
/usr/bin/ld: /tmp/cclQ3Exr.o: in function `calculateSD':
stand.c:(.text+0x1eb): undefined reference to `pow'
/usr/bin/ld: stand.c:(.text+0x233): undefined reference
to `sqrt'
collect2: error: ld returned 1 exit status
$ gcc stand.c -o stand -lm
$ ./stand
Media = 8.178182
Standard Deviation = 0.705653

```



Ejercicios

1.5.1 Implemente la función: `void print_array(int a[], int count);`, donde `a[]` es un arreglo de tamaño `count`.

1.5.2 Escriba una función que cuente el número total de elementos duplicados en un arreglo.

1.5.3 Implementar una función que imprima sólo los elementos impares de un arreglo.

1.5.4 Escribir una función que elimine un elemento en una posición dada de un arreglo.

1.5.5 Escribir una función que encuentre el 2do menor elemento de un arreglo.

1.5.6 Implementar una función que imprima la representación binaria de un número entero.

1.5.7 Escribir una función que invierta la representación de un número entero.

1.5.8 En los siguientes códigos, indique la cantidad de veces que se realiza el 'block':

```
for(i=0;i<10;i++)
    block;
```

```
for(i=0;i<10;i++)
    for(j=1; j<10;j*=2)
        block;
```

```
for(i=1;i<1000;i*=2)
    block;
```

```
for(i=1000;i>=1;i/=2)
    block;
```

```
for(i=0;i<10;i++)
    for(j=0; j<10;j++)
        block;
```

```
for(i=0;i<10;i++)
    for(j=0; j<=i;j++)
        block;
```

Ejercicios

- 1.5.1** Dado el siguiente conjunto A={-2,8,12,34,2,67,8,5,-78,99,-34,78} se pide ordenar usando los métodos: burbuja, selección e inserción.
- 1.5.2** Dado el conjunto B={8,4,2,7,9,12,-45,3,-46,5,34,23,10,0,33,-10,-8,6} se pide ordenar entre los índices [0..7] con el método burbuja, en el rango [8..13] con el método inserción, y el rango sobrante con el método selección.
- 1.5.3** Dado el arreglo ordenado anterior, buscar el elemento 67 con búsqueda binaria iterativa.
- 1.5.4** Implemente la función `merge` que mezcla 2 arreglos ordenados.

SECCIÓN 2

Más C

SUBSECCIÓN 2.1

Estructuras

Una estructura es una colección de una o más variables agrupadas bajo un solo nombre para un manejo cómodo. Las variables en una estructura se llaman *miembros* y puede tener cualquier tipo, incluidos arreglos u otros estructuras.

```
struct Date{
    int day;
    int month;
    int year;
};

struct Book{
    char title[80];
    char author[80];
    float price;
    char isbn[20];
};

struct Library_member{
    char name[80];
    char address[200];
    long member_number;
    float fines[10];
    struct Date dob;
    struct Date enrolled;
};

struct Library_book{
    struct Book b;
    struct Date due;
    struct Library_member who[100];
};
```

Instancia y acceso a los miembros

```
struct Date{
    int day;
    int month;
    int year;
} today, tomorrow;

struct Date next_monday;
struct Date next_week[7];
```

```
today.day
next_week[0].day;
```

Asignación

```
struct Library_member m = {
    "James Bond",
    "Av Bulnes 01855",
    007,
    {0.10, 2.58, 0.13, 1.10},
    {18,9,1959},
    {1,4,1978}
};

struct Library_member m;
strcpy(m.name, "James Bond");
strcpy(m.address, "Av Bulnes 01855");
m.member_number = 007;
m.fines[0] = 0.10; m.fines[1] = 2.58; m.fines[2] = 0.13; m.fines[3] = 1.10;
m.fines[4] = 0.00; m.fines[5] = 0.00; m.fines[6] = 0.00; m.fines[7] = 0.00;
m.fines[8] = 0.00; m.fines[9] = 0.00;
m.dob.day = 18; m.dob.month = 9; m.dob.year = 1959;
m.enrolled.day = 1; m.enrolled.month = 4;
m.enrolled.year = 1978;

struct Library_member tmp;
tmp = m;
```

Nota

```
int a[10];
int b[10];

b=a; // No existe esta asignación
```

```
struct A {
    int array[10];
};

struct A a, b;
a = b;
```

typedef

```
1 struct Library_member;
2 typedef struct Library_member Member;
3 typedef int ElementType;
4
5 Member m;
6 Member p[100];
7 ElementType X;
8 ElementType P[];
```

Paso de estructuras a funciones

```
void display_member(struct Library_member M){
    printf("Name: %s\n", M.name);
    printf("Nro: %d\n", M.member_number);
    ...
}

// typedef struct Library_member Member;
void display_member(Member M){
    printf("Name: %s\n", M.name);
    printf("Nro: %d\n", M.member_number);
    ...
}
```

```
...
Member M[100];
...
for(i=0;i<100;i++){
    display_member(M[i]);
    printf("-----\n");
}
...
```

Ejercicios

- 2.1.1** Se desea crear un sistema de notas, cree las estructuras necesarias para almacenar los datos.
- 2.1.2** Se desea implementar un sistema de matrículas de automóviles, cree las estructuras de datos necesarias.
- 2.1.3** Se desea implementar un repositorio de documentos, cree las estructuras necesarias para la búsqueda de un documento dada una palabra.
- 2.1.4** Se desea crear un juego de naipes(cartas), debe crear la estructura necesaria para el almacenamiento de dichos valores.
- 2.1.5** Implementar las estructuras necesarias para definir un sistema de polinomios.

SUBSECCIÓN 2.2

Makefile

make es una herramienta de automatización de compilación. Las herramientas de automatización de compilación como make permiten a los desarrolladores describir los pasos de compilación y ejecutarlos todos a la vez.

makefile es archivo de texto que describe el proceso de compilación de sus programas. El comando make se usa para ejecutar convenientemente las instrucciones en el Makefile.

Dependencias

Las dependencias especifican cómo se relaciona cada archivo de la aplicación final con los archivos de origen.

Por ejemplo, `myapp: main.o 2.o 3.o`, significa que `myapp` depende de los archivos `main.o`, `2.o` y `3.o`.

En un archivo `makefile`, escribimos estas reglas escribiendo el nombre del objetivo, dos puntos, espacios o tabulaciones y luego una lista de archivos separados por espacios o tabuladores que se utilizan para crear el archivo objetivo.

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

Si deseamos crear varios archivos, podemos usar el objetivo falso `all`.

```
all: myapp
```

Reglas

Ahora veamos las reglas que describen cómo crear un objeto. Si es necesario reconstruir un archivo, ¿qué comando se debe usar?

Podría ser tan simple como usar `gcc -c 2.c`.

La mayoría de las reglas consisten en un comando simple que podría haberse escrito en la línea de comandos.

Para nuestro ejemplo, el archivo `makefile` que incluye las reglas, se convierte en:

```
myapp: main.o 2.o 3.o
      gcc -o myapp main.o 2.o 3.o

main.o: main.c a.h
      gcc -c main.c

2.o: 2.c a.h b.h
      gcc -c 2.c

3.o: 3.c b.h c.h
      gcc -c 3.c
```

IMPORTANTE: debe usar la tecla `TAB` o `➡⬅` para las líneas que necesiten separarse del margen izquierdo.

target:	dependencies
➡⬅	command

Una versión más completa sería:

```
all: myapp

#Que compilador
CC=gcc

#Donde estan los archivos incluidos
INCLUDE=.

#Opciones de desarrollo
CFLAGS=-g -Wall -ansi

#Opciones de produccion o lanzamiento
#CFLAGS=-O -Wall -ansi

myapp: main.o 2.o 3.o
      $(CC) -o myapp main.o 2.o 3.o

main.o: main.c a.h
      $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c

2.o: 2.c a.h b.h
      $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
```

```
3.o: 3.c b.h c.h
 $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

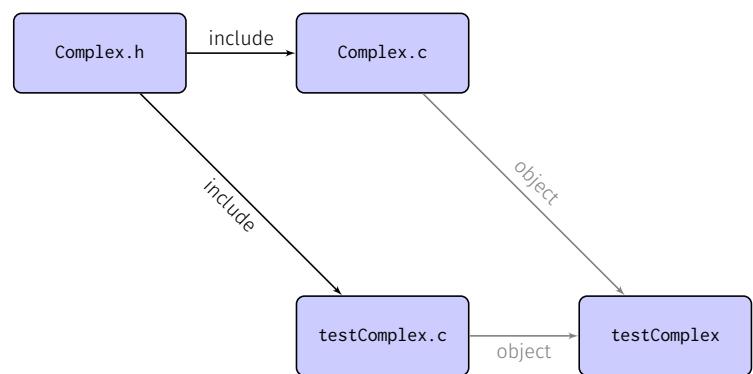
Manual GNU make

```
#include <stdio.h>
...
typedef struct _complex Complex;
struct _complex {
    float real;
    float imag;
};

// Funciones
//struct _complex AddComplex(struct _complex A, struct
//    _complex B);
Complex AsigComplex(float R, float I){
    ...
}
Complex AddComplex(Complex A, Complex B){
    ...
}

void PrintComplex(Complex A){
    ...
}

int main(){
    Complex A,B;
    ...
    return 0;
}
```

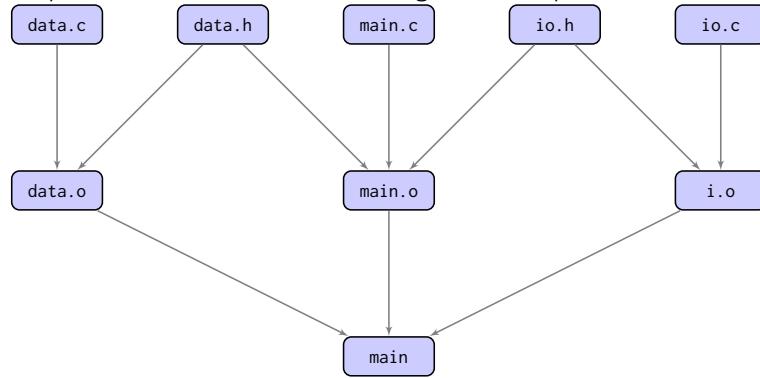


Ejercicios

2.2.1 Implemente un makefile para el ejercicio de Complex.

2.2.2 Probar el makefile presentado en el ejemplo.

2.2.3 Se pide crear el makefile de las siguientes dependencias:



SUBSECCIÓN 2.3

Acceso a línea de comandos

La linea de comando se puede acceder a través de dos parámetros a `main()`. Por convención estos se llaman `argc` y `argv`.

`argc` es la cantidad de argumentos agregados incluyendo el nombre del ejecutable.

`argv` es un arreglo a los punteros de cada argumento.

Ejemplo 7

```

1 #include<stdio.h>
2
3 int main(int argc, char* argv[]){
4
5     for(int i=0; i<argc ;i++){
6         printf("[%d]:%s\n", i, argv[i]);
7     }
8     printf("\n");
9
10    return 0;
11 }
```

Output:
./commline uno 2 tres 4444
[0]:./commline
[1]:uno
[2]:2
[3]:tres
[4]:4444

Ejemplo 8

```

1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(int argc, char* argv[]){
5     int sum = 0;
6
7     if(argc>1){
8         for(int i=0; i<argc ;i++){
9             sum += atoi(argv[i]);
10        }
11        printf("sum:%d\n",sum);
12    }
13
14    return 0;
15 }
```

Output:
./sumline 1 2 3 88
sum:94

SECCIÓN 3

Búsqueda

SUBSECCIÓN 3.1

Búsqueda secuencial(o linear search)**Algoritmo 1** Búsqueda secuencial #1

```

1: procedure BSEC1( $V, x$ )       $\triangleright V[0..n - 1]$ 
2:    $r \leftarrow n$ 
3:    $i \leftarrow 0$ 
4:   while  $i \leq n - 1 \wedge r = n$  do
5:     if  $x = V[i]$  then
6:        $r \leftarrow i$ 
7:      $i \leftarrow i + 1$ 
8:   return  $r$ 
```

Algoritmo 2 Búsqueda secuencial #2

```

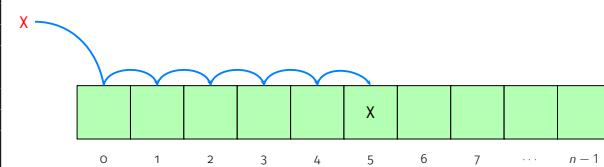
1: procedure BSEC2( $V, x$ )       $\triangleright V[0..n - 1]$ 
2:    $r \leftarrow 0$ 
3:   while  $r \leq n - 1 \wedge x \neq V[r]$  do
4:      $r \leftarrow r + 1$ 
5:   return  $r$ 
```

```

1 //bseq2.c
2 #include <stdio.h>
3
4 int bseq2(int v[], int x, int n){
5     int r=0;
6
7     while((r <= n-1)&&(x != v[r])){
8         r = r+1;
9     }
10    return r;
11}
12
13 int main(){
14     int a[] = {3,6,23,45,7,88,9,2,66};
15     int x = 88;
16
17     printf("%d\n",bseq2(a,x,9));
18 }
```

Compilación:
\$ gcc bseq2.c -o bseq2

Ejecución:
\$./bseq2
5
\$



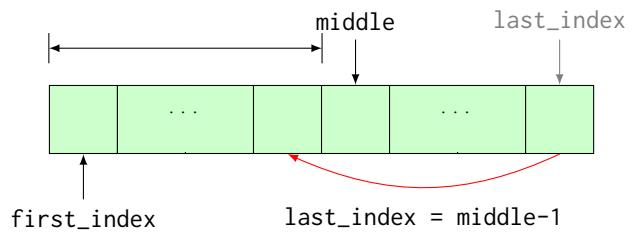
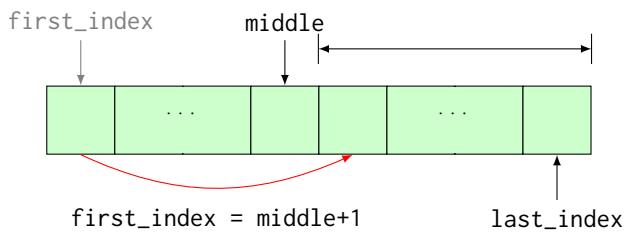
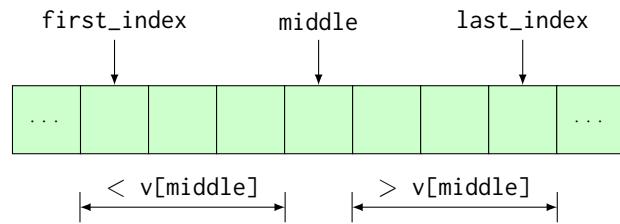
SUBSECCIÓN 3.2

Búsqueda binaria iterativa

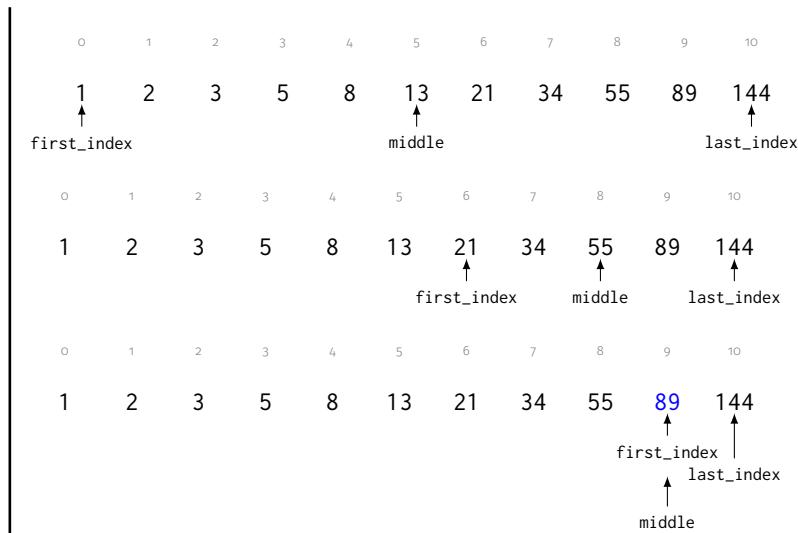
Condición: arreglo ordenado

```

1 int BinarySearchIte(int v[], int first_index, int last_index, int element)
2 {
3     while (first_index <= last_index)
4     {
5         int middle = first_index + (last_index - first_index) / 2;
6         if (v[middle] == element)
7             return middle;
8         if (v[middle] < element)
9             first_index = middle + 1;
10        else
11            last_index = middle - 1;
12    }
13    return -1;
14 }
```



Ejemplo 9 | element=89



SECCIÓN 4

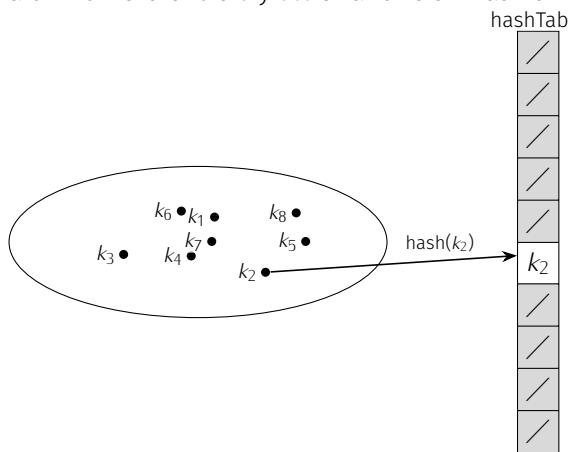
Hashing

"Hash" es realmente un término amplio con diferentes significados formales en diferentes contextos. Un *hash* es una función a la que se hace referencia como función hash que toma como objetos de entrada y genera una cadena o un número. Los objetos de entrada suelen ser miembros de tipos de datos básicos como cadenas, enteros o tipos más grandes compuestos por otros objetos como estructuras definidas por el usuario. La salida suele ser un número o una cadena. El sustantivo "hash" a menudo se refiere a esta salida.

Las principales propiedades que debe tener una función hash son:

- Debe ser fácil de calcular y
- las salidas deben ser relativamente pequeñas.

Ejemplo: Digamos que queremos hacer un hash de números en el rango de 0 a 999.999.999 a un número entre 0 y 99. Una función hash simple puede ser $h(x) = x \bmod 100$.



SUBSECCIÓN 4.1

Hash Sencillo (Módulo)

Una de las funciones hash más simples es tomar un valor y calcular su módulo con un número primo. Este tipo de hash es común en tablas hash básicas.

Ejemplo

```
int simpleHash(int key, int tableSize) {
    return key % tableSize;
}
```

Aplicación: Se usa para mapear claves numéricas a índices en una tabla hash de tamaño fijo.

SUBSECCIÓN 4.2

Hash de Suma de Caracteres

Una función hash simple para cadenas es sumar los valores ASCII de todos los caracteres en la cadena y luego aplicar una operación de módulo.

Ejemplo

```
int sumHash(char *str, int tableSize) {
    int hash = 0;
    while (*str) {
        hash += (int)(*str);
        str++;
    }
    return hash % tableSize;
}
```

Aplicación: Mapear cadenas (como nombres o identificadores) a índices en una tabla hash.

SUBSECCIÓN 4.3

Hash Polinómico

Otra técnica común es tratar la cadena como un polinomio de base b y tomar el valor modulado para reducir colisiones.

Ejemplo

```
int polynomialHash(char *str, int tableSize) {
    int hash = 0;
    int p = 31; // Un número primo pequeño
    int p_pow = 1;

    while (*str) {
        hash = (hash + (*str - 'a' + 1) * p_pow) % tableSize;
        p_pow = (p_pow * p) % tableSize;
        str++;
    }
    return hash;
}
```

Aplicación: Mapeo de cadenas con mejor distribución y menor probabilidad de colisiones.

SUBSECCIÓN 4.4

Hash de Multiplicación

El método de multiplicación utiliza una constante multiplicativa para generar el hash, que luego se reduce mediante una operación de módulo.

Ejemplo

```
int multiplicationHash(int key, int tableSize) {
    float A = 0.6180339887; // Una constante (phi)
    return (int)(tableSize * (key * A - (int)(key * A)));
}
```

Aplicación: Generación de índices para claves numéricas con menor probabilidad de colisiones.

SUBSECCIÓN 4.5

Hash de Jenkins (Jenkins One-at-a-time)

Una función hash desarrollada por Bob Jenkins, diseñada para proporcionar un buen rendimiento en general con claves de diferentes longitudes.

Ejemplo

```
unsigned int jenkinsHash(char *key, int tableSize) {
    unsigned int hash = 0;
    while (*key) {
        hash += (unsigned char)(*key);
        hash += (hash << 10);
        hash ^= (hash >> 6);
        key++;
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash % tableSize;
}
```

Aplicación: Generalmente utilizado en aplicaciones de sistemas donde se requiere una distribución muy uniforme.

SUBSECCIÓN 4.6

Hash SHA-1 o MD5

Para criptografía o situaciones donde se requiere un hash seguro, se pueden usar funciones hash criptográficas como SHA-1 o MD5. Estos no son típicamente usados en tablas hash debido a su costo computacional, pero son ejemplos útiles de funciones hash.

Ejemplo Básico de MD5 en C

```
#include <openssl/md5.h>

void computeMD5(char *str, unsigned char digest[16]) {
    MD5_CTX ctx;
    MD5_Init(&ctx);
    MD5_Update(&ctx, str, strlen(str));
    MD5_Final(digest, &ctx);
}
```

Aplicación: Verificación de integridad de datos, generación de firmas digitales.

SUBSECCIÓN 4.7

Aplicar una Función Hash a un Archivo

En este ejemplo, vamos a leer un archivo de texto, calcular un hash para su contenido, y luego mostrar el resultado. Utilizaremos el algoritmo Jenkins One-at-a-time para generar el hash del contenido del archivo.

Código en C para Aplicar una Función Hash a un Archivo

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

// Función de hash de Jenkins (One-at-a-time)
unsigned int jenkinsHash(unsigned char *key, size_t len) {
    unsigned int hash = 0;
    for (size_t i = 0; i < len; i++) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}

// Función para leer el contenido de un archivo y calcular el hash
unsigned int hashFile(const char *filename) {
    FILE *file = fopen(filename, "rb"); // Abrir el archivo en modo binario
    if (!file) {
        perror("No se puede abrir el archivo");
        exit(EXIT_FAILURE);
    }

    // Determinar el tamaño del archivo
    fseek(file, 0, SEEK_END);
    long fileSize = ftell(file);
    fseek(file, 0, SEEK_SET);

    // Leer el contenido del archivo
    unsigned char *buffer = (unsigned char *)malloc(fileSize * sizeof(unsigned char));
    if (!buffer) {
        perror("No se puede asignar memoria");
        fclose(file);
        exit(EXIT_FAILURE);
    }

    fread(buffer, sizeof(unsigned char), fileSize, file);
    fclose(file);

    // Calcular el hash del contenido
    unsigned int hash = jenkinsHash(buffer, fileSize);

    // Liberar la memoria del buffer
    free(buffer);

    return hash;
}

int main() {
    const char *filename = "archivo.txt"; // Nombre del archivo a leer
    unsigned int hashValue = hashFile(filename);

    printf("El hash del archivo '%s' es: %u\n", filename, hashValue);

    return 0;
}
```

Cuando se ejecuta el programa, leerá el contenido del archivo, calculará su hash y mostrará algo como:

El hash del archivo 'archivo.txt' es: 123456789

Aplicaciones

- **Verificación de Integridad:** Usar hashes para verificar que un archivo no ha sido modificado.
- **Detección de Duplicados:** Comparar hashes de archivos diferentes para detectar duplicados.

- **Cálculo de Firmas Digitales:** Aunque en la criptografía se usan funciones hash más complejas, este ejemplo básico muestra el concepto.

Este ejemplo demuestra cómo se puede aplicar una función hash a un archivo, que es una técnica común en muchas aplicaciones de informática.

SECCIÓN 5

Ordenación

Ordenar corresponde al proceso de reubicar(permutar) un conjunto de elementos en orden ascendente o descendente. El objetivo principal es facilitar la recuperación de dichos elementos. La gran mayoría de los métodos de ordenación se basan en comparaciones entre los elementos.

SUBSECCIÓN 5.1

Intercambio**Bubble Sort**

Método sencillo que consiste en revisar cada elemento del conjunto con el siguiente, intercambiándolos de posición si están en orden inverso. Es necesario revisar varias veces toda la lista hasta que ya no se necesiten intercambios. Este método también se llama por intercambio directo.

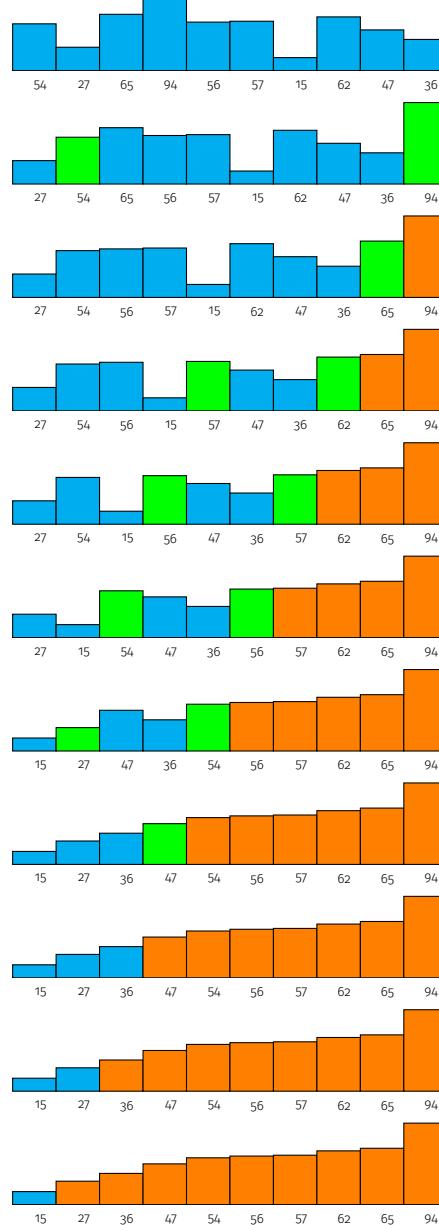
```
void bubbleSort(int v[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (v[j] > v[j + 1])
                swap(&v[j], &v[j + 1]);
        }
    }
}
```

Algoritmo 3 Ordenación por intercambio

```
1: procedure SWAPSORT( $V$ )  $\triangleright V[0..n - 1]$ 
2:   for  $i \leftarrow 1$  to  $n - 1$  do
3:     for  $j \leftarrow n - 1$  to  $i$  do
4:       if  $V[j] < V[j - 1]$  then
5:          $V[j] \leftrightarrow V[j - 1]$   $\triangleright \text{swap}(V[j], V[j - 1])$ 
```

$$T(n) \in O(n^2)$$

0:	54	27	65	94	56	57	15	62	47	36
1:	27	54	65	56	57	15	62	47	36	94
2:	27	54	56	57	15	62	47	36	65	94
3:	27	54	56	15	57	47	36	62	65	94
4:	27	54	15	56	47	36	57	62	65	94
5:	27	15	54	47	36	56	57	62	65	94
6:	15	27	47	36	54	56	57	62	65	94
7:	15	27	36	47	54	56	57	62	65	94
8:	15	27	36	47	54	56	57	62	65	94
9:	15	27	36	47	54	56	57	62	65	94
	15	27	36	47	54	56	57	62	65	94



SUBSECCIÓN 5.2

Selección**Selection Sort**

Este método consiste en seleccionar el menor elemento del conjunto y a continuación intercambiarlo con el elemento que ocupa la primera posición del vector. Hay que repetir esta operación con los $n - 1$ elementos restantes, luego los $n - 2$ elementos restantes y así sucesivamente hasta que solo quede un elemento.

```

1 void selectionSort(int v[], int n) {
2     int i, j, min_idx;
3
4     for (i = 0; i < n - 1; i++) {
5         min_idx = i;
6         for (j = i + 1; j < n; j++)
7             if (v[j] < v[min_idx])
8                 min_idx = j;
9
10        swap(&v[min_idx], &v[i]);
11    }
12}
```

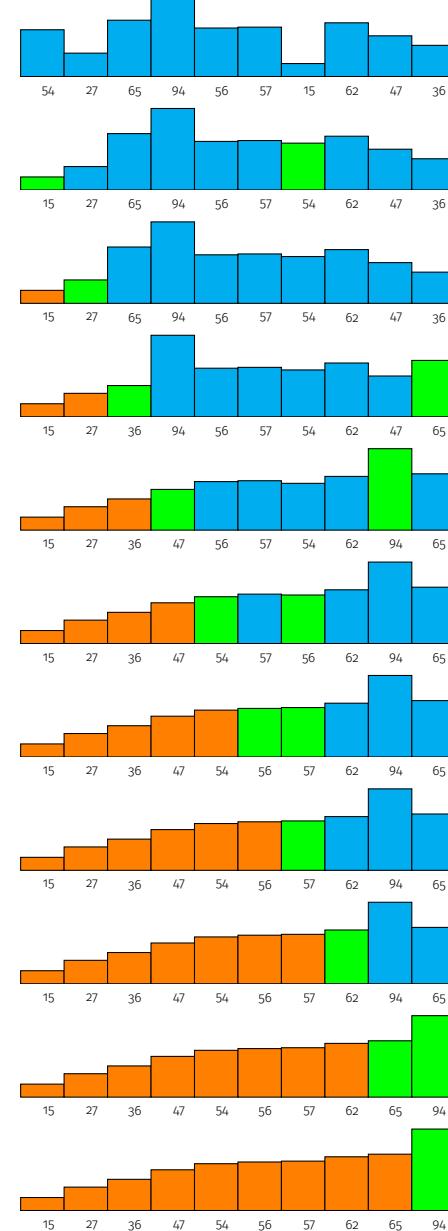
Algoritmo 4 Ordenación por Selección

```

1: procedure SELECTIONSORT( $V$ )  $\triangleright V[0..n - 1]$ 
2:   for  $i \leftarrow 1$  to  $n - 1$  do
3:      $k \leftarrow i - 1$ 
4:      $m \leftarrow V[i - 1]$ 
5:     for  $j \leftarrow i$  to  $n - 1$  do
6:       if  $V[j] < m$  then
7:          $k \leftarrow j$ 
8:          $m \leftarrow V[j]$ 
9:        $V[k] \leftarrow V[i - 1]$ 
10:       $V[i - 1] \leftarrow m$ 
```

$$T(n) \in O(n^2)$$

0:	54	27	65	94	56	57	15	62	47	36
1:	15	27	65	94	56	57	54	62	47	36
2:	15	27	65	94	56	57	54	62	47	36
3:	15	27	36	94	56	57	54	62	47	65
4:	15	27	36	47	56	57	54	62	94	65
5:	15	27	36	47	54	57	56	62	94	65
6:	15	27	36	47	54	56	57	62	94	65
7:	15	27	36	47	54	56	57	62	94	65
8:	15	27	36	47	54	56	57	62	94	65
9:	15	27	36	47	54	56	57	62	65	94
	15	27	36	47	54	56	57	62	65	94



$$t_{\min}(n) = n - 1 \in O(n)$$

$$t_{\max}(n) = \frac{n(n-1)}{2} \in O(n^2)$$

$$\bar{T}(n) \in O(n^2)$$

SUBSECCIÓN 5.3

Inserción**Insertion Sort**

Método utilizado por los jugadores de cartas(naipes). En cada paso, a partir de $i = 2$, el i -ésimo elemento de la secuencia se procesa y se transfiere a la secuencia destino(que ya esta ordenada), insertándolo en el lugar correspondiente.

```
void insertionSort(int v[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = v[i];
        j = i - 1;

        while (j >= 0 && v[j] > key) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}
```

Algoritmo 5 Ordenación por Inserción

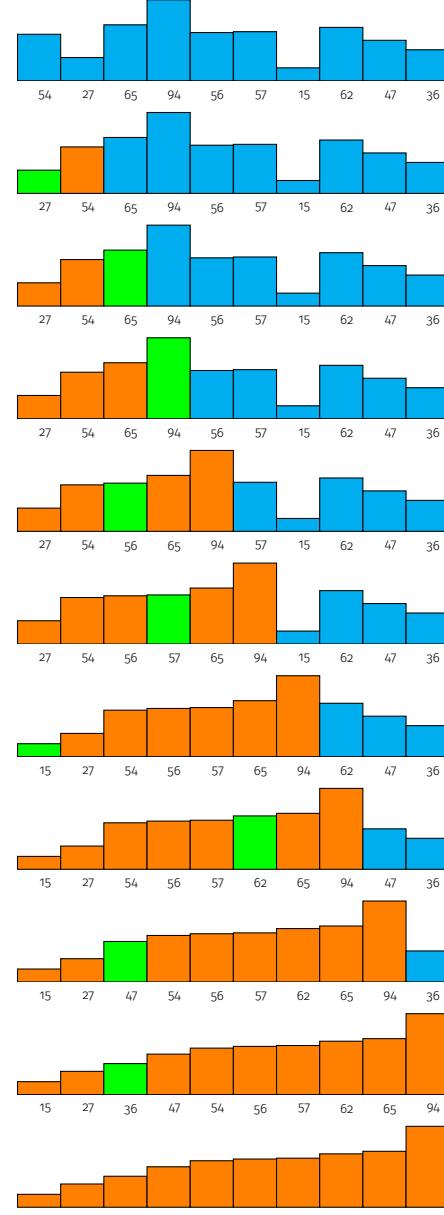
```
1: procedure INSERTIONSORT( $V$ ) ▷  $V[0..n - 1]$ 
2:   for  $i \leftarrow 1$  to  $n - 1$  do
3:      $x \leftarrow V[i]$ 
4:      $j \leftarrow i - 1$ 
5:     while  $j \geq 1 \wedge x < V[j]$  do
6:        $V[j + 1] \leftarrow V[j]$ 
7:        $j \leftarrow j - 1$ 
8:      $V[j + 1] \leftarrow x$ 
```

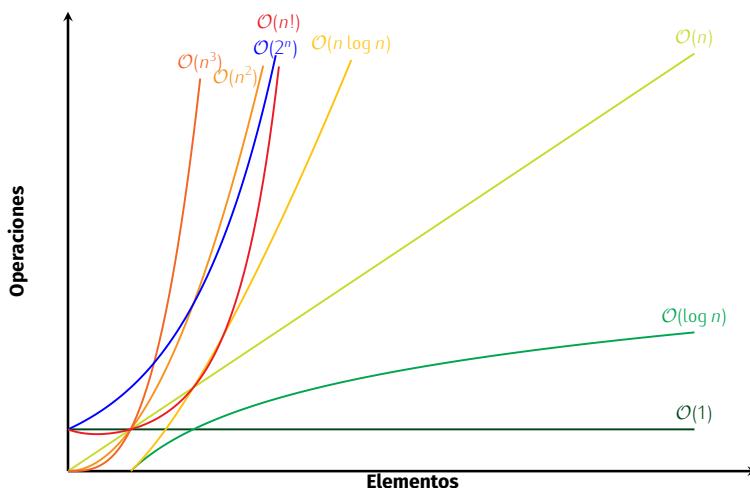
$$T(n) \in \mathcal{O}(n^2), \in \mathcal{O}(n)$$

0:	54	27	65	94	56	57	15	62	47	36
1:	27	54	65	94	56	57	15	62	47	36
2:	27	54	65	94	56	57	15	62	47	36
3:	27	54	65	94	56	57	15	62	47	36
4:	27	54	56	65	94	57	15	62	47	36
5:	27	54	56	57	65	94	15	62	47	36
6:	15	27	54	56	57	65	94	62	47	36
7:	15	27	54	56	57	62	65	94	47	36
8:	15	27	47	54	56	57	62	65	94	36
9:	15	27	36	47	54	56	57	62	65	94
	15	27	36	47	54	56	57	62	65	94

Otros métodos

- HeapSort
- MergeSort
- QuickSort



Big O

SUBSECCIÓN 5.4

Punteros

Un puntero es una variable. Los punteros permiten:

- Lograr la llamada por referencia en funciones
- manejar arreglos de forma eficiente
- manejar estructuras de forma eficiente
- crear estructuras dinámicas
- ...

Nota: ya lo ha usado con `scanf()`.

Operadores

Operador '*'

```
int *p;
int* p;
char *s;
char* s;
```

Operador '&'

```
char c = 'A';
char *p;
p = &c;
```

p

0x1132

c

'A'

0X1132

```

1 #include <stdio.h>
2 int main() {
3     int a = 7;
4     int *b; //
5     b = &a; //

6     printf("%d\n", a);
7     printf("%p\n", b);
8     printf("%p\n", &a);
9     printf("%d\n", *b); // a == *b
10    //printf("%d\n", *a); // <--?
11    printf("%p\n", &b);

12    b = 8; //
13    printf("%p\n", b);
14    printf("%d\n", *b); // <--?

15    return 0;
16 }
```

Output:

```

7
0xffff5e85b8ac
0xffff5e85b8ac
7
0xffff5e85b8b0
0x8
[1] 11458 segmentation fault ./ej

```

Diferencia

```

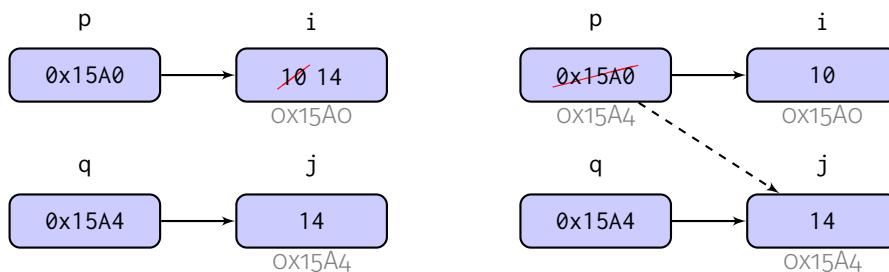
1 int i=10, j=14;
2 int *p = &i;
3 int *q = &j;
4
5 *p = *q;

```

```

1 int i=10, j=14;
2 int *p = &i;
3 int *q = &j;
4
5 p = q;

```



Pasar punteros a funciones

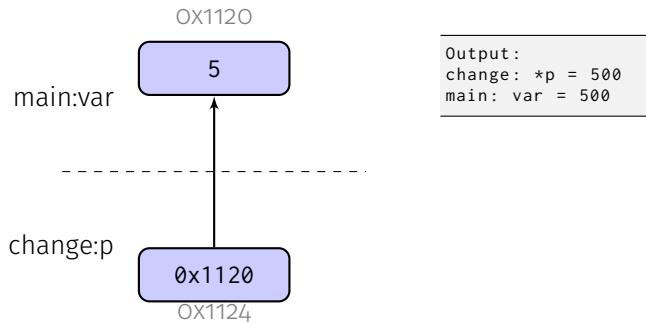
Llamada por referencia.

```

#include <stdio.h>
void change(int* p);

int main(void){
    int var = 5;
    change(&var);
    printf("main: var = %i\n", var);
    return 0;
}
void change(int* p)
{
    *p *= 100;
    printf("change: *p=%i\n", *p);
}

```



Memoria dinámica, punteros y arreglos

Existe una parte del programa en ejecución donde el heap, puede variar en tamaño. Su tamaño está controlado por llamadas a las rutinas dinámicas de asignación de memoria que define C: `malloc` y `free`. Los arreglos en C tienen un problema fundamental: su tamaño debe fijarse cuando se escribe el programa. Los arreglos dinámicos se les puede fijar el tamaño en el tiempo de ejecución y se puede cambiar tan a menudo como sea requerido.

malloc:MemoryALLOCate

Todo los programas requieren un puntero que contendrá la dirección en que comienza el bloque de memoria.

Ej: <code>long p[100];</code>	Sería como: <code>long *p; p = malloc(100 * sizeof(long));</code>
----------------------------------	--

terminamos con la misma cantidad de memoria, 400 bytes, pero almacenada en el heap en lugar de la pila(stack) o segmento de datos.

free

Finalmente, cuando todas las variables ya han sido manipuladas, el almacenamiento debe ser liberado. Estrictamente hablando, esto no es necesario ya que el heap es

parte del proceso. Cuando el proceso termina, toda la memoria asociada con él es reclamada por el sistema operativo. Sin embargo, es una buena práctica liberar la memoria en caso de que el programa se altere para que una rutina "única" sea llamada repetidamente. Llamar repetidamente a una rutina que no puede desasignar memoria es garantía de que el programa eventualmente fallará, incluso si la cantidad de la memoria en cuestión es pequeña. Dichos errores se conocen como *fugas de memoria* (*memory leaks*).

Ejemplo 10

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void)
4 {
5     unsigned i, s=100;
6     double *p;
7     if((p = malloc(s * sizeof(double))) == NULL) {
8         fprintf(stderr, "Cannot allocate %u bytes for %u doubles\n", s * sizeof(double), s);
9         return -1;
10    }
11    for(i = 0; i < s; i++)
12        p[i] = i;
13
14    free(p);
15    return 0;
16 }
```

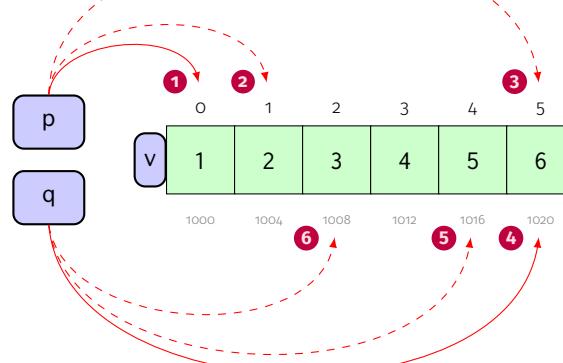
Los punteros se pueden usar para acceder a los elementos de un vector(arreglo) evitando usar construcciones que involucran "[]". Los punteros en C se escalan automáticamente según el tamaño del elemento señalado.

```

#include <stdio.h>

int main(void){
    int v[6]={1,2,3,4,5,6};
    int *p,*q;

    p = v; ①
    printf("%d\n",*p);
    p++; ②
    printf("%d\n",*p);
    p += 4; ③
    printf("%d\n",*p);
// -----
    q = v + 5; ④
    printf("%d\n",*q);
    q--; ⑤
    printf("%d\n",*q);
    q -= 2; ⑥
    printf("%d\n",*q);
}
```

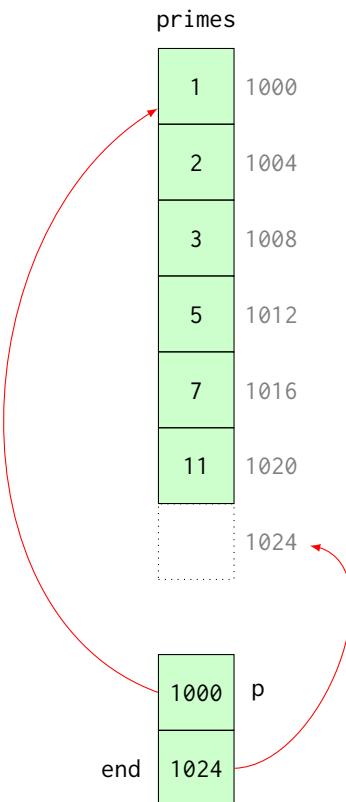


Un axioma de C establece que $a[i]$ es equivalente a $*(\mathbf{a} + i)$. Y cuando es de dos dimensiones $a[j][k] = *(\mathbf{a} + j * \text{largo_fila} + k)$.

```

1 #include <stdio.h>
2
3 int sum(int*, int);
4
5 int main(void){
6     int primes[6] = {1,2,3,5,7,11};
7
8     printf("%d\n", sum(primes,6));
9
10    return 0;
11}
12
13 int sum(int *p, int sz){
14     int *end = p + sz;
15     int total = 0;
16
17     while(p < end)
18         total += *p++;
19
20     return total;
21}

```



```

1 int a[8] = { 10, 20, 30, 40, 50, 60, 70, 80 };
2 int *p = a;
3
4 printf("%i\n", a[3]);
5 printf("%i\n", *(a + 3));
6 printf("%i\n", *(p + 3));
7 printf("%i\n", p[3]);
8 printf("%i\n", 3[a]); // *(a+3)=*(3+a)

```

Ejercicios

- 5.4.1** Escriba un programa para leer dos números y sumarlos usando punteros.
- 5.4.2** Escriba un programa para ingresar elementos en un arreglo y los imprima usando punteros.
- 5.4.3** Escriba un programa para ingresar elementos en un arreglo y busque un elemento usando punteros.
- 5.4.4** Escriba un programa para devolver múltiples valores de una función usando punteros.

SECCIÓN 6

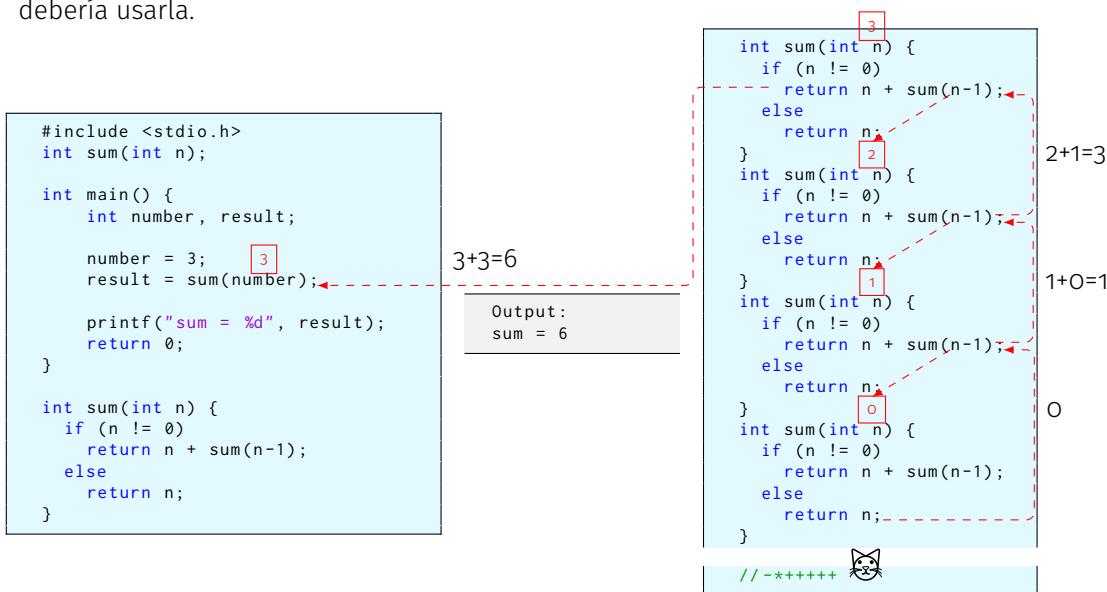
Recursividad

La recursividad consiste en funciones que se llaman a sí mismas, evitando el uso de ciclos.

Una de las principales razones por las que se utiliza la recursividad es que muchas estructuras de datos son, por naturaleza, recursivas. Esto significa que es más intuitivo procesarlos recursivamente. Un buen ejemplo de esto es el recorrido para un árbol binario; verá que la solución recursiva tiene un código simple y elegante.

Los algoritmos recursivos tienden a tener menos pasos que las soluciones no recursivas. Esto significa que son más fáciles de leer para los humanos. Sin embargo, no son necesariamente muy fáciles de rastrear o depurar, ya que debe estar familiarizado con la pila de llamadas y cómo se usa. Tener varias versiones simultáneas del mismo procedimiento puede resultar confuso.

Para cada solución recursiva, hay una solución iterativa (el mismo problema se puede resolver usando ciclos). Puede encontrar difícil la recursividad y preguntarse por qué debería usarla.

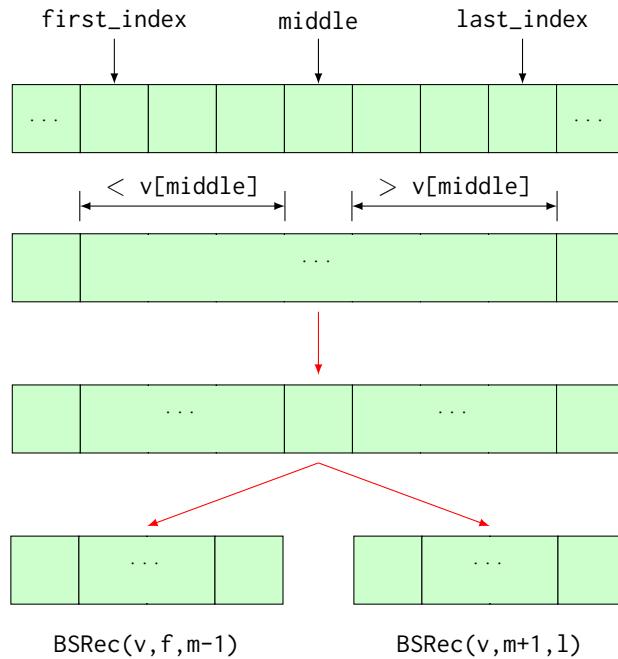


Nota:Dividir para reinar

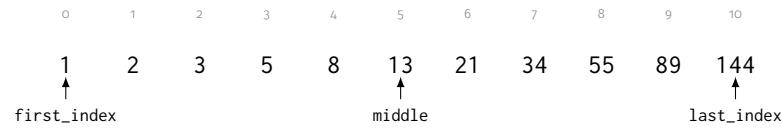
```

1 int BinarySearchRec(int v[], int first_index, int last_index, int element)
2 {
3     if (last_index >= first_index)
4     {
5         int middle = first_index + (last_index - first_index) / 2;
6         if (array[middle] == element)
7             return middle;
8         if (array[middle] > element)
9             return BinarySearchRec(v, first_index, middle - 1, element);
10        return BinarySearchRec(v, middle + 1, last_index, element);
11    }
12    return -1;
13}

```



Ejemplo 11 | element=90



SUBSECCIÓN 6.1

MergeSort(fusión)

Este algoritmo funciona por *equilibrado binario*. Descompone el arreglo en dos subarreglos de tamaños similares, los ordena recursivamente y combina los resultados en un arreglo ordenado.

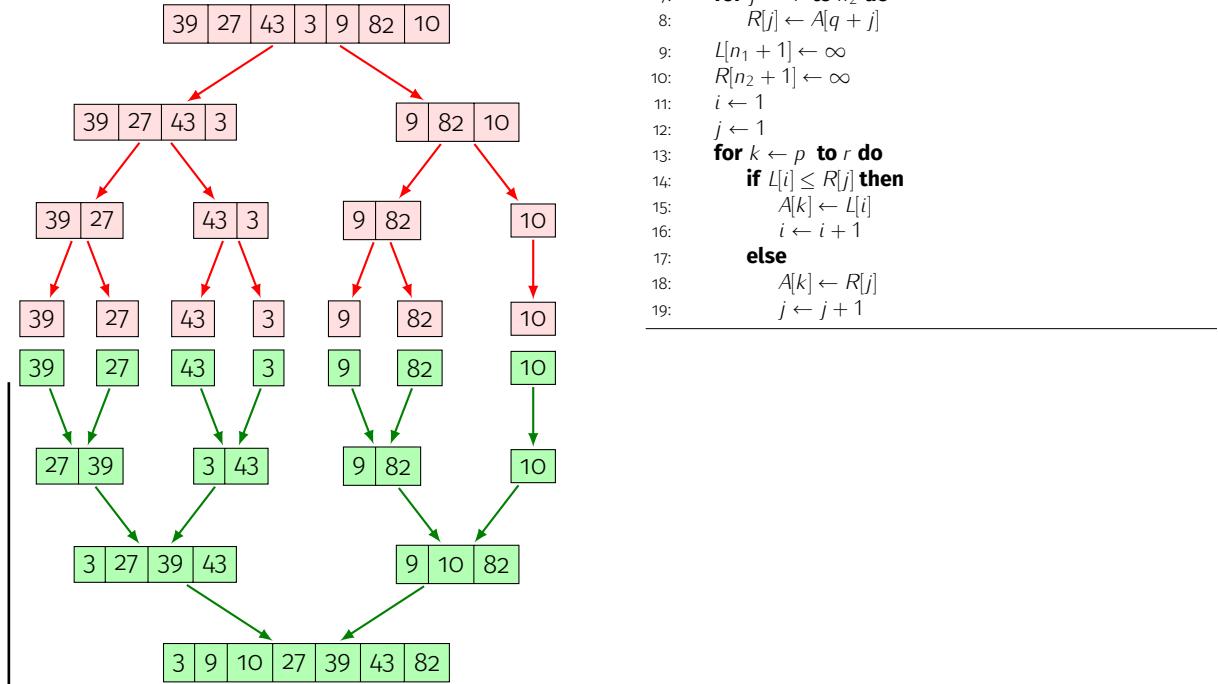
Input:	S O R T I N G E X A M P L
Divide:	S O R T I N G E X A M P L
Recursividad Izq:	I N O R S T G E X A M P L
Recursividad Der:	I N O R S T A E G L M P X
Merge:	A E G I L M N O P R S T X

Algoritmo 6 Ordenación por mezcla

```

1: procedure MERGESORT( $A, low, high$ )
2:   if  $low < high$  then
3:      $mid \leftarrow \lfloor (low + high)/2 \rfloor$ 
4:     MERGESORT( $A, low, mid$ )
5:     MERGESORT( $A, mid + 1, high$ )
6:     MERGE( $A, low, mid, high$ )

```



Ejemplo 12

SUBSECCIÓN 6.2

Quicksort

QUICKSORT es otro algoritmo recursivo de ordenación. El funcionamiento ocurre al dividir un arreglo en subarreglos más pequeños *antes* de la recursión, de forma que la mezcla de los subarreglos ordenados sea trivial. Esto es:

1. Elegir un elemento *pivote* del arreglo.
2. Partitionar el arreglo en tres subarreglos, conteniendo los elementos más pequeños que el pivote, el pivote en sí y los elementos mayores.
3. Realizar QUICKSORT en el primer y último subarreglos.

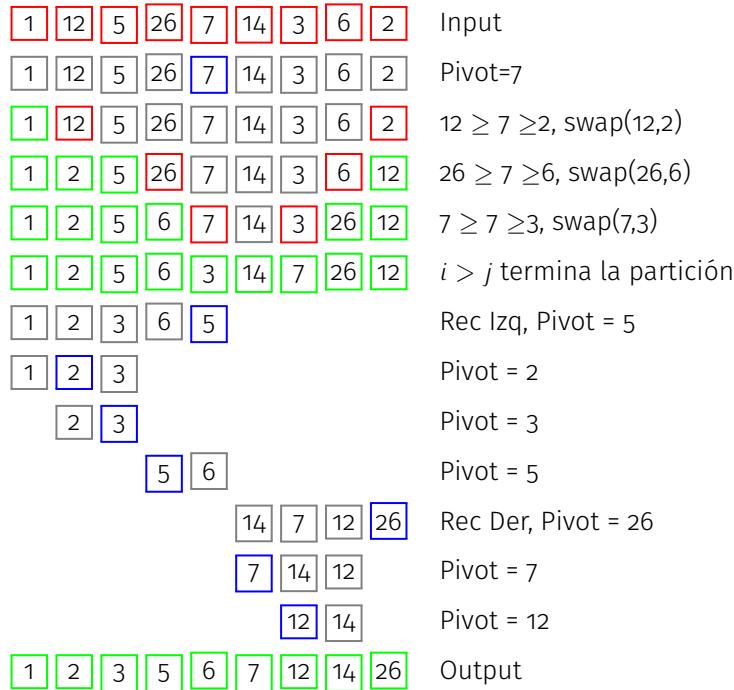
Input:	S O R T I N G E X A M P L
Pivote:	S O R T I N G E X A M P L
Partición:	A G O E I N L M P T X S R
Recursividad Izq:	A E G I L M N O P T X S R
Recursividad Der:	A E G I L M N O P R S T X

Algoritmo 7 Merge

```

1: procedure MERGE( $A, p, q, r$ ) ▷ A arreglo
2:    $n_1 \leftarrow q - p + 1$ 
3:    $n_2 \leftarrow r - q$ 
4:   Sean  $L[1..n_1 + 1]$  y  $R[1..n_2 + 1]$  nuevos arreglos
5:   for  $i \leftarrow 1$  to  $n_1$  do
6:      $L[i] \leftarrow A[p + i - 1]$ 
7:   for  $j \leftarrow 1$  to  $n_2$  do
8:      $R[j] \leftarrow A[q + j]$ 
9:    $L[n_1 + 1] \leftarrow \infty$ 
10:   $R[n_2 + 1] \leftarrow \infty$ 
11:   $i \leftarrow 1$ 
12:   $j \leftarrow 1$ 
13:  for  $k \leftarrow p$  to  $r$  do
14:    if  $L[i] \leq R[j]$  then
15:       $A[k] \leftarrow L[i]$ 
16:       $i \leftarrow i + 1$ 
17:    else
18:       $A[k] \leftarrow R[j]$ 
19:       $j \leftarrow j + 1$ 

```

**Algoritmo 8** Ordenación rápida

```

1: procedure QUICKSORT( $A$ )
2:   if  $n > 1$  then
3:     Elegir como pivote al elemento  $A[p]$ 
4:      $r \leftarrow \text{PARTITION}(A, p)$ 
5:     QUICKSORT( $A[1 \dots r - 1]$ )
6:     QUICKSORT( $A[r + 1 \dots n]$ )

```

Algoritmo 9 Partición Lomuto

```

1: function LOMUTO-PARTITION( $A, p$ )  $\triangleright A[1 \dots n]$ 
2:    $A[p] \leftrightarrow A[n]$ 
3:    $l \leftarrow 0$ 
4:   for  $i \leftarrow 1$  to  $n - 1$  do
5:     if  $A[i] < A[n]$  then
6:        $l \leftarrow l + 1$ 
7:        $A[l] \leftrightarrow A[i]$ 
8:    $A[n] \leftrightarrow A[l + 1]$ 
9:   return  $l + 1$ 

```

Algoritmo 10 Partición Hoare

```

1: function HOARE-PARTITION( $A, p, r$ )  $\triangleright A[1 \dots n]$ 
2:    $x \leftarrow A[p]$ 
3:    $i \leftarrow p - 1$ 
4:    $j \leftarrow r + 1$ 
5:   while  $\text{TRUE}$  do
6:     repeat
7:        $j \leftarrow j - 1$ 
8:     until  $A[j] \leq x$ 
9:     repeat
10:     $i \leftarrow i + 1$ 
11:  until  $A[i] \geq x$ 
12:  if  $i < j$  then
13:    SWAP( $A[i], A[j]$ )
14:  else
15:    return  $j$ 

```

Ejercicios

- 6.2.1** La sucesión de Fibonacci es la sucesión de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Definida recursivamente como: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$. Se pide crear la función recursiva que determine F_n .

- 6.2.2** Dada la función recursiva, implementar la función:

$$gcd(n, m) = \begin{cases} n & \text{si } m = 0 \\ gcd(m, resto(n/m)) & \text{si } n \geq m \text{ y } m > 0 \end{cases}$$

6.2.3 Implementar la función para calcular el factorial de n .

6.2.4 Crear la función para determinar: x^n .

6.2.5 Aplicar la siguiente implementación de un algoritmo recursivo a beta(23):

```
int beta(int n)
{
    if( n == 0)
        return n;
    return((n % 2)+ 10*beta(n/2));
}
```

SECCIÓN 7

Cuestiones y problemas

7.1 Una palabra se llama *palíndroma* si la sucesión de sus letras no varía al invertir el orden. Especifique e implemente una función recursiva que decida si una palabra dada, representada como vector(arreglo) de caracteres, es o no palíndroma. Ej: La ruta natural.

7.2 La función de Ackermann se define recursivamente como sigue:

$$Ack(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ Ack(m - 1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ Ack(m - 1, Ack(m, n - 1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Se pide hacer el programa en lenguaje C y determinar **Ack(4,3)**.

7.3 Aplicar al siguiente conjunto $S = \{15, 67, 08, 16, 44, 27, 12, 35\}$, ordenamiento por intercambio y por selección. Justifique.

7.4 Modificar el método de inserción de manera que use búsqueda binaria para localizar dónde agregar el siguiente elemento. Se pide diseñar el nuevo algoritmo.

7.5 Sea $A = a_1a_2\dots a_n$ un arreglo desordenado de tamaño n , donde los elementos $a_i \in [0 \dots 9]$, para $i \in [1 \dots n]$. Se pide diseñar un algoritmo que permita determinar la repetición de cada elemento en el arreglo. Notar que algún elemento del rango puede no estar.

7.6 Hacer la implementación de un algoritmo recursivo que determine la cantidad de dígitos de un entero dado. El prototipo de la función es: `int printDig(int num)`. Un ejemplo de como ocupar dicha función es:

```
1 printf("%d\n",printDig(1234));
2 printf("%d\n",printDig(147734));
```

<pre>1 printf("%d\n",printDig(1234)); 2 printf("%d\n",printDig(147734));</pre>	Output: 4 6
--	-------------------

7.7 Aplicar la siguiente implementación de un algoritmo recursivo:

```
int pinta(int i, int d, int n)
{
    if(n < 1)
```

```

    return;

    int m = (i+d)/2;

    trazo(m,15,m,15-n*5);
    pinta(i,m-1,n-1);
    pinta(m+1,d,n-1);

}

```

Debe pensar que **trazo(x_1, y_1, x_2, y_2)**, **dibuja** una línea desde la coordenada cartesiana (x_1, y_1) a (x_2, y_2) . Asuma que la primera llamada a la función es: **pinta(1, 7, 3)**.

- 7.8** Aplicar al siguiente conjunto $S = \{8, 4, 9, 13, 5, 2\}$, ordenamiento por inserción y por selección.

- 7.9** Diseñar el TAD Cadena como una secuencia de caracteres, de la siguiente manera:

```

TAD Cadena(VALORES: Caracter;
            OPERACIONES: CadenaVacia, Poner, EsVacia, Longitud,
                        Concatenar, BorrarCar, CaracterN)

```

Donde:

CadenaVacia Crea la cadena vacía

Poner Añade un carácter al final de la cadena

EsVacia Decide si una cadena está vacía

Longitud Da la longitud de una cadena

Concatenar Devuelve una cadena resultado de la concatenación de otras dos

BorrarCar Elimina todas las apariciones del carácter

CaracterN Da el carácter que se encuentra en la posición indicada

Indique (**en palabras**) como sería la implementación de **Poner** y **BorrarCar**.

- 7.10** En los algoritmos **Burbuja** o **Selección**, el elemento más pequeño de $a[i \dots n]$ es colocado en la posición i mediante intercambios, para valores sucesivos de i . Otra posibilidad es colocar el elemento máximo de $a[1 \dots j]$ en la posición j , para valores de j entre n y 1. A este algoritmo se le denomina ordenación por Ladrillos (*bricksort*). Diseñar dicho algoritmo.

- 7.11** Se desea hacer un programa en C que permita hacer el intercambio de valores de 2 variables, por ejemplo $A=4$ y $B=5$. La idea de su implementación es que al final del programa se tenga $A=5$ y $B=4$. Obviamente su programa debe servir para cualquier caso. **NO se puede** usar una variable auxiliar, ni llamadas a alguna función. También puede indicar que dicho programa no se puede hacer, indicando las razones. (Para hacerlo más sencillo, suponga que A y B son de tipo **int**).

- 7.12** Escriba una función eficiente para determinar si existe un entero i tal que $A_i = i$ en un vector A de enteros ordenados de forma creciente.

7.13 Analice el siguiente algoritmo:

```
void sort ( int a[], int i, int j )
{
    if ( a[i] > a[j] )
    {
        intercambiar a[i] y a[j]
    }
    if (j ≤ i + 1 ){
        return;
    }
    int k = ⌊ 1/3 * ( j - i + 1 ) ⌋;

    sort (A,i,j-k);
    sort (A,i+k,j);
    sort (A,i,j-k);
}
```

Suponga que $A = \{15, 8, 23, 72, 45, 4\}$, aplique $\text{sort}(A, 1, n)$. Muestre los pasos mas relevantes.

7.14 Implemente una función eficiente que dado un arreglo de enteros positivos a y un número K , devuelva *verdadero* en caso de existir un subconjunto consecutivo de elementos que sumen exactamente K y *falso* en caso contrario.

7.15 Aplicar al siguiente conjunto $B=\{8,4,2,7,9,12,-45,3,-46,5,34,23,10,0,33,-10,-8,6\}$ los métodos de ordenación vistos en clases. Entre los índices $[0..7]$ con el método burbuja, en el rango $[8..13]$ con el método inserción, y el rango sobrante con el método selección. Indique las iteraciones obtenidas.

7.16 Suponga que se tiene la siguiente declaración:

```
struct datos{           struct atleta{           struct atleta ats[30];
    char nombre[40];       char deporte[30];
    char pais[25];         struct datos pers;
};                      int nmedallas;
};
```

Se pide implementar una función que determine el nombre y número de medallas del atleta que tenga más medallas.

7.17 Usar QUICKSORT para ordenar $B=\{8,4,2,7,9,12,-45,3,-46,5,34,23,10,0,33,-10,-8,6\}$.

7.18 Utiliza MERGESORT para ordenar $B=\{8,4,2,7,9,12,-45,3,-46,5,34,23,10,0,33,-10,-8,6\}$.

7.19 Implementar la solución al problema de las Torres de Hanoi con n discos y 3 varillas o pértigas.

7.20 Se tiene el siguiente programa. Se pide indicar cuál es el resultado de la impresión.

```
#include <stdio.h>
int main(void){
    int a,b,c,*p1,*p2;

    p1 = &a;
    *p1 = 1;
    p2 = &b;
    b = 2;
    p1 = p2;
    *p1 = 0;
```

```

p2 = &c;
*p2 = 3;
printf("%d %d %d\n",a,b,c);
return 0;
}

```

- 7.21** Implementa una función recursiva que calcule el MCD de dos números enteros utilizando el algoritmo de Euclides. La función debe recibir dos enteros como parámetros y devolver su máximo común divisor.
- 7.22** Escribe una función que reciba un puntero a un arreglo de enteros y su tamaño, y cuente cuántos de los elementos del arreglo son números pares. La función debe devolver el total de números pares encontrados.
- 7.23** Implementa una función que inserte un valor en una tabla hash utilizando la técnica de encadenamiento para manejar colisiones. La función debe recibir el valor a insertar y la tabla hash, calcular su posición en la tabla y agregarlo a la lista enlazada correspondiente en caso de colisión.
- 7.24** Implementa una función que reciba dos punteros a arreglos y un tamaño, y copie los elementos del primer arreglo en el segundo. Asegúrate de que ambos arreglos tengan suficiente espacio asignado y que la copia se realice correctamente sin pérdida de datos.
- 7.25** Escribe una función recursiva que tome un número entero como entrada y devuelva la cantidad de dígitos que contiene. Asegúrate de que tu función maneje correctamente tanto números positivos como negativos.
- 7.26** Implementa una función que reciba un arreglo de enteros y su tamaño, y lo rote hacia la izquierda una posición. La rotación debe ser circular, es decir, el primer elemento debe moverse al final del arreglo.
- 7.27** Escribe una función que reciba un puntero a un arreglo y devuelva su tamaño. La función debe calcular el tamaño del arreglo utilizando la aritmética de punteros, considerando el tamaño de cada elemento y el total de elementos.
- 7.28** Implementa una función que reciba una tabla hash, su tamaño actual, y un nuevo tamaño. La función debe redimensionar la tabla para adaptarse a la nueva capacidad si la carga supera un umbral definido (por ejemplo, 0.75). La función debe redistribuir correctamente todos los elementos en la nueva tabla.
- 7.29** Implementa una función que reciba dos punteros a cadenas de caracteres y compare su contenido carácter por carácter. La función debe devolver 0 si las cadenas son iguales, un valor positivo si la primera es mayor, y un valor negativo si es menor.
- 7.30** Implementa una función que reciba una matriz cuadrada de tamaño $n \times n$ y la invierta en su lugar. La función debe cambiar las filas por columnas sin utilizar una matriz auxiliar, es decir, realizando la inversión directamente en la matriz original.
- 7.31** Escribe una función que reciba un arreglo de enteros y devuelva el índice del primer número negativo en el arreglo. Si no hay números negativos en el arreglo, la función debe devolver -1.
- 7.32** Crea una función hash que reciba una clave numérica y un tamaño de tabla, y devuelva la posición en la tabla calculando el módulo de la clave con el tamaño de la tabla. Este tipo de función es comúnmente utilizada en implementaciones básicas de tablas hash.

- 7.33** Escribe una función que reciba una matriz 2D creada dinámicamente, su tamaño (número de filas y columnas), y libere la memoria asignada para evitar fugas de memoria. Asegúrate de liberar primero cada fila y luego la matriz completa.
- 7.34** Crea una función que reciba un valor y lo busque en una tabla hash. La función debe calcular la posición del valor utilizando la función hash y devolver la posición en la tabla. Si el valor no se encuentra, la función debe devolver -1.
- 7.35** Implementa una función recursiva que reciba un arreglo ordenado y un valor a buscar, y devuelva la posición del valor en el arreglo utilizando la técnica de búsqueda binaria. La función debe ser eficiente y manejar correctamente los casos en los que el valor no se encuentra en el arreglo.
- 7.36** Implementa una función que reciba dos tablas hash y determine si son idénticas en contenido y estructura. La función debe verificar que ambas tablas contengan los mismos elementos en las mismas posiciones y devolver verdadero si son iguales o falso en caso contrario.
- 7.37** Implementa una función recursiva que determine si una cadena de caracteres es un palíndromo. La función debe comparar el primer y último carácter de la cadena, y llamar a sí misma para los caracteres internos. La función debe devolver verdadero si la cadena es un palíndromo y falso si no lo es.
- 7.38** Crea un Tipo Abstracto de Datos (TAD) que maneje un diccionario, donde cada entrada sea un par clave-valor. El TAD debe permitir agregar, eliminar, buscar, y actualizar pares clave-valor de manera eficiente, asegurando que no haya claves duplicadas.
- 7.39** Escribe una función que calcule y devuelva la carga de una tabla hash. La carga se define como el número de elementos almacenados en la tabla dividido por el tamaño de la tabla. Esta métrica es importante para evaluar la eficiencia de la tabla hash y determinar si es necesario redimensionarla.
- 7.40** Implementa una función que reciba un arreglo de enteros y su tamaño, y elimine todos los elementos duplicados. La función debe modificar el arreglo original y devolver el nuevo tamaño del arreglo después de eliminar los duplicados.
- 7.41** Escribe una función que reciba dos arreglos de enteros y sus tamaños, y determine si ambos arreglos son idénticos en contenido y longitud. La función debe comparar elemento por elemento y devolver verdadero si los arreglos son iguales o falso si difieren en algún aspecto.
- 7.42** Crea una función recursiva que reciba un puntero a una cadena de caracteres y la invierta en su lugar. La función debe utilizar recursividad en lugar de bucles para invertir la cadena y no debe usar estructuras de datos auxiliares.
- 7.43** Escribe una función que reciba un puntero a un arreglo y su tamaño, y lo invierta en su lugar, es decir, sin usar un arreglo auxiliar. La función debe invertir los elementos del arreglo directamente utilizando punteros para intercambiar los valores.
- 7.44** Implementa una función que reciba dos punteros a cadenas de caracteres y compare su contenido carácter por carácter. La función debe devolver 0 si las cadenas son iguales, un valor positivo si la primera es mayor, y un valor negativo si es menor.
- 7.45** Escribe una función que reciba dos arreglos de enteros ya ordenados y los combine en un tercer arreglo que también esté ordenado. La función debe asegurar que el arreglo resultante mantenga el orden ascendente de los elementos.

TAD Lineales

SECCIÓN 8

Tipo de Dato Abstracto

Una abstracción es una vista o representación de una entidad que incluye solo los atributos más significativos. En un sentido general, la abstracción permite recopilar instancias de entidades en grupos en los que no es necesario considerar sus atributos comunes. En el mundo de los lenguajes de programación, la abstracción es un recurso contra la complejidad de la programación; su propósito es simplificar el proceso de programación. Es un recurso eficaz porque permite a los programadores centrarse en los atributos esenciales, ignorando los atributos subordinados.

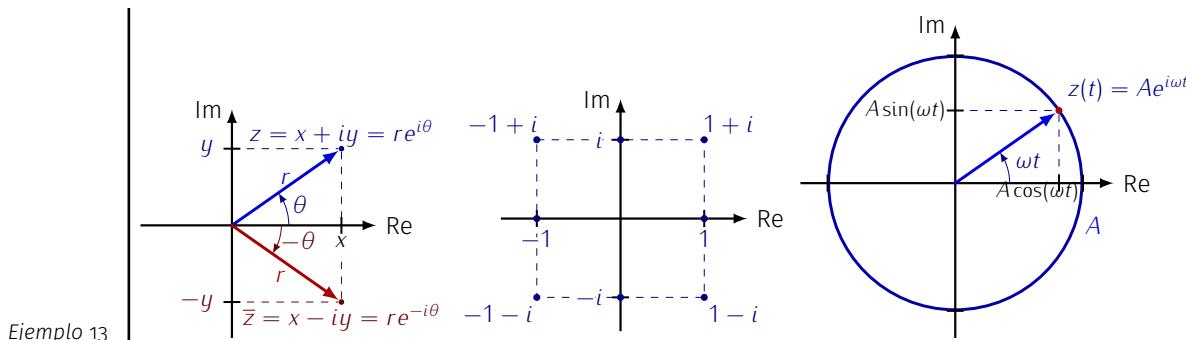
Los dos tipos fundamentales de abstracción en los lenguajes de programación contemporáneos son la abstracción de procesos y la abstracción de datos.

El tipo de datos abstractos (ADT o TDA o TAD) es un tipo cuyo comportamiento está definido por un conjunto de valores y un conjunto de operaciones. La definición de TAD solo menciona qué operaciones se realizarán, pero no cómo se implementarán estas operaciones. No especifica cómo se organizarán los datos en la memoria y qué algoritmos se utilizarán para implementar las operaciones. Se llama "abstracto" porque ofrece una vista independiente de la implementación.

Estas son algunas de las ventajas de TAD:

- Confiabilidad: al ocultar las representaciones de datos, el código de usuario no puede acceder directamente a los objetos del tipo o depender de la representación, lo que permite cambiar la representación sin afectar el código de usuario
- Reduce el rango de código y variables que el programador debe conocer
- Los conflictos de nombres son menos probables
- Proporciona un método de organización del programa
- Ayuda a la modificabilidad (todo lo asociado con una estructura de datos está junto)
- Compilación separada

Complejo



```

1 // Complex.h
2 #include <stdio.h>
3
4 typedef struct _complex Complex;
5 struct _complex {
6     float real;
7     float imag;
8 };
9
10 //struct _complex addComplex(struct _complex A,
11 //                             struct _complex B);
12 Complex setComplex(float R, float I);
13 float getReal(Complex A);
14 float getImag(Complex A);
15 Complex addComplex(Complex A, Complex B);
16 Complex divComplex(Complex A, Complex B);
17 Complex conjComplex(Complex A);
18
19 void printComplex(Complex A);

```

```

1 // Complex.c
2 #include "Complex.h"
3
4 Complex setComplex(float R, float I) {
5     Complex C;
6
7     C.real = R;
8     C.imag = I;
9
10    return C;
11 }
12
13 float getReal(Complex A){
14     return A.real;
15 }
16
17 float getImag(Complex A){
18     return A.imag;
19 }
20
21 Complex addComplex(Complex A, Complex B) {
22     Complex C;
23
24     C.real = A.real + B.real;
25     C.imag = A.imag + B.imag;
26
27     return C;
28 }
29
30 Complex divComplex(Complex A, Complex B){
31     Complex C;
32     float d;
33
34     d = (B.real*B.real)+(B.imag*B.imag);
35     C.real =((A.real*B.real)+(A.imag*B.imag))/d;
36     C.imag =((A.imag*B.real)-(A.real*B.imag))/d;
37
38     return C;
39 }
40
41 void printComplex(Complex A){
42     printf("%f\n",A.real);
43     printf("%f\n",A.imag);
44 }

```

$$\{z, \bar{z}\} \in \mathbb{C}, z = a+ib, \bar{z} = a-ib, \Re(z) = a, \Im(z) = b, i^2 = -1$$

$$z_1 \pm z_2 = (a+ib) \pm (c+id) = (a \pm c) + i(b \pm d)$$

$$z_1 \times z_2 = (a+ib) \times (c+id) = (ac-bd) + i(ad+bc)$$

$$\frac{z_1}{z_2} = \frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i \frac{bc-ad}{c^2+d^2}$$

```

1 // testcomplex.c
2 #include "Complex.h"
3
4 int main() {
5     Complex A, B, C;
6
7     A = setComplex(3, 7);
8     B = setComplex(8, 9);
9
10    // C = AddComplex(A,B);
11    C = divComplex(A, B); // ---> A/B
12
13    // printf("Parte real: %f,
14    //         parte img: %f\n",getReal(C),getImag(C));
15    printf("Parte real: %f\n", getReal(addComplex(
16                                setComplex(3, 7),
17                                setComplex(8, 9))));
18
19    printf("I: %f\n", getImag(C));
20
21    return 0;
22 }

```

```

# Makefile
# La siguiente no es necesariamente requerida,
# se agrega para mostrar como funciona.

.SUFFIXES: .o .c
.c.o:
    $(CC) -c $(CFLAGS) $<

# Macros
CC = gcc
CFLAGS = -g -Wall -O2

#SRC = Complex.c testcomplex.c
OBJ = Complex.o testcomplex.o

# Reglas explícitas
all: $(OBJ)
    $(CC) $(CFLAGS) -o testcomplex $(OBJ)

clean:
    $(RM) $(OBJ) testcomplex

# Reglas implícitas
#Complex.o: Complex.c Complex.h
#testcomplex.o: testcomplex.c Complex.h

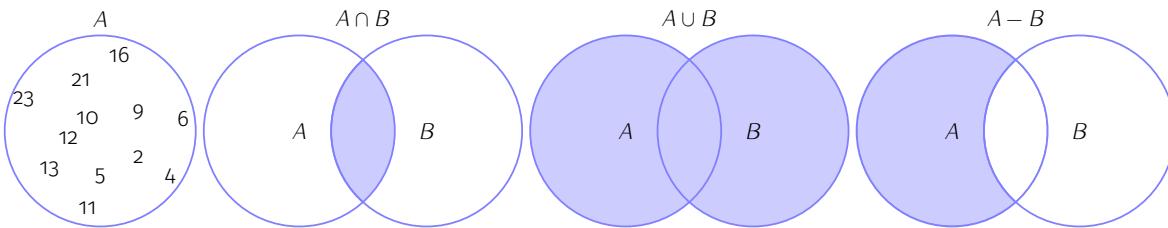
```

```

Output:
make
gcc -c -g -Wall -O2 Complex.c
gcc -c -g -Wall -O2 testcomplex.c
gcc -g -Wall -O2 -o testcomplex Complex.o testcomplex.o
./testcomplex
Parte real: 11.000000
I: 0.200000

```

Conjunto



Ejemplo 14

```

1 typedef struct setElems Set;
2 struct setElems{
3     int NElem;
4     Elemt L[MAXELEM];
5 };
6
7     int inSet(Set C, Elemt x);
8     void addElemSet(Set *C, Elemt e);
9     void printSet(Set C);

```

Sean A,B conjuntos.

A = {35,20,10,45,15,40,30,5,25},
B = {12,15,9,3,30,20,24,21,6,18}.

$$A \cup B = \{35, 20, 10, 45, 15, 40, 30, 5, 25, 12, 9, 3, 24, 21, 6, 18\}$$

$$A \cap B = \{15, 20, 30\}$$

$$A - B = \{35, 10, 45, 40, 5, 25\}$$

¿Qué es un Diccionario ?

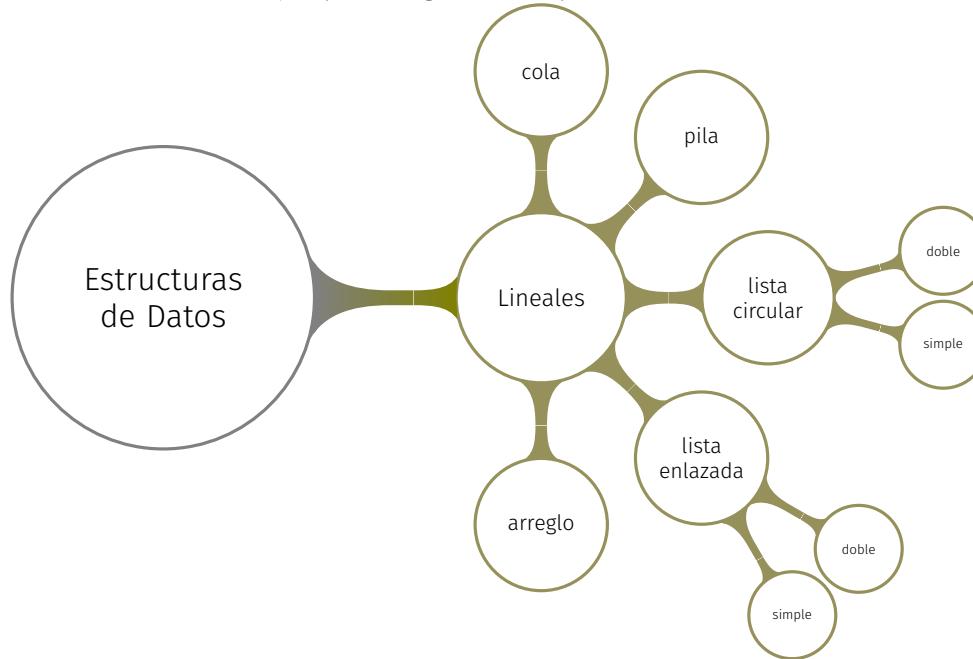
Dado un conjunto de elementos X_1, X_2, \dots, X_N , todos distintos entre sí, se desea almacenarlos en una estructura de datos que permita la implementación eficiente de las operaciones:

- Find(X): dado un elemento X, conocido como llave de búsqueda, encontrarlo dentro del conjunto o decir que no está.
- Insert(X): agregar un nuevo elemento X al conjunto.
- Delete(X): eliminar el elemento X del conjunto.

SECCIÓN 9

TAD Lineales

Se denomina TDA Lineales cuando los elementos de datos se organizan en orden secuencial, uno tras otro. Todos los elementos están presentes en una sola capa. Se puede recorrer en una sola carrera. Es decir, si comenzamos desde el primer elemento, podemos recorrer todos los elementos secuencialmente en una sola pasada. La utilización de la memoria no es eficiente. La complejidad del tiempo aumenta con el tamaño de los datos. Ejemplo: arreglos, listas, pila, cola



SUBSECCIÓN 9.1

Estructuras dinámicas

```

// ----- Set.h
typedef struct setElems Set;
struct setElems{
    int len;
    int max_len;
    int *data;
};

Set *makeSet(void);
Set *unionSet(Set A, Set B);
Set *interSet(Set A, Set B);
void printSet(Set C);
  
```

```

// ----- Set.c
Set *makeSet(void) {
    Set *s = malloc(sizeof(struct setElems));
    s->len = 0;
    s->max_len = 1;
    s->data = malloc(s->max_len * sizeof(int));
    return s;
}
  
```

SUBSECCIÓN 9.2

Lista enlazada simple

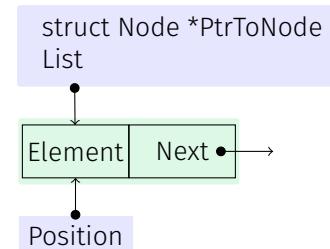
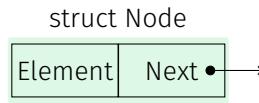
Una lista enlazada, en términos simples, es una colección lineal de elementos de datos. Estos elementos de datos son llamados *nodos*. La lista enlazada es una estructura de datos que a su vez se puede utilizar para implementar otras estructuras de datos. Por lo tanto, actúa como un bloque de construcción para implementar estructuras de datos

como pilas, colas, y sus variaciones. Una lista enlazada puede percibirse como un tren o una secuencia de nodos en los que cada nodo contiene uno o más campos de datos y un puntero al siguiente nodo.

```

1  typedef int ElementType;
2
3  typedef struct Node *PtrToNode;
4  typedef PtrToNode List;
5  typedef PtrToNode Position;
6
7  struct Node {
8      ElementType Element;
9      Position Next;
10 }

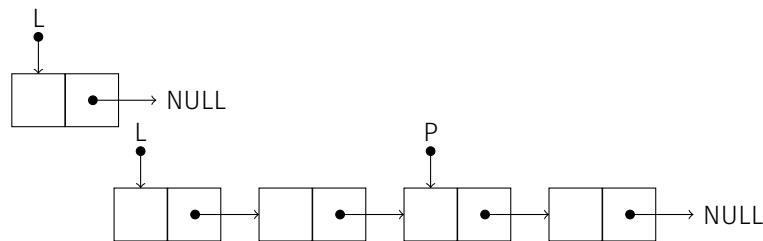
```



```

1 List MakeEmpty(List L) {
2     if (L != NULL)
3         DeleteList(L);
4
5     L = malloc(sizeof(struct Node));
6     if (L == NULL)
7         FatalError("Out of memory!");
8     L->Next = NULL;
9     return L;
10 }

```



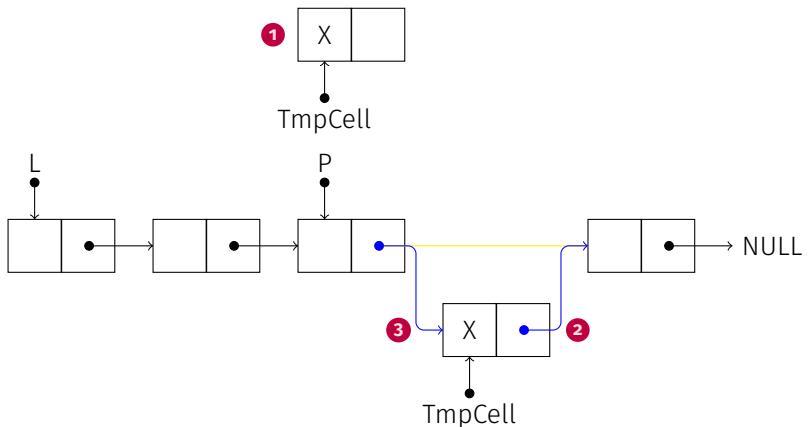
```

void Insert(ElementType X, List L, Position P) {
    Position TmpCell;

    TmpCell = malloc(sizeof(struct Node));
    if (TmpCell == NULL)
        FatalError("Out of space!!!");

    ① TmpCell->Element = X;
    ② TmpCell->Next = P->Next;
    ③ P->Next = TmpCell;
}

```



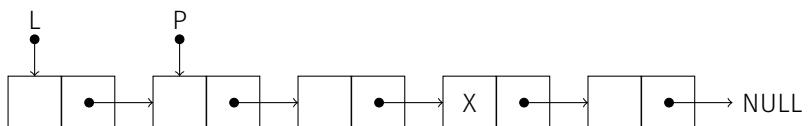
```

Position Find(ElementType X, List L) {
    Position P;

    P = L->Next;
    while (P != NULL && P->Element != X)
        P = P->Next;

    return P;
}

```



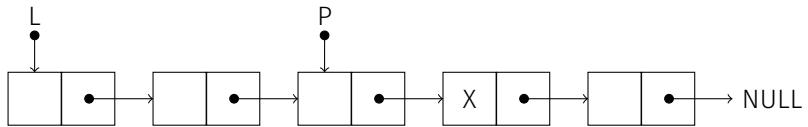
```

Position FindPrevious(ElementType X, List L) {
    Position P;

    P = L;
    while (P->Next != NULL && P->Next->Element != X)
        P = P->Next;

    return P;
}

```

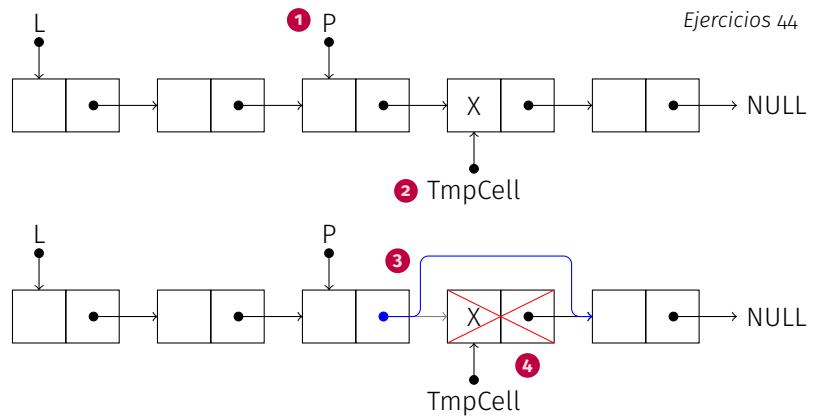


TAD LINEALES

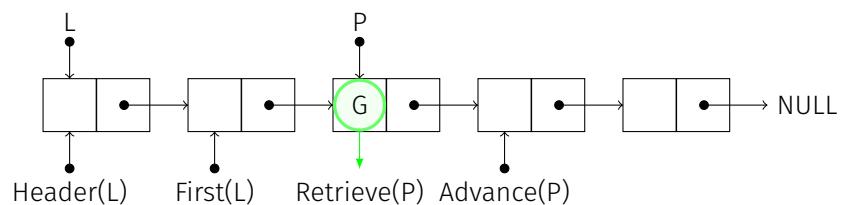
Ejercicios 44

```
void Delete(ElementType X, List L) {
    Position P, TmpCell;
```

```
① P = FindPrevious(X, L);
if (!IsLast(P, L)) /* Assumption of header use */
{
    ② TmpCell = P->Next;
    ③ P->Next = TmpCell->Next; // Bypass deleted cell
    ④ free(TmpCell);
}
```



```
1 Position Header(List L) {
2     return L;
3 }
4
5 Position First(List L) {
6     return L->Next;
7 }
8
9 Position Advance(Position P) {
10    return P->Next;
11 }
12
13 ElementType Retrieve(Position P) {
14     return P->Element;
15 }
```



```
1 int IsEmpty(List L) {
2     return L->Next == NULL;
3 }
4
5 int IsLast(Position P, List L) {
6     return P->Next == NULL;
7 }
```

Ejercicios

9.2.1 Crear una función que permita modificar el valor de un nodo.

9.2.2 Implementar función que permita invertir la lista.

9.2.3 Hacer la implementación `merge` de dos listas

9.2.4 Cree la función `insertLast(L,X)` que inserta el elemento x al final de la lista.

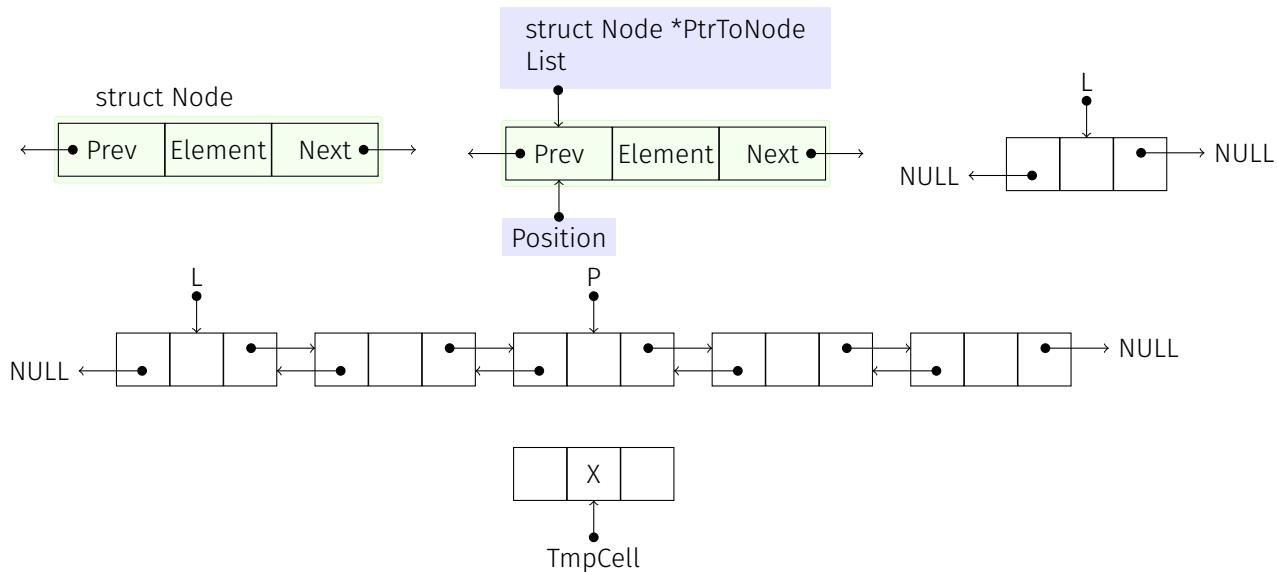
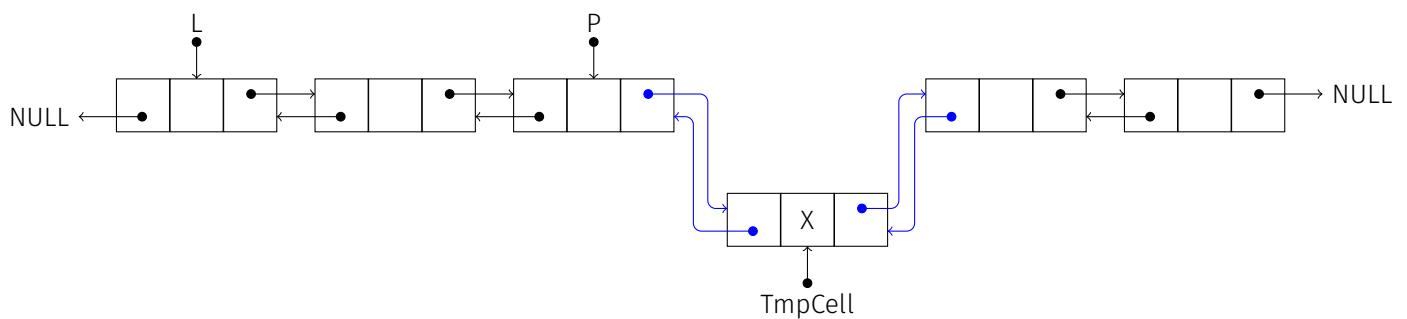
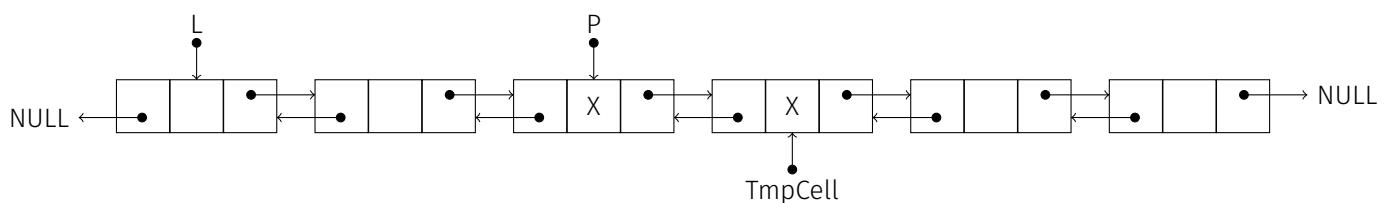
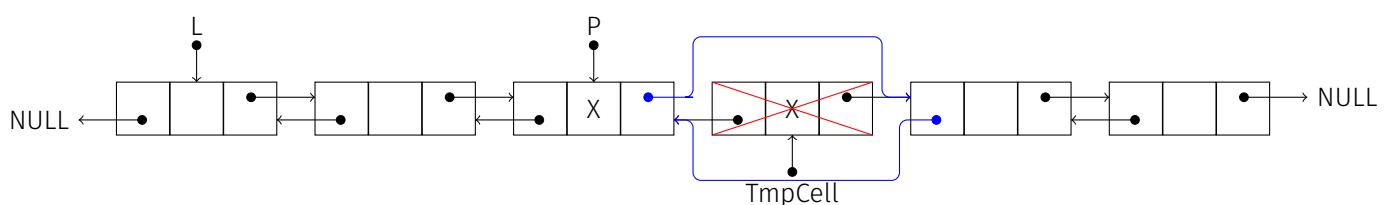
9.2.5 Implementar la función `deleteLast(L)` que borra el último elemento.

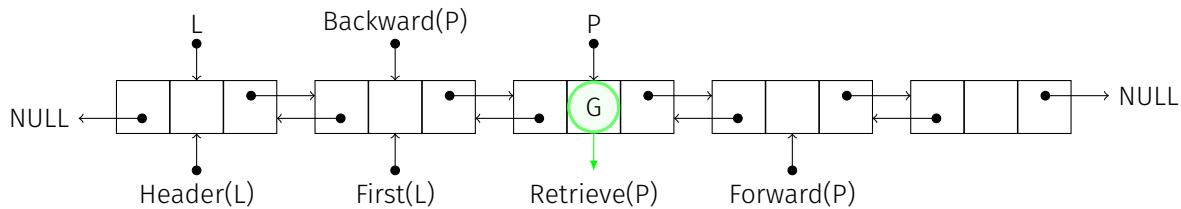
9.2.6 Crear la función `display(L)` que muestra los elementos de una lista.

9.2.7 Implementar el TAD Conjunto usando Lista Simplemente Enlazada.

SUBSECCIÓN 9.3

Listas dobles enlazadas

List**Insert****Find****Delete**



Ejercicios

- 9.3.1** Implementar todas las funciones que muestran las figuras.
- 9.3.2** Crear la función `DisplayList` que muestra los elementos de las lista.
- 9.3.3** Implementar `InvertDisplayList` que muestra los elementos de la lista pero en forma invertida.
- 9.3.4** Implementar `InsertLast` que permite insertar un elemento al final de la lista.
- 9.3.5** Crear la función `DeleteLast` que borra el último elemento de la lista.
- 9.3.6** Implementar el TAD Conjunto usando Lista Dblemente Enlazada.

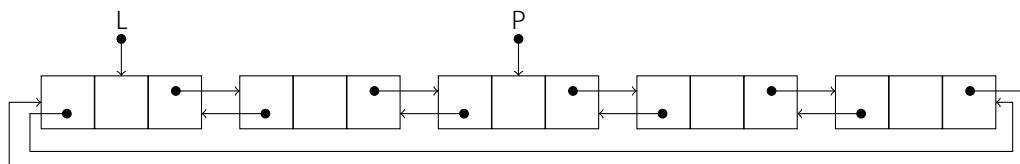
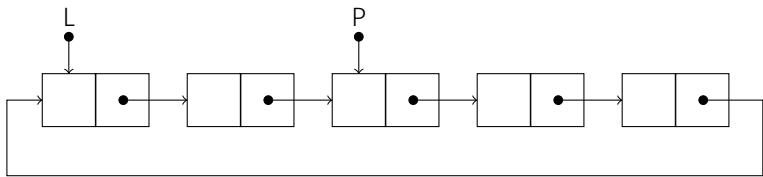
SUBSECCIÓN 9.4

Listas circulares

```

1 List MakeEmpty(List L) {
2     if (L != NULL)
3         DeleteList(L);
4
5     L = malloc(sizeof(struct Node));
6     if (L == NULL)
7         FatalError("Out of memory!");
8     L->Next = L;
9     return L;
10}

```



Ejercicios

- 9.4.1** Implementar la función `Insert` que permite insertar un elemento en una lista circular.
- 9.4.2** Crear la función `Delete` que borra un elemento en una lista circular.
- 9.4.3** Implementar la función `Find` que busca un elemento en una lista circular.
- 9.4.4** Implementar la función `RotateLeft` que permite hacer una rotación a la izquierda de la lista circular.
- 9.4.5** Implementar la función `RotateRight` que permite hacer una rotación a la derecha de la lista circular.
- 9.4.6** Implementar el TAD Conjunto usando Lista Circular.

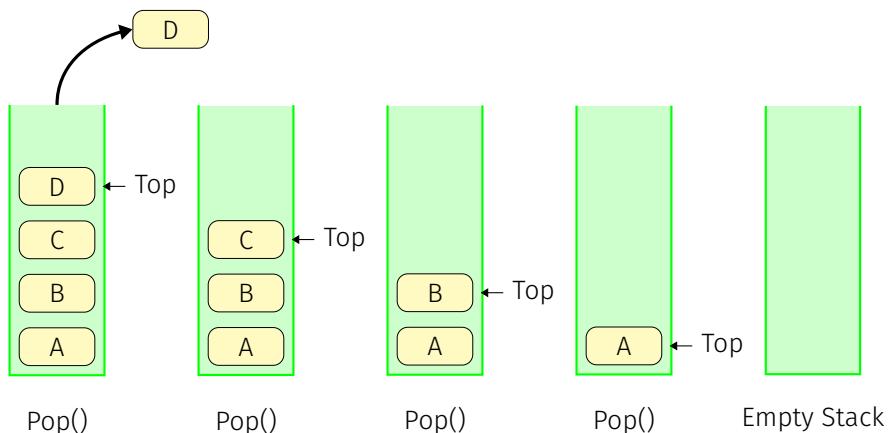
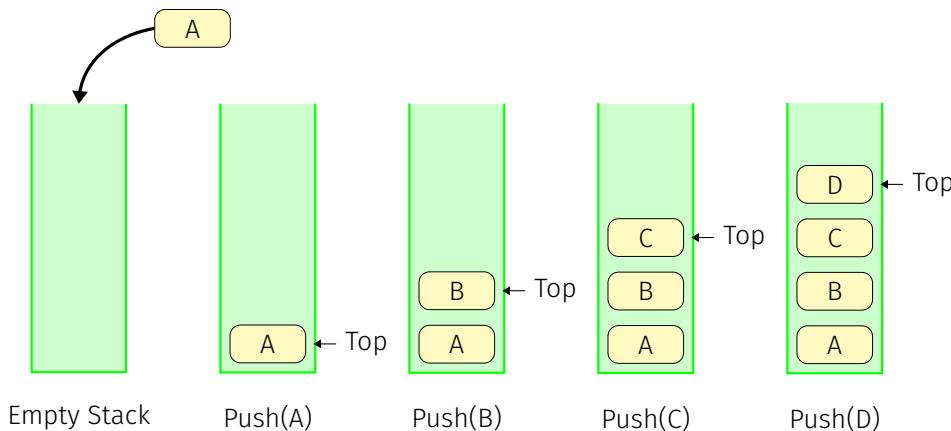
SECCIÓN 10

Pilas**LIFO: Last In First Out**

Estructura de datos lineal en la que los elementos se pueden insertar y eliminar solo desde un lado, llamado Top. Una pila(Stack) sigue el principio LIFO (último en entrar primero en salir). La inserción de un elemento en la pila se denomina operación Push y la eliminación de un elemento de la pila se denomina Pop.

Aplicaciones de la estructura de datos Pila

- algoritmos de seguimiento.
- funcionalidad de deshacer/rehacer en un software.
- análisis de sintaxis para muchos compiladores.
- verificar la apertura y el cierre correctos de los paréntesis.



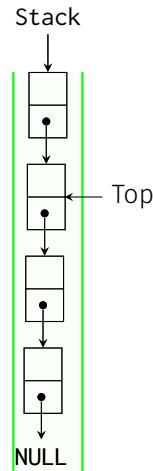
SUBSECCIÓN 10.1

Implementación con Lista Enlazada Simple

```

1 struct Node
2 {
3     ElementType Element;
4     PtrToNode Next;
5 };
6 typedef struct Node *PtrToNode;
7 typedef PtrToNode Stack;

```

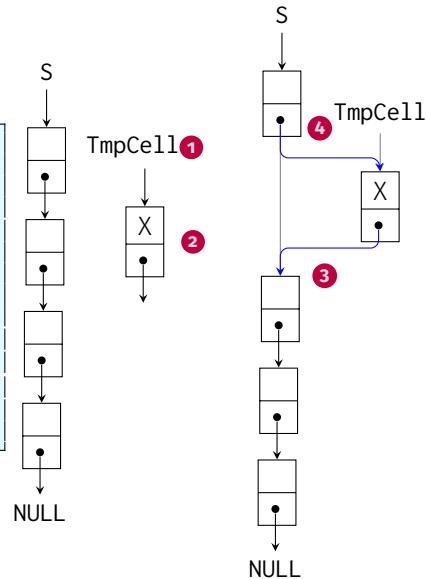


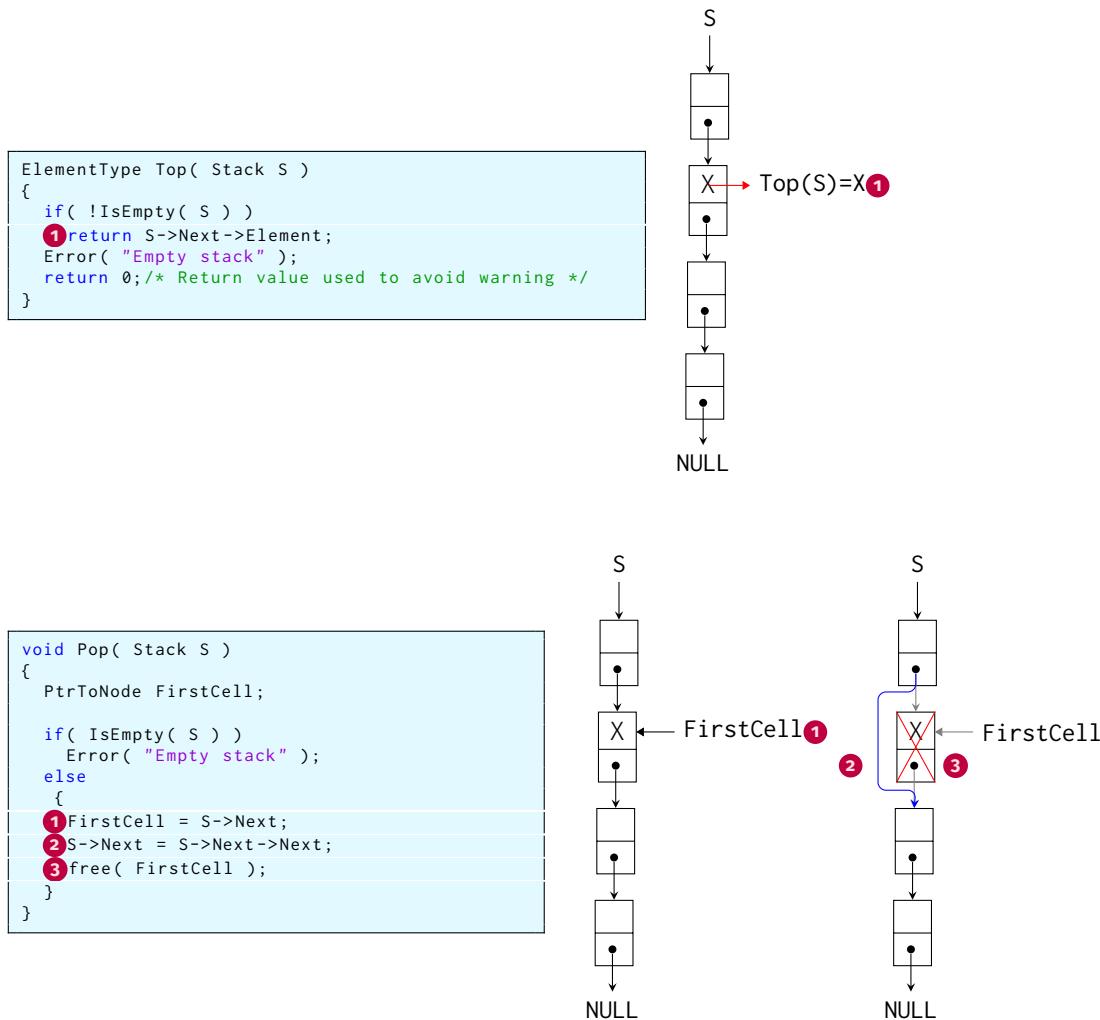
```

void Push( ElementType X, Stack S )
{
    PtrToNode TmpCell;

    ① TmpCell = malloc( sizeof( struct Node ) );
    if( TmpCell == NULL )
        FatalError( "Out of space!!!" );
    else
    {
        ② TmpCell->Element = X;
        ③ TmpCell->Next = S->Next;
        ④ S->Next = TmpCell;
    }
}

```



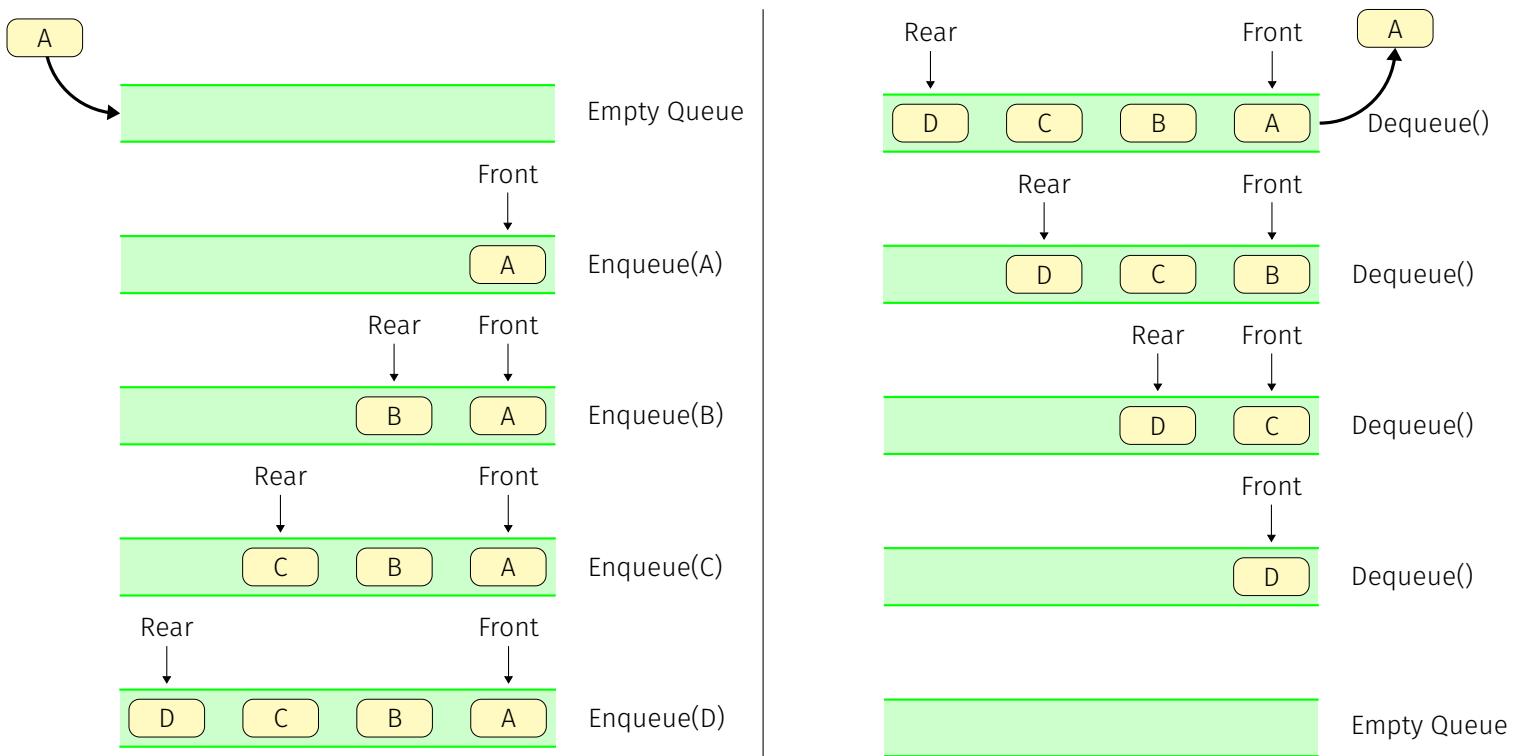


SECCIÓN 11

Colas**FIFO: First In First Out**

La cola es una estructura de datos lineal en la que los elementos se pueden insertar solo desde un lado de la lista llamado Rear y los elementos se pueden eliminar solo desde el otro lado llamado Front. La estructura de datos Cola(Queue) sigue el principio FIFO (primero en entrar, primero en salir), es decir, el elemento insertado primero en la lista es el primer elemento que se elimina de la lista. La inserción de un elemento en una cola se denomina Enqueue y la eliminación de un elemento se denomina Dequeue. Usos de la estructura de datos Colas:

- Programación(scheduling) de CPU, Programación(scheduling) de disco
- Cuando los datos se transfieren de forma asíncrona entre dos procesos.
- La cola se utiliza para la sincronización. Por ejemplo: IO Buffers, pipes, file IO, etc.
- Manejo de interrupciones en sistemas de tiempo real.
- Los sistemas telefónicos de Call Center usan colas para mantener en orden a las personas que los llaman.



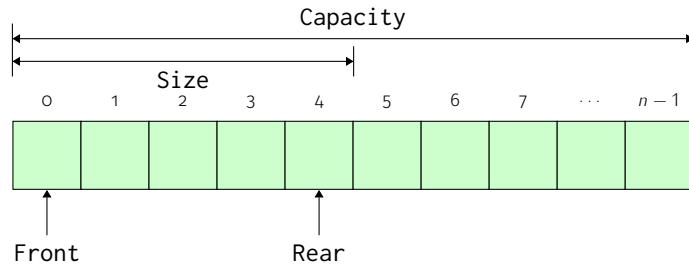
SUBSECCIÓN 11.1

Implementación con arreglo

```

1 #define MinQueueSize ( 5 )
2
3 struct QueueRecord
4 {
5     int Capacity;
6     int Front;
7     int Rear;
8     int Size;
9     ElementType *Array;
10 };
11 typedef struct QueueRecord *Queue;

```



```

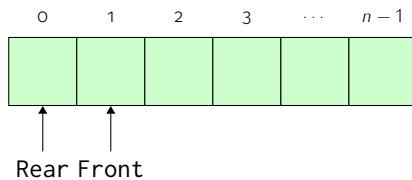
1 Queue CreateQueue( int MaxElements )
2 {
3     Queue Q;
4
5     if( MaxElements < MinQueueSize )
6         Error( "Queue size is too small" );
7
8     Q = malloc( sizeof( struct QueueRecord ) );
9     if( Q == NULL )
10        FatalError( "Out of space!!!" );
11
12     Q->Array = malloc( sizeof( ElementType ) * MaxElements );
13     if( Q->Array == NULL )
14        FatalError( "Out of space!!!" );
15     Q->Capacity = MaxElements;
16     MakeEmpty( Q );
17
18     return Q;
19 }

```

```

1 void MakeEmpty( Queue Q )
2 {
3     Q->Size = 0;
4     Q->Front = 1;
5     Q->Rear = 0;
6 }

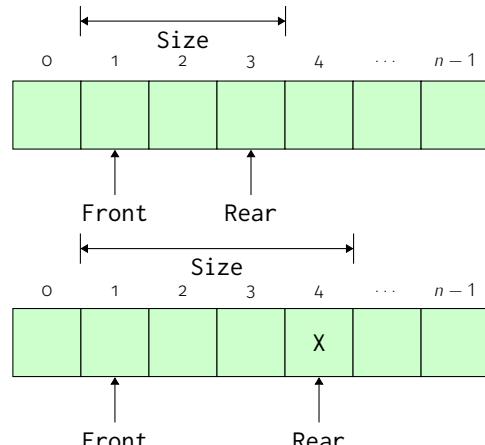
```



```

1 void Enqueue( ElementType X, Queue Q )
2 {
3     if( IsFull( Q ) )
4         Error( "Full queue" );
5     else
6     {
7         Q->Size++;
8         Q->Rear = Succ( Q->Rear, Q );
9         Q->Array[ Q->Rear ] = X;
10    }
11 }

```



```

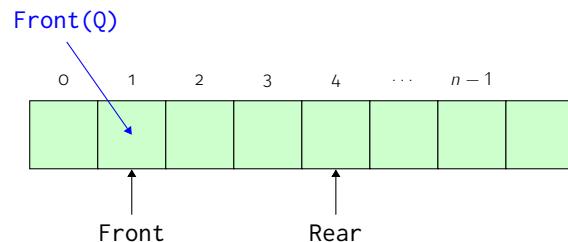
1 static int Succ( int Value, Queue Q )
2 {
3     if( ++Value == Q->Capacity )
4         Value = 0;
5     return Value;
6 }

```

```

1 ElementType Front( Queue Q )
2 {
3     if( !IsEmpty( Q ) )
4         return Q->Array[ Q->Front ];
5     Error( "Empty queue" );
6     return 0; /* Return value used to avoid warning */
7 }

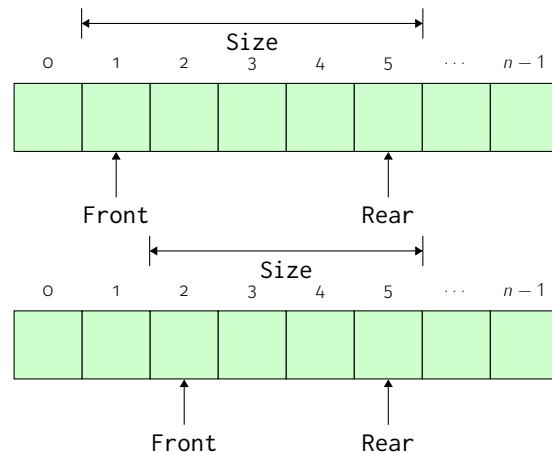
```

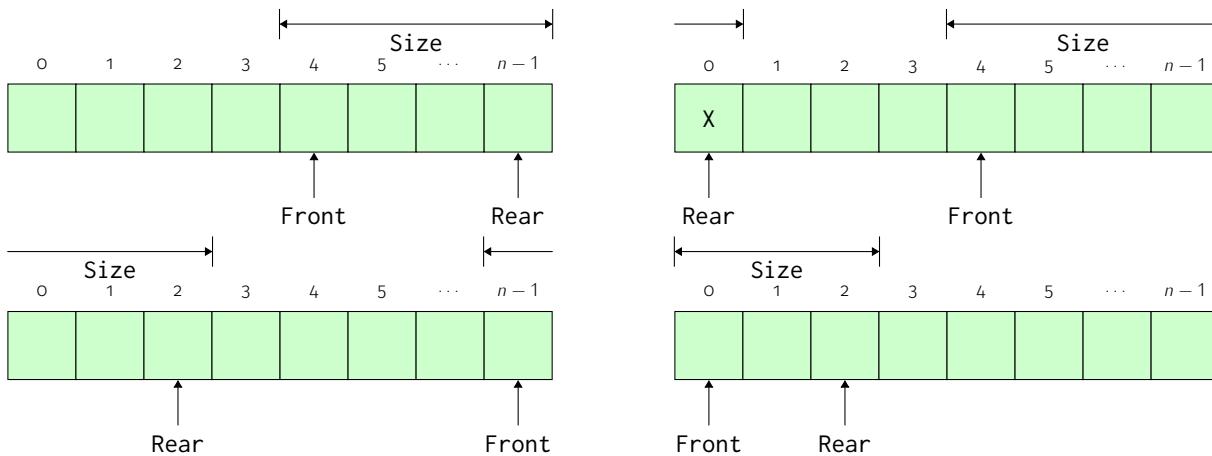


```

1 void Dequeue( Queue Q )
2 {
3     if( IsEmpty( Q ) )
4         Error( "Empty queue" );
5     else
6     {
7         Q->Size--;
8         Q->Front = Succ( Q->Front, Q );
9     }
10 }

```



Casos**Ejercicios**

-
- 11.1.1** Implementar una Pila usando lista circular
 - 11.1.2** Implementar una Cola usando lista circular
 - 11.1.3** Implementar una Pila usando arreglo
 - 11.1.4** Implementar una Cola usando lista enlazada

SECCIÓN 12

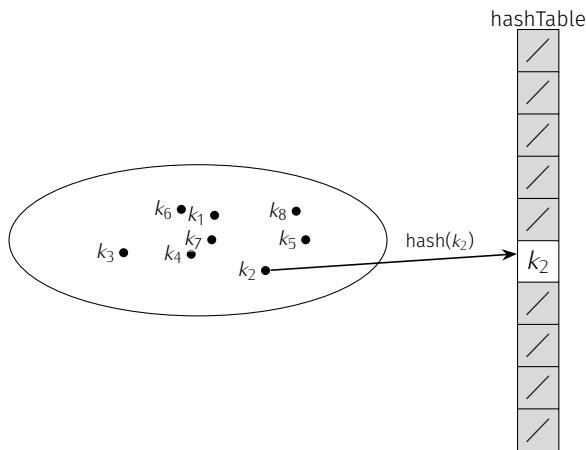
Hashing

"Hash" es realmente un término amplio con diferentes significados formales en diferentes contextos. Un *hash* es una función a la que se hace referencia como función hash que toma como objetos de entrada y genera una cadena o un número. Los objetos de entrada suelen ser miembros de tipos de datos básicos como cadenas, enteros o tipos más grandes compuestos por otros objetos como estructuras definidas por el usuario. La salida suele ser un número o una cadena. El sustantivo "hash" a menudo se refiere a esta salida.

Las principales propiedades que debe tener una función hash son:

- Debe ser fácil de calcular y
- las salidas deben ser relativamente pequeñas.

Ejemplo: Digamos que queremos hacer un hash de números en el rango de 0 a 999.999.999 a un número entre 0 y 99. Una función hash simple puede ser $h(x) = x \bmod 100$.

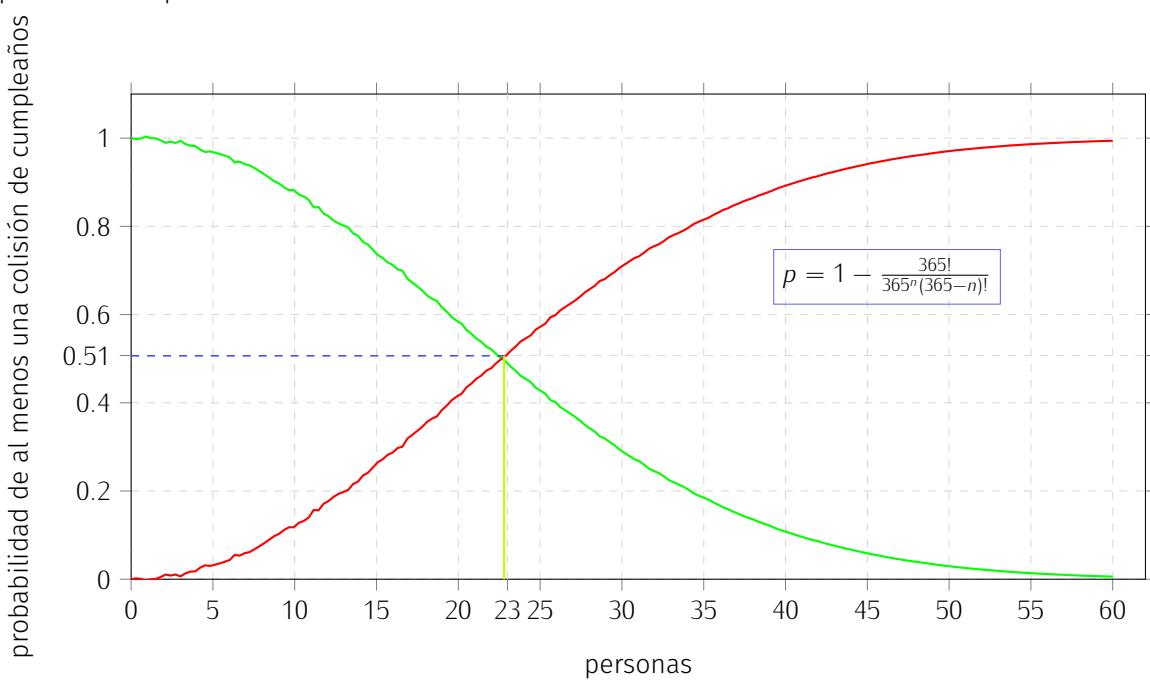


Algunas aplicaciones

Una aplicación común es en estructuras de datos como una tabla hash, que es una forma de implementar diccionarios.

Tenga en cuenta que también podría implementar diccionarios de otras formas.

La *Paradoja del Cumpleaños* es ¿sabría calcular la probabilidad de que al menos dos personas cumplan años el mismo día?



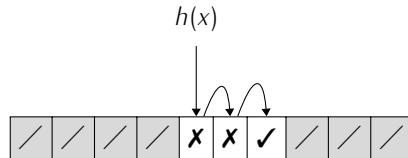
Las colisiones ocurren cuando una función hash mapea dos claves diferentes para el mismo lugar. En estos casos se aplica alguna técnica de resolución de colisiones. Los dos métodos más populares para resolver colisiones son: (a) direccionamiento abierto y (b) encadenamiento.

El direccionamiento abierto es una técnica de resolución de colisiones en la que todos los registros de entrada se almacenan en la propia matriz de cubos y la resolución de hash se realiza a través del sondeo(*probing*).

Las secuencias de sondeo más conocidas son:

linear probing: en el que se fija el intervalo entre sondeos (normalmente 1).

$$h_0(x) = h(x), h_{i+1} = (h_i(x) + 1) \bmod m$$



quadratic probing, en el que el intervalo entre sondeos aumenta al sumar las salidas sucesivas de un polinomio cuadrático al valor dado por el cálculo hash original.

doble hashing, en el que el intervalo entre sondeos se calcula mediante una función hash secundaria.

$$h_0(x) = h(x), h_{i+1} = (h_i(x) + s(x)) \bmod m$$

El rendimiento del direccionamiento abierto puede ser más lento en comparación con el encadenamiento separado, ya que la secuencia de sondeo aumenta cuando el factor de carga $\alpha = \frac{n}{m}$ se acerca a 1.

El sondeo da como resultado un ciclo infinito si el factor de carga llega a 1, en el caso de una tabla completamente llena.

Ejemplo 15 Sea $S=\{10,12,18,21,40,33,14,15,100\}$, aplique *hashing doble* para hacer la inserción y búsqueda.

Las funciones de hash:

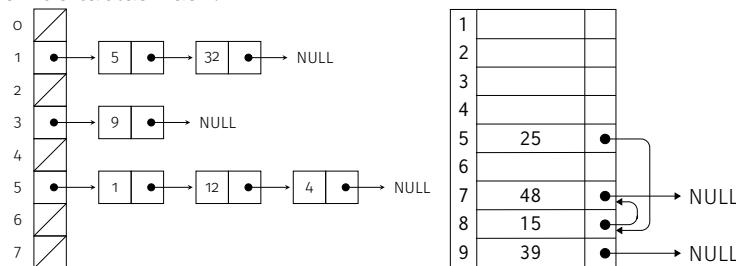
$$H(x) = x \bmod 10; h_0(x) = (x + 3) \bmod 10, h_i = (h_{i-1}(x) + 1) \bmod 10$$

0	10	$H(10) = 0$
1	21	$H(21) = 1$
2	12	$H(12) = 2$
3	40	$h_0(40) = 3, H(40) = 0$
4	14	$H(14) = 4$
5	15	$H(15) = 5$
6	33	$h_0(33) = 6, H(33) = 3$
7	100	$h_4(100) = 7, h_3(100) = 6, h_2(100) = 5, h_1(100) = 4, h_0(100) = 3, H(100) = 0$
8	18	$H(18) = 8$
9		

SUBSECCIÓN 12.2

Encadenamiento separado

En un encadenamiento separado, el proceso implica la creación de una lista enlazada con un par clave-valor para cada índice de matriz de búsqueda. Los elementos colisionados se encadenan a través de una sola lista enlazada, que se puede recorrer para acceder al elemento con una clave de búsqueda única. La resolución de colisiones a través del encadenamiento con lista enlazada es un método común de implementación de tablas hash.



Ejercicios

12.2.1 Inserte las llaves 7, 24, 18, 52, 36, 54, 11, y 23 en un hash encadenado. Use $h(k) = k \bmod m$. Donde $m = 9$. Inicialmente, la tabla de hash esta vacía.

12.2.2 Considere una tabla hash con tamaño 10. Usando *linear probing*, inserte las llaves 27, 72, 63, 42, 36, 18, 29, y 101 en la tabla.

12.2.3 Considere una tabla hash con tamaño 11. Usando *double hashing*, inserte las llaves 27, 72, 63, 42, 36, 18, 29, y 101 en la tabla. Tome $h_1 = k \bmod 10$ y $h_2 = k \bmod 8$.

SECCIÓN 13

Cuestiones y problemas

13.1 Se tienen 2 listas enlazadas, L_1 y L_2 , ordenadas. Se desea crear una función $\text{merge}(L_1, L_2)$ que realiza la mezcla de ambas listas en forma ordenada.

13.2 Dada una lista enlazada L , se pide implementar una función que devuelva una lista con los mismos elementos pero dispuestos en forma inversa.

13.3 Una *deque* es un tipo de dato abstracto que permite almacenar una lista de elementos y realizar operaciones de inserción y eliminación.

```
/* funciones de deque */
push(x) // inserta el elemento x al principio de la deque
pop() // saca el elemento que se encuentra al principio de la deque y lo retorna
inject(x) // inserta el elemento x al final de la deque
eject() // saca el elemento que se encuentre al final de la deque y lo retorna
isEmpty() // retorna true si la deque esta vacia
```

Se pide implementar las funciones *inject* y *eject*. Especifique la estructura de datos base.

13.4 Implemente el código en C para verificar el equilibrio de paréntesis ('(')') en una cadena de caracteres usando una pila. La cadena es el dato de entrada, no debe tomar en cuenta caracteres distintos a los paréntesis.

- 13.5** Escribir una función `sum(Stack S, int m)`, que reemplaza los primeros m elementos de la pila S por su suma. Si hay menos de m elementos en S entonces debe reemplazar todos por su suma. Ejemplo, si $S=\{1,3,2,5,4,3,2\}$, entonces $\text{sum}(S,4)$ debe dejar $S=\{11,4,3,2\}$. Si $S=\{1,3,2\}$ entonces $\text{sum}(S,4)$ debe dejar $S=\{6\}$.
- 13.6** Dada la implementación vista en clases de Lista Enlazada, se pide implementar el algoritmo de ordenamiento llamado de intercambio aplicado a una lista simplemente enlazada. La función debe estar definida como: `void Sort(List L);`
- 13.7** Un número natural enorme se puede representar como una lista circular doblemente enlazada, de forma que cada nodo contenga cuatro dígitos del número, excepto el que contiene los dígitos más a la izquierda, que podría tener menos. Diseñar una función que dado dos números naturales enormes calcule la suma de ellos, se debe tener en cuenta que el contenido de las listas no deben ser alterados.
- 13.8** Implementar la función `jph()` que dado un entero k , recorra los elementos eliminando el k -ésimo elemento, hasta dejar sólo uno. Ejemplo: Si la lista tiene los elementos 1,2,3,4,5 y $k=3$, la secuencia de elementos eliminados sería: 3,1,5,2 y el elemento sobreviviente sería el 4.
- 13.9** Dado el siguiente conjunto $\{2, 45, 34, 67, 5, 45, 78, 90\}$, muestre las secuencias de inserción en una tabla hashing.

- hashing encadenado:

$$h(x) = (3 + x) \bmod m$$

- hashing de direccionamiento simple:

$$h_0(x) = h(x)$$

$$h_{i+1} = (h_i(x) + 1) \bmod m$$

- hashing doble:

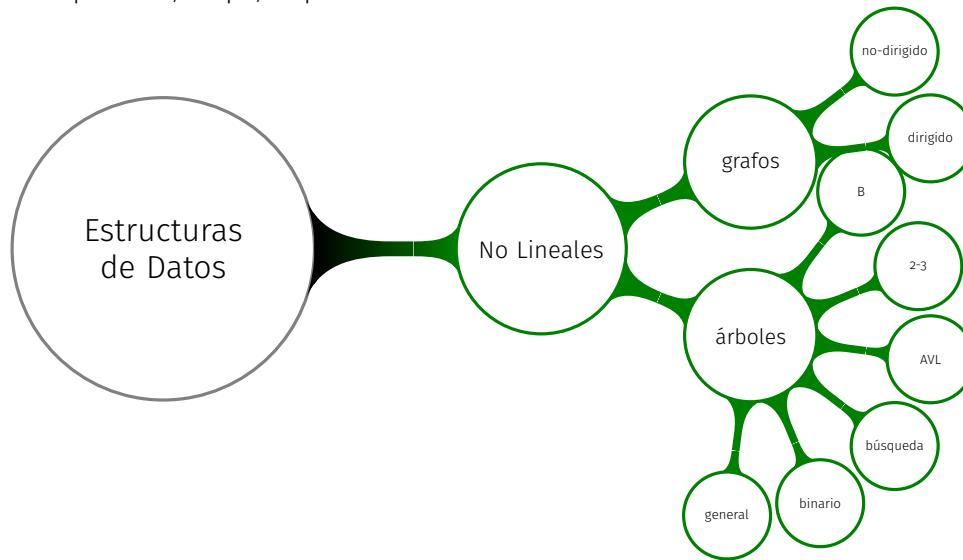
$$h_0(x) = h(x)$$

$$h_{i+1} = (h_i(x) + s(x)) \bmod m, s(x) = 2$$

TAD no Lineales

Los elementos de datos se organizan en orden no secuencial (manera jerárquica). Los elementos de datos están presentes en diferentes capas. Requiere múltiples ejecuciones. Es decir, si comenzamos desde el primer elemento, es posible que no sea posible recorrer todos los elementos en una sola pasada.

Example: Tree, Graph, Map



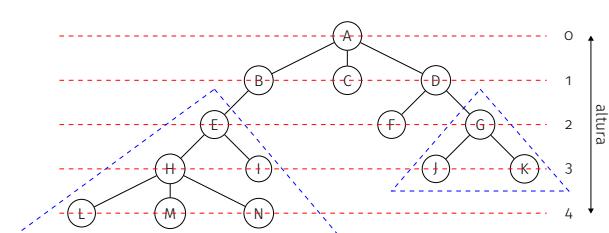
SECCIÓN 14

Árboles

Es una estructura de datos no lineal donde cada elemento individual se llama nodo. El nodo en un árbol almacena los datos reales de ese elemento en particular y se vincula al siguiente elemento en la estructura jerárquica mediante un enlace o arista.

SUBSECCIÓN 14.1

Árbol general



Terminología

1. Raíz: En una estructura de datos de árbol, el primer nodo se denomina nodo raíz. Cada árbol debe tener un nodo raíz. Podemos decir que el nodo raíz es el origen de la estructura de datos del árbol. En cualquier árbol, debe haber solo un nodo raíz. Nunca tenemos múltiples nodos raíz en un árbol.

2. Arista: El vínculo de conexión entre dos nodos se denomina arista. En un árbol con 'N' nodos habrá un máximo de 'N-1' aristas.

3. Padre: El nodo que es un predecesor de cualquier nodo se denomina nodo padre. Es

el nodo que tiene una rama desde él a cualquier otro nodo se denomina nodo principal. El nodo padre también se puede definir como "El nodo que tiene hijo/hijos".

4. Hijo: El nodo que es descendiente de cualquier nodo se denomina nodo hijo. El nodo que tiene un enlace desde su nodo principal se denomina nodo secundario. Cualquier nodo principal puede tener cualquier número de nodos secundarios. Todos los nodos, excepto el raíz, son nodos secundarios.

5. Hermanos: Los nodos con el mismo parente se denominan nodos hermanos.

6. Hoja: El nodo que no tiene hijos se denomina nodo hoja. Los nodos hoja también se denominan nodos externos. En un árbol, el nodo hoja también se denomina nodo 'terminal'.

7. Nodos Internos: El nodo que tiene al menos un hijo se denomina nodo interno. Los nodos distintos de los nodos de hoja se denominan nodos internos. También se dice que el nodo raíz es un nodo interno si el árbol tiene más de un nodo. Los nodos internos también se denominan nodos "no terminales".

8. Grado: El número total de hijos de un nodo se denomina grado de ese nodo. El grado más alto de un nodo entre todos los nodos de un árbol se denomina "grado del árbol".

9. Nivel: Se dice que el nodo raíz está en el nivel 0 y los hijos del nodo raíz están en el nivel 1 y los hijos de los nodos que están en el nivel 1 estarán en el nivel 2 y así sucesivamente... osea, cada paso de arriba hacia abajo se llama nivel y el recuento de niveles comienza con '0' y se incrementa en uno en cada nivel.

10. Altura: El número total de aristas desde el nodo hoja hasta un nodo particular en la ruta más larga se denomina altura de ese nodo. En un árbol, se dice que la altura del nodo raíz es la altura del árbol. En un árbol, la altura de todos los nodos hoja es '0'. En un árbol vacío la altura es -1.

11. Profundidad: El número total de aristas desde el nodo raíz hasta un nodo en particular se denomina profundidad de ese nodo. En un árbol, se dice que el número total de aristas desde el nodo raíz hasta un nodo hoja en el camino más largo es la profundidad del árbol. En un árbol, la profundidad del nodo raíz es '0'.

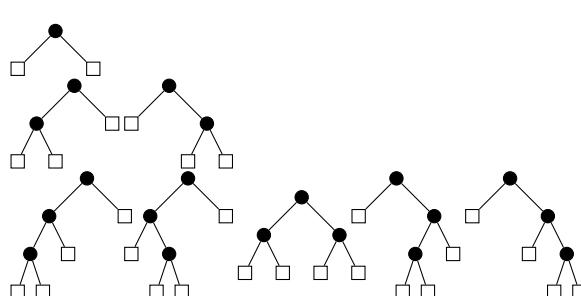
12. Camino: La secuencia de nodos y aristas de un nodo a otro nodo se denomina camino entre esos dos nodos. La longitud de un camino es el número total de nodos en esa ruta. En el siguiente ejemplo, el camino A - B - E - J tiene una longitud de 4.

13. Árbol secundario o subárbol: Cada hijo de un nodo forma un subárbol recursivamente. Cada nodo secundario formará un subárbol en su nodo principal.

Indicar el ejemplo de la terminología con respecto al árbol ejemplo

SUBSECCIÓN 14.2

Binarios



Números de Catalán:
Recurrencia:

$$\begin{aligned} b_0 &= 1 \\ b_{n+1} &= \sum_{k=0}^n b_k b_{n-k} \end{aligned}$$

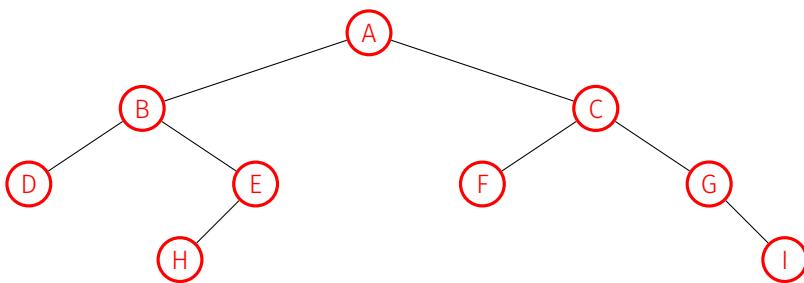
$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

Recorridos

- PreOrder: Raíz-Izquierda-Derecha
- InOrder: Izquierda-Raíz-Derecha

- PostOrder: Izquierda-Derecha-Raíz

Ejemplo 16



Preorder : ABDEHCFGI

InOrder: DBHEAFCGI

PostOrder: DHEBFIGCA

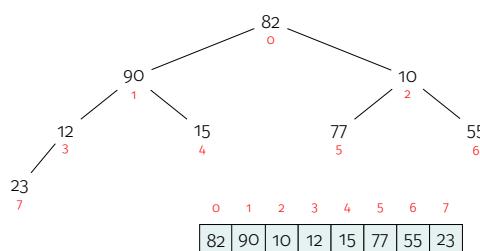
SUBSECCIÓN 14.3

Heap

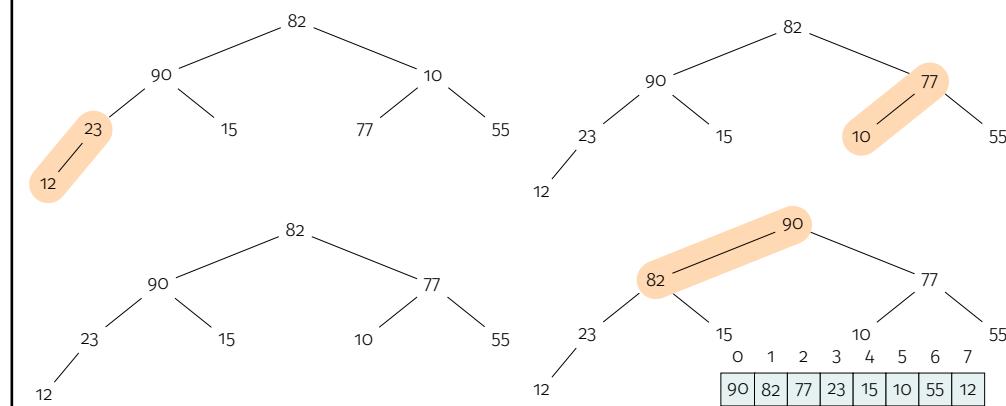
Un heap es una estructura de datos que tiene como representación un árbol binario "semiordenado", pero en la práctica es un arreglo. Cada nodo del árbol corresponde a un elemento del arreglo. Si bien pareciera que el árbol se puede construir balanceado y completo, usualmente el subárbol derecho puede carecer de hojas en el mismo nivel que el subárbol izquierdo. La condición de un max-heap establece que el padre es mayor que los hijos.

Ejemplo 17

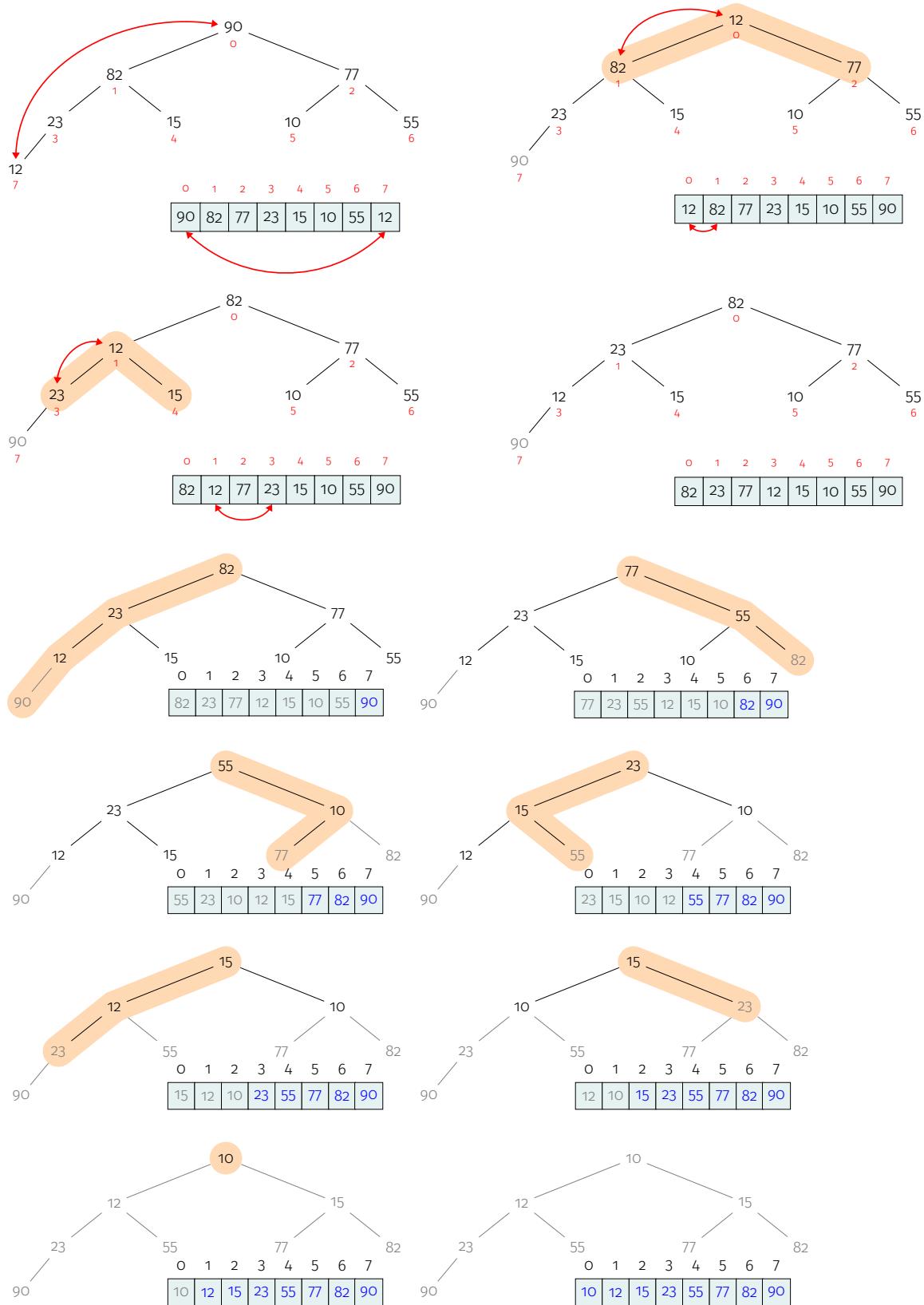
Se desea ordenar el conjunto: 82, 90, 10, 12, 15, 77, 55, 23.



Construir el HEAP



HEAPSORT



Algoritmo 11 Max Heapify

```

1: procedure MAX-HEAPIFY( $A, i$ ) ▷  $A$  arreglo
2:    $l \leftarrow \text{LEFT}(i)$ 
3:    $r \leftarrow \text{RIGHT}(i)$ 
4:   if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$  then
5:      $\text{largest} \leftarrow l$ 
6:   else
7:      $\text{largest} \leftarrow i$ 
8:   if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$  then
9:      $\text{largest} \leftarrow r$ 
10:    if  $\text{largest} \neq i$  then
11:      exchange  $A[i]$  with  $A[\text{largest}]$ 
12:      MAX-HEAPIFY( $A, \text{largest}$ )

```

Algoritmo 12 HeapSort

```

1: procedure HEAPSORT( $A$ ) ▷  $A$  arreglo
2:   BUILD-MAX-HEAP( $A$ )
3:   for  $i \leftarrow A.\text{length}$  downto 2 do
4:     exchange  $A[1]$  with  $A[i]$ 
5:      $A.\text{heap-size} \leftarrow A.\text{heap-size} - 1$ 
6:     MAX-HEAPIFY( $A, 1$ )

```

Algoritmo 13 Construir Heap de Máximos

```

1: procedure BUILD-MAX-HEAP( $A$ ) ▷  $A$  arreglo
2:    $A.\text{heap-size} \leftarrow A.\text{length}$ 
3:   for  $i \leftarrow \lfloor A.\text{length}/2 \rfloor$  downto 1 do
4:     MAX-HEAPIFY( $A, 1$ )

```

Ejercicios

14.3.1 Construya un heap-max H con los números dados a continuación: 45, 36, 54, 27, 63, 72, 61, y 18. Vaya dibujando la representación del heap.

14.3.2 Dado un arreglo con los siguientes elementos 45 27 36 18 16 21 23 10, se pide crear el heap.

14.3.3 Del problema anterior borre el máximo elemento he indique cuales serían las operaciones para volver a la condición de heap.

14.3.4 Usar HEAPSORT para ordenar 45 27 36 18 16 21 23 10.

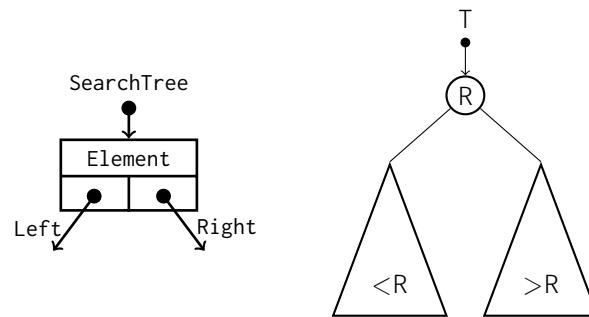
SUBSECCIÓN 14.4

Árbol Binario de Búsqueda (ABB)

```

1 struct TreeNode
2 {
3     ElementType Element;
4     SearchTree Left;
5     SearchTree Right;
6 };
7 typedef struct TreeNode *Position;
8 typedef struct TreeNode *SearchTree;

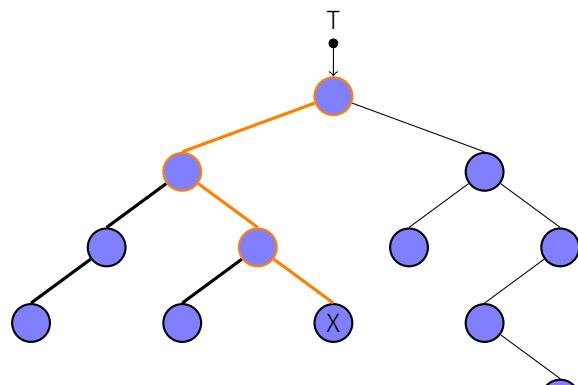
```



```

1 SearchTree Insert( ElementType X, SearchTree T )
2 {
3     if( T == NULL ) {
4         T = malloc( sizeof( struct TreeNode ) );
5         if( T == NULL )
6             FatalError( "Out of space!!!" );
7         else {
8             T->Element = X;
9             T->Left = T->Right = NULL;
10        }
11    }
12    else
13        if( X < T->Element )
14            T->Left = Insert( X, T->Left );
15        else
16            if( X > T->Element )
17                T->Right = Insert( X, T->Right );
18
19    return T;
20 }

```



```

1 Position Find( ElementType X, SearchTree T )
2 {
3     if( T == NULL )
4         return NULL;
5     if( X < T->Element )
6         return Find( X, T->Left );
7     else
8         if( X > T->Element )
9             return Find( X, T->Right );
10        else
11            return T;
12 }

```

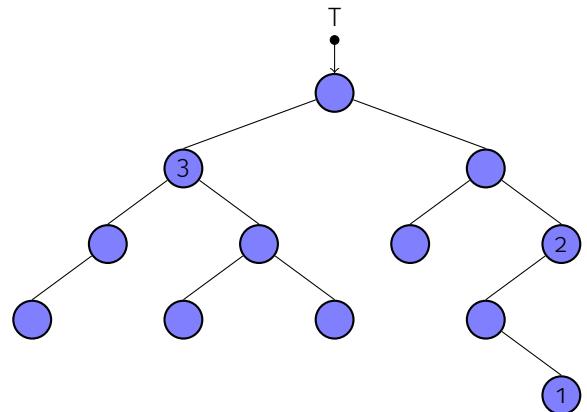
```

1 Position FindMin( SearchTree T )
2 {
3     if( T == NULL )
4         return NULL;
5     else
6         if( T->Left == NULL )
7             return T;
8         else
9             return FindMin( T->Left );
10 }

```

```

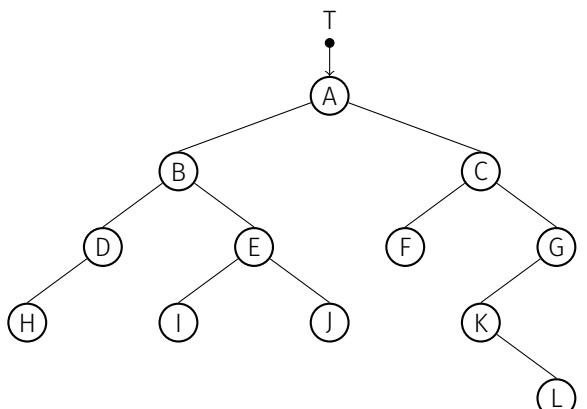
1 SearchTree Delete( ElementType X, SearchTree T )
2 {
3     Position TmpCell;
4
5     if( T == NULL )
6         Error( "Element not found" );
7     else
8         if( X < T->Element )
9             T->Left = Delete( X, T->Left );
10        else
11            if( X > T->Element )
12                T->Right = Delete( X, T->Right );
13            else
14                if( T->Left && T->Right )
15                {
16                    TmpCell = FindMin( T->Right );
17                    T->Element = TmpCell->Element;
18                    T->Right = Delete( T->Element, T->Right );
19                }
20            else
21            {
22                TmpCell = T;
23                if( T->Left == NULL )
24                    T = T->Right;
25                else if( T->Right == NULL )
26                    T = T->Left;
27                free( TmpCell );
28            }
29
30    return T;
31 }
```



Recorridos

```

1 void InOrder(SearchTree T , void (*func)(ElementType ))
2 {
3     if(T->Left) InOrder(T->Left, func);
4     func(T->Element);
5     if(T->Right) InOrder(T->Right, func);
6 }
7
8 void PreOrder(SearchTree T, void (*func)(ElementType ))
9 {
10    func(T->Element);
11    if(T->Left) PreOrder(T->Left, func);
12    if(T->Right) PreOrder(T->Right, func);
13 }
14
15 void PostOrder(SearchTree T, void (*func)(ElementType ))
16 {
17     if(T->Left) PostOrder(T->Left, func);
18     if(T->Right) PostOrder(T->Right, func);
19     func(T->Element);
20 }
```



RID: ABDHEIJCFGKL
 IRD: HDBIEJAFCCKLG
 IDR: HDIJEFBGLKCA

Ejercicios

14.4.1 Dados los dos recorridos siguientes, encontrar el árbol binario correspondiente:

- preorden: 16, 12, 30, 21, 4, 8
- inorden: 12, 30, 16, 4, 8, 21

14.4.2 Implementar una función que imprima el nodo más profundo

14.4.3 Implementar una función que imprima el camino de la raíz al nodo más profundo

14.4.4 Convierta la expresión prefija $-/ab^*+bcd$ en la expresión infija correspondiente.

14.4.5 Implementar la función `int height(T)`, que entrega la altura de T.

14.4.6 Crear un ABB con el siguiente conjunto: 98, 2, 48, 12, 56, 32, 4, 67, 23, 87, 23, 55, 46.

14.4.7 En el ejercicio anterior: (a) Inserte 21, 39, 45, 54, y 63. (b) Borre los valores 23, 56, 2, y 45.

14.4.8 Para el ejercicio anterior imprima el recorrido: preorden, inorder y postorden.

14.4.9 Modificar la estructura para que cada nodo tenga un puntero al padre.

14.4.10 Implementar una función que imprima todas las hojas de un ABB.

SUBSECCIÓN 14.5

AVL

Condición de balance: $|height(Left_s) - height(Right_s)| \leq 1$

```

1 struct AvlNode
2 {
3     ElementType Element;
4     AvlTree Left;
5     AvlTree Right;
6     int Height;
7 };
8 typedef struct AvlNode *Position;
9 typedef struct AvlNode *AvlTree;
```

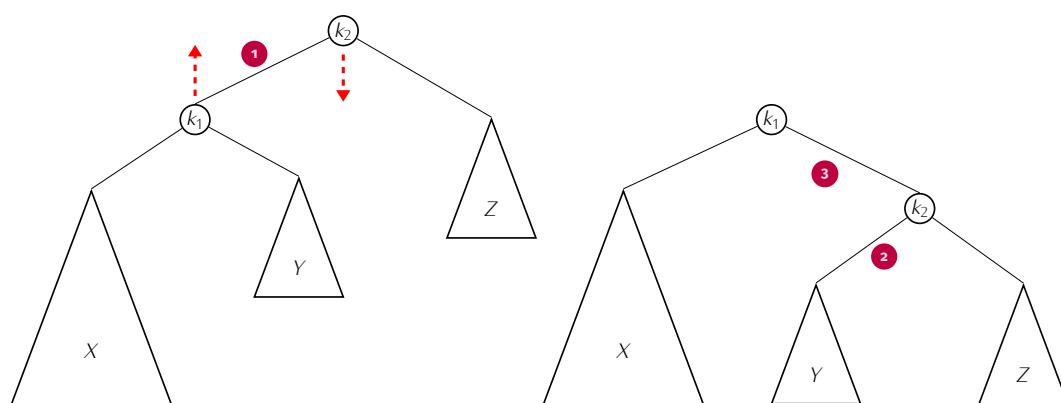
```

static Position SingleRotateWithLeft( Position K2 )
{
    Position K1;

    ① K1 = K2->Left;
    ② K2->Left = K1->Right;
    ③ K1->Right = K2;

    K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) + 1;
    K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;

    return K1;
}
```



```

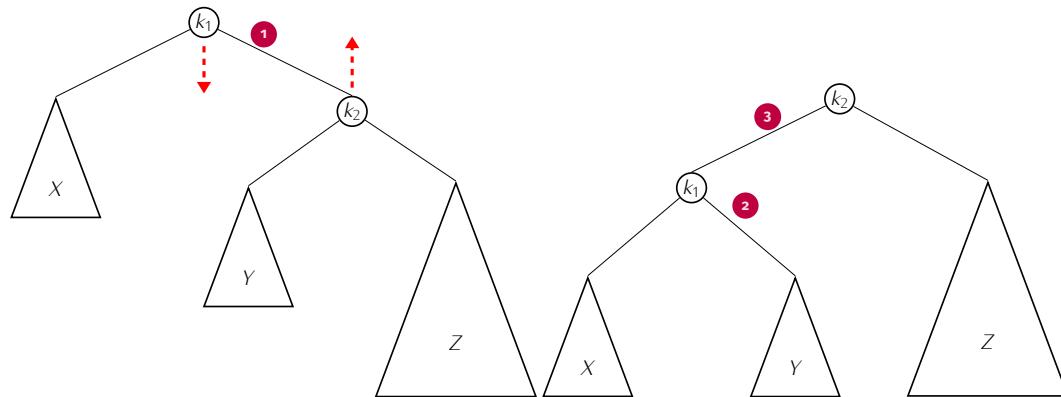
static Position SingleRotateWithRight( Position K1 )
{
    Position K2;

    ① K2 = K1->Right;
    ② K1->Right = K2->Left;
    ③ K2->Left = K1;

    K1->Height = Max( Height( K1->Left ), Height( K1->Right ) ) + 1;
    K2->Height = Max( Height( K2->Right ), K1->Height ) + 1;

    return K2;
}

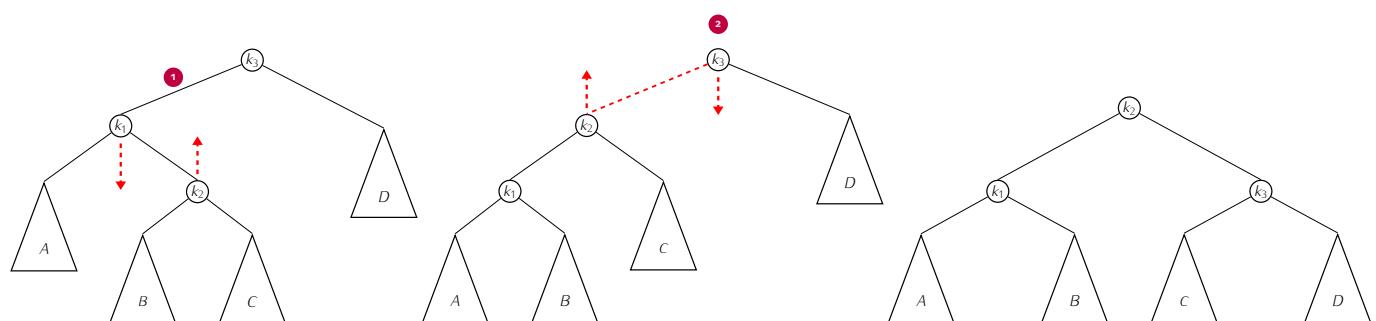
```



```

static Position DoubleRotateWithLeft( Position K3 )
{
    ① K3->Left = SingleRotateWithRight( K3->Left );
    return SingleRotateWithLeft( K3 ); ②
}

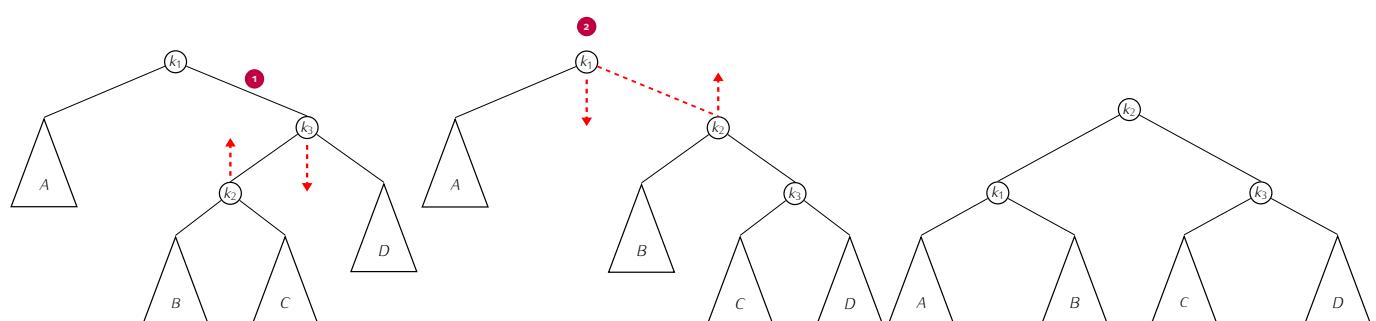
```



```

static Position DoubleRotateWithRight( Position K1 )
{
    ① K1->Right = SingleRotateWithLeft( K1->Right );
    return SingleRotateWithRight( K1 ); ②
}

```

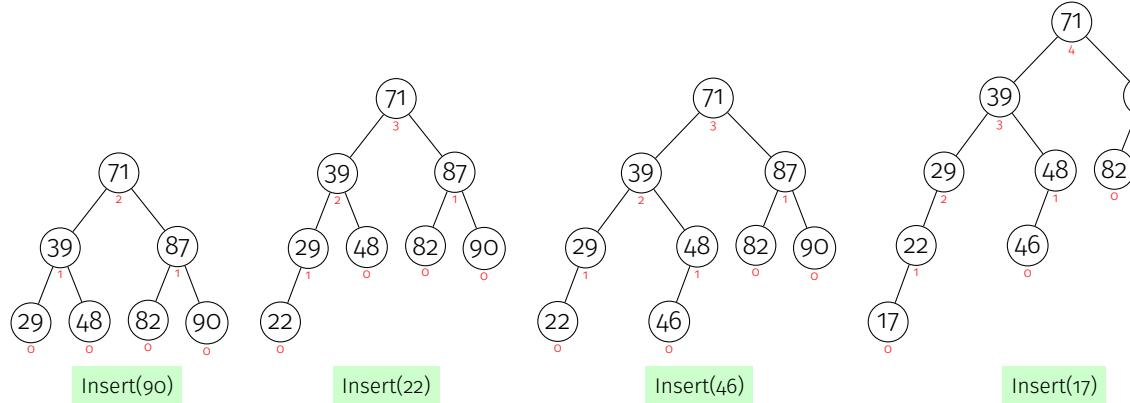
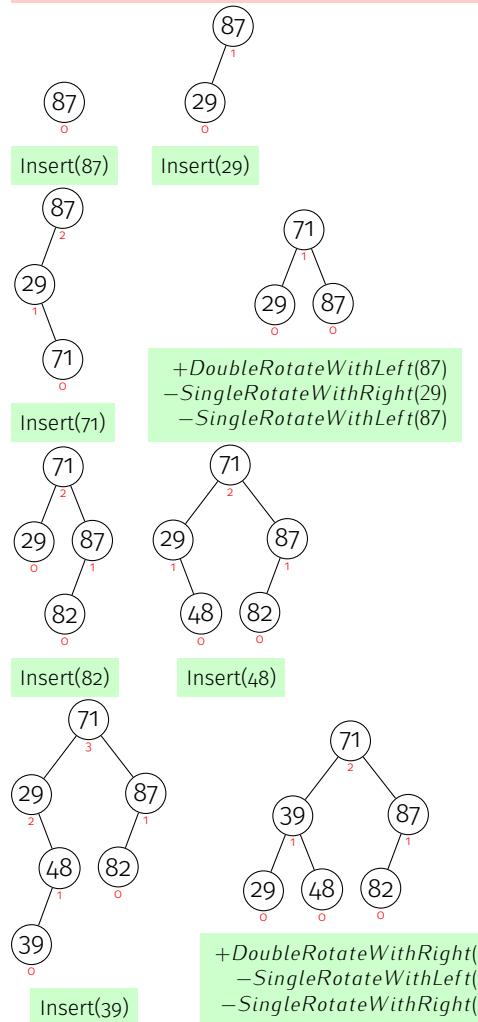


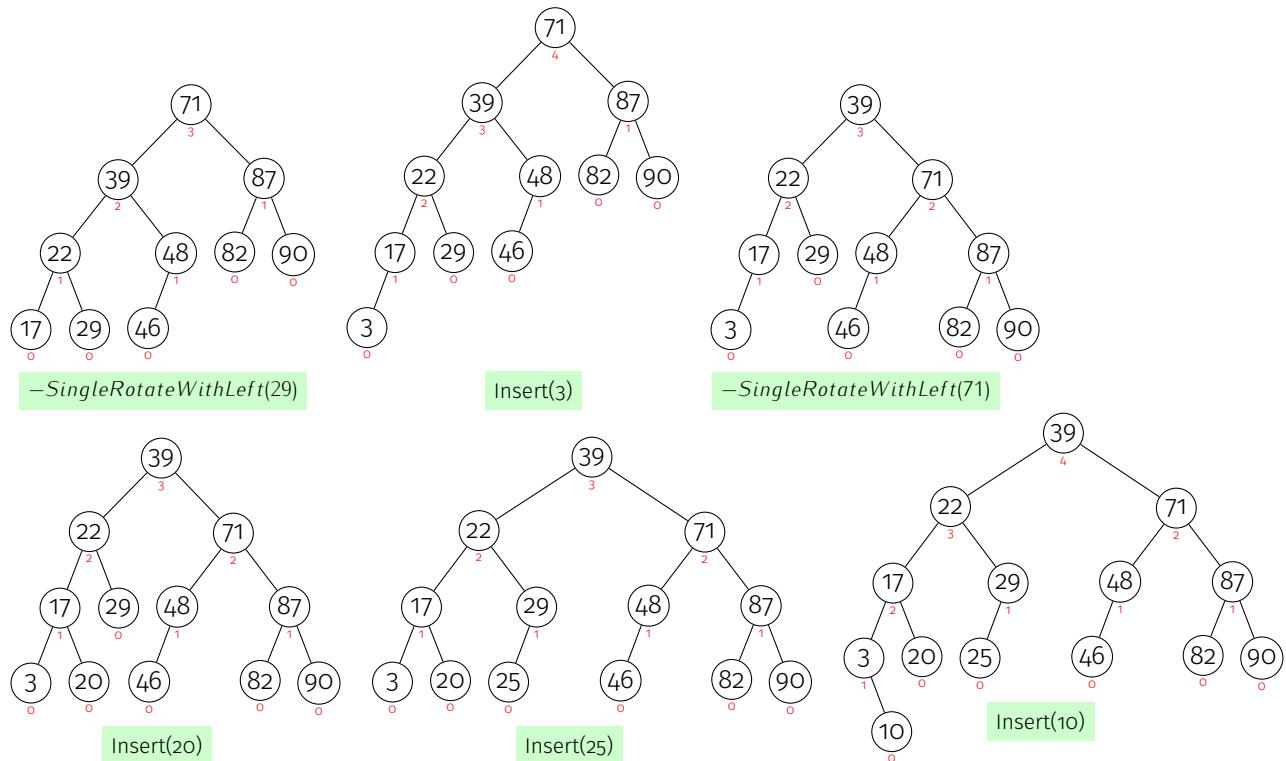
```

1 AvlTree Insert( ElementType X, AvlTree T )
2 {
3     if( T == NULL )
4     {
5         T = malloc( sizeof( struct AvlNode ) );
6         if( T == NULL )
7             FatalError( "Out of space!!!" );
8         else
9         {
10            T->Element = X; T->Height = 0;
11            T->Left = T->Right = NULL;
12        }
13    }
14    else
15    {
16        if( X < T->Element )
17        {
18            T->Left = Insert( X, T->Left );
19            if( Height( T->Left ) - Height( T->Right ) == 2 )
20                if( X < T->Left->Element )
21                    T = SingleRotateWithLeft( T );
22                else
23                    T = DoubleRotateWithLeft( T );
24        }
25        else
26        {
27            T->Right = Insert( X, T->Right );
28            if( Height( T->Right ) - Height( T->Left ) == 2 )
29                if( X > T->Right->Element )
30                    T = SingleRotateWithRight( T );
31                else
32                    T = DoubleRotateWithRight( T );
33        }
34        T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
35        return T;
36    }
}

```

Insert:87 29 71 82 48 39 90 22 46 17 3 20 25 10





Ejercicios

14.5.1 Dado el siguiente conjunto realizar la inserción en un árbol AVL, A=36,27,63,72,45,39,54,70.

14.5.2 Al árbol anterior insertar: 18, 81, 29, 15, 19, 25, 26 y 1.

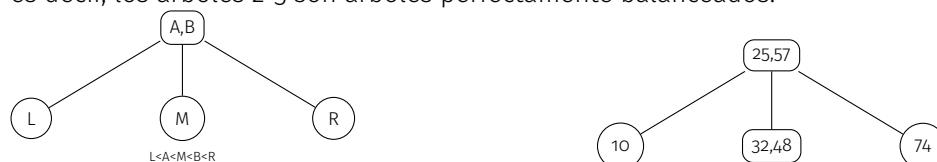
14.5.3 Insertar 63, 9, 19, 27, 18, 108, 99, 81 en un árbol AVL vacío.

SUBSECCIÓN 14.6

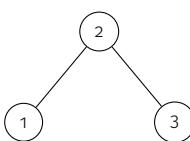
Árbol 2-3

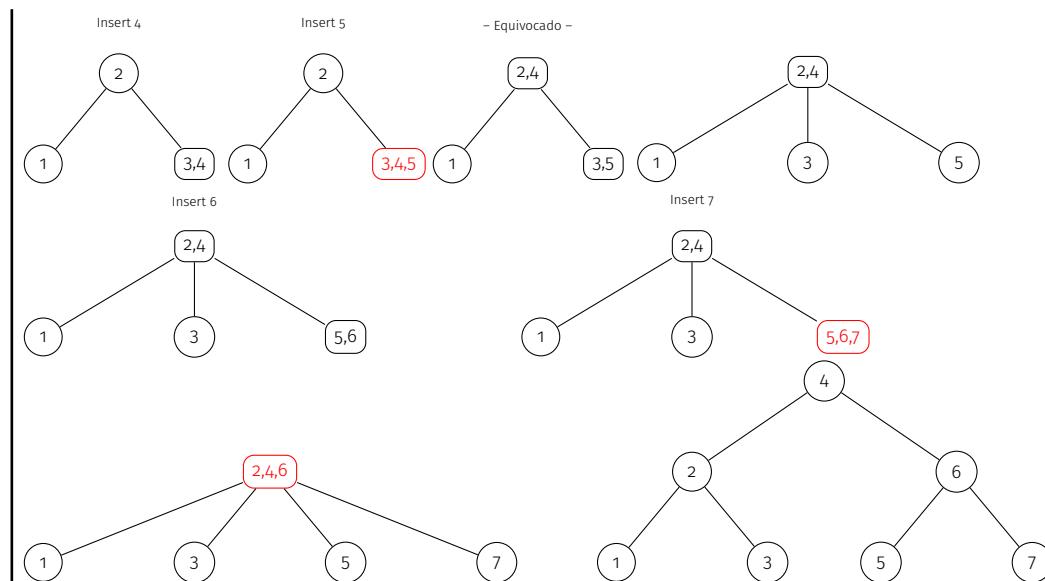
Los árboles 2-3 son árboles cuyos nodos internos pueden contener hasta 2 elementos y por lo tanto un nodo interno puede tener 2 o 3 hijos, dependiendo de cuántos elementos posea el nodo.

Una propiedad de los árboles 2-3 es que todas las hojas están a la misma profundidad, es decir, los árboles 2-3 son árboles perfectamente balanceados.



Ejemplo 18 | Insertar el conjunto A={1,2,3,4,5,6,7} en un árbol 2-3.





Ejercicios

14.6.1 Insertar el siguiente conjunto en un árbol 2-3, $A=\{15, 67, 08, 16, 44, 27, 12, 35\}$.

14.6.2 Como podría implementar la búsqueda y el borrado en un árbol 2-3.

SUBSECCIÓN 14.7

Árbol B

La idea de los árboles 2-3 se puede generalizar a árboles $t - (2t - 1)$, donde $t \geq 2$ es un parámetro fijo, es decir, cada nodo del árbol posee entre t y $2t - 1$ hijos, excepto por la raíz que puede tener entre 2 y $2t - 1$ hijos. En la práctica, t puede ser bastante grande, por ejemplo $t = 100$ o más.



Comparison of B-Tree and Hash Indexes in MySQL

SECCIÓN 15

Trie

Un Trie, también conocido como árbol de prefijos, es una estructura de datos en forma de árbol donde las claves son generalmente cadenas de caracteres. Cada nodo en el trie representa un carácter, y el camino desde la raíz hasta una hoja representa una clave.

```

1 #define ALPHABET_SIZE 26
2
3 typedef struct TrieNode {
4     struct TrieNode *children[ALPHABET_SIZE];
5     int isEndOfWord;
6 } TrieNode;
```

Ejemplo de Uso: Los Tries son ideales para:

- **Autocompletado:** Sugerir palabras a medida que el usuario escribe.
- **Búsqueda de Prefijos:** Encontrar todas las palabras que comienzan con un prefijo dado.
- **Ordenamiento Lexicográfico:** Ordenar un conjunto de cadenas eficientemente.

SECCIÓN 16

Construcción del Trie

Imaginemos que tenemos un conjunto de palabras: "trie", "trigger", "trick", y "trip". Vamos a construir un Trie que almacene estas palabras.

SUBSECCIÓN 16.1

Paso 1: Insertar "trie"

Insertamos "t", "r", "i", "e" como nodos en el Trie.

SUBSECCIÓN 16.2

Paso 2: Insertar "trigger"

"t", "r", "i" ya están presentes, por lo que reutilizamos esos nodos. Añadimos "g", "g", "e", "r".

SUBSECCIÓN 16.3

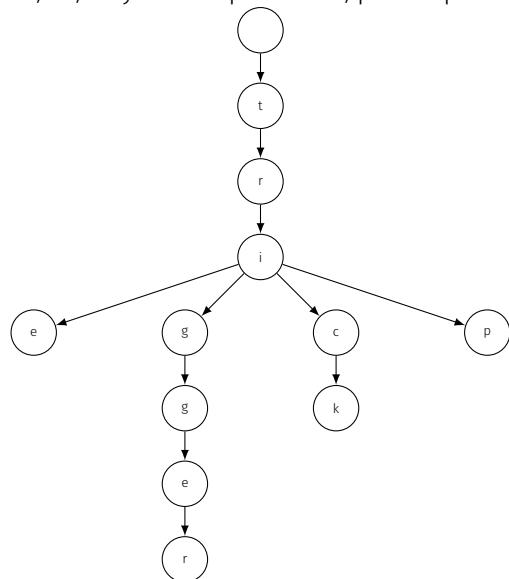
Paso 3: Insertar "trick"

"t", "r", "i" ya están presentes, por lo que reutilizamos esos nodos. Añadimos "c", "k".

SUBSECCIÓN 16.4

Paso 4: Insertar "trip"

"t", "r", "i" ya están presentes, por lo que reutilizamos esos nodos. Añadimos "p".



SECCIÓN 17

Suffix Tree

Un Suffix Tree es una estructura de datos en forma de árbol que representa todos los sufijos de una cadena dada. Se utiliza para realizar búsquedas de patrones en tiempo lineal, encontrar el sufijo más largo común entre cadenas, y otros problemas relacionados con cadenas. Esencialmente, es un Trie comprimido de todos los sufijos.

SUBSECCIÓN 17.1

Ejemplo de Uso:

Los Suffix Trees son potentes para:

- **Búsqueda de Patrones más Rápida:** Similar a los Suffix Arrays, pero con potencialmente mejor rendimiento en algunos casos.
- **Longest Common Substring:** Encontrar la subcadena más larga compartida entre dos o más cadenas.

SECCIÓN 18

Construcción del Suffix Tree para "banana\$"

SUBSECCIÓN 18.1

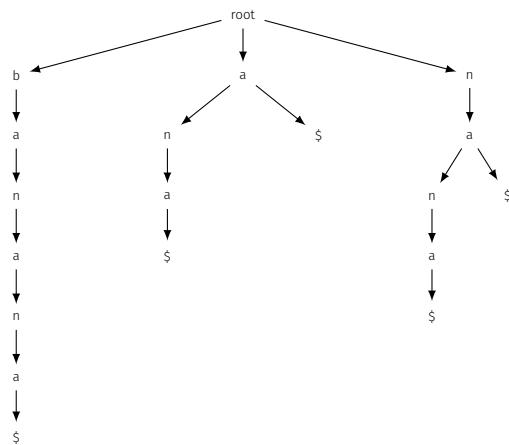
Paso 1: Identificar los Sufijos

Primero, enumeramos todos los sufijos de la cadena "banana\$":

- banana\$
- anana\$
- nana\$
- ana\$
- na\$
- a\$
- \$

SUBSECCIÓN 18.2

Paso 2: Construcción del Árbol



SECCIÓN 19

Suffix Array

Un Suffix Array es una estructura de datos que contiene todos los sufijos de una cadena en orden lexicográfico. Es una alternativa más compacta al Suffix Tree y permite búsquedas eficientes.

Un Suffix Array es un array ordenado de los índices de inicio de todos los sufijos de una cadena. Es decir, si tienes la cadena "banana", el Suffix Array sería [5, 3, 1, 0, 4, 2], ya que los sufijos ordenados lexicográficamente son "a", "ana", "anana", "banana", "na", "nana".

```

1 int cmpSuffix(const void *a, const void *b) {
2     return strcmp((*(const char **))a, (*(const char **))b));
3 }
4
5 void buildSuffixArray(char *text, int n, char **suffixArray) {
6     for (int i = 0; i < n; i++) {
7         suffixArray[i] = text + i;
8     }
9     qsort(suffixArray, n, sizeof(char *), cmpSuffix);
10}
  
```

Ejemplos de uso:

- **Búsqueda de Subcadenas:** Verificar si una subcadena está presente en la cadena original.
- **Comparación de Cadenas:** Para encontrar el LCS (Longest Common Substring).
- **Compresión de Texto:** Utilizado en algoritmos como Burrows-Wheeler Transform.

SUBSECCIÓN 19.1

Paso 1: Generar los Sufijos

Primero, generamos todos los sufijos de la cadena "banana\$":

0 1 2 3 4 5 6

b	a	n	a	n	a	\$
---	---	---	---	---	---	----

O banana\$

- 1 anana\$
- 2 nana\$
- 3 ana\$
- 4 na\$
- 5 a\$
- 6 \$

SUBSECCIÓN 19.2

Paso 2: Ordenar los Sufijos en Orden Lexicográfico

A continuación, ordenamos estos sufijos en orden lexicográfico (es decir, como en un diccionario):

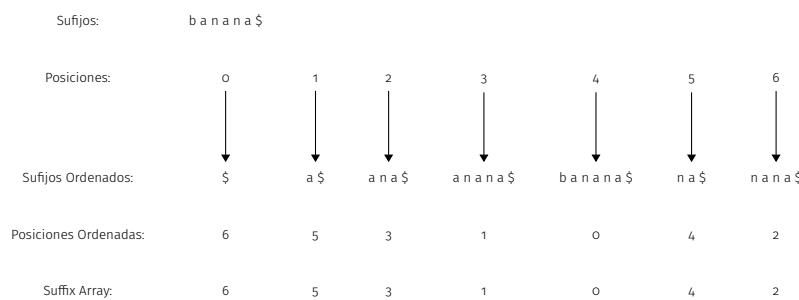
- 6 \$
- 5 a\$
- 3 ana\$
- 1 anana\$
- 0 banana\$
- 4 na\$
- 2 nana\$

SUBSECCIÓN 19.3

Paso 3: Construir el Suffix Array

El Suffix Array es simplemente un arreglo que contiene las posiciones iniciales de estos sufijos en el orden ordenado: **Suffix Array:** [6, 5, 3, 1, 0, 4, 2] Este arreglo indica que:

- El sufijo en la posición 6 ("\$") es el menor lexicográficamente.
- El sufijo en la posición 5 ("a\$") es el siguiente menor.
- Y así sucesivamente.



Ejemplo de Búsqueda: Si queremos buscar la subcadena "na" en el Suffix Array, podemos utilizar una búsqueda binaria. Como los sufijos están ordenados, podemos encontrar rápidamente el rango de sufijos que comienzan con "na":

- **Búsqueda binaria:** Buscamos "na" en el Suffix Array y encontramos que los sufijos que comienzan con "na" están en los índices 0 y 1.

- **Ocurrencias:** Los sufijos en los índices 0 y 1 son "na\$" y "banana\$", lo que significa que la subcadena "na" aparece dos veces en la cadena original: comenzando en la posición 0 ("na\$") y en la posición 4 ("na").

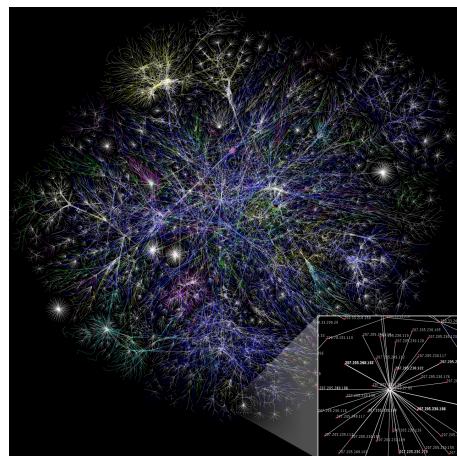
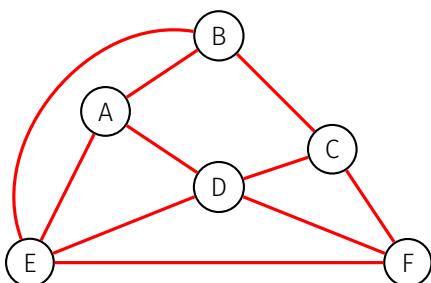
Ventajas del Suffix Array:

- **Espacio eficiente:** Ocupa menos memoria que un Suffix Tree.
- **Construcción eficiente:** Existen algoritmos eficientes para construir el Suffix Array.
- **Búsqueda eficiente:** Permite buscar subcadenas en tiempo logarítmico utilizando búsqueda binaria.

SECCIÓN 20

Grafos

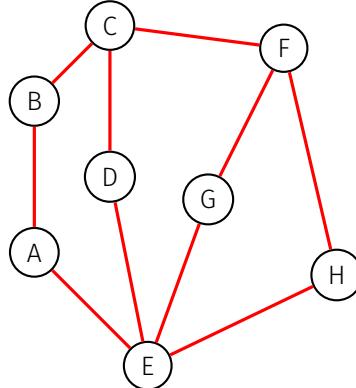
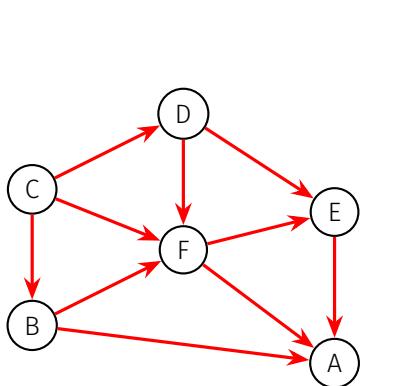
Los grafos son una forma de representar formalmente una red de datos o una colección de objetos interconectados. Se ve como $G = (V, E)$, donde V es el conjunto de vértices, también llamados nodos y E es el conjunto de aristas, también llamados enlaces.



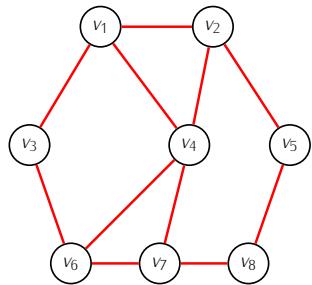
The Internet 1997 - 2021
Internet map 1024

Al momento de definir o reconocer los tipos de grafos son importantes los tipos de aristas. De hecho, ese es uno de los diferenciadores más grandes y obvios entre un grafo y otro: el tipo de arista que tiene. Los grafos pueden tener dos tipos de bordes: uno que tiene dirección o flujo y otro que no. Nos referimos a estos como dirigidos y no-dirigidos.

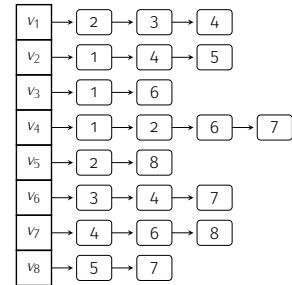
Luego tendremos grafos dirigidos y no-dirigidos.



SUBSECCIÓN 20.1

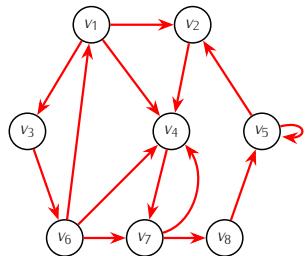
Representación

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	0	1	1	1	0	0	0	0
v_2	1	0	0	1	1	0	0	0
v_3	1	0	0	0	0	1	0	0
v_4	1	1	0	0	0	1	1	0
v_5	0	1	0	0	0	0	0	1
v_6	0	0	1	1	0	0	1	0
v_7	0	0	0	1	0	1	0	1
v_8	0	0	0	0	1	0	1	0

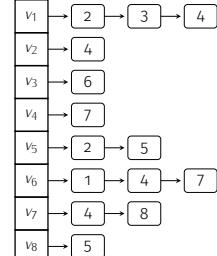


$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\},$$

$$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_4), (v_3, v_6), (v_4, v_7), (v_5, v_8), (v_6, v_7), (v_7, v_8)\}.$$



	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	0	1	1	1	0	0	0	0
v_2	0	0	0	1	0	0	0	0
v_3	0	0	0	0	0	1	0	0
v_4	0	0	0	0	0	0	1	0
v_5	0	1	0	0	1	0	0	0
v_6	1	0	0	1	0	0	1	0
v_7	0	0	0	1	0	0	0	1
v_8	0	0	0	0	1	0	0	0

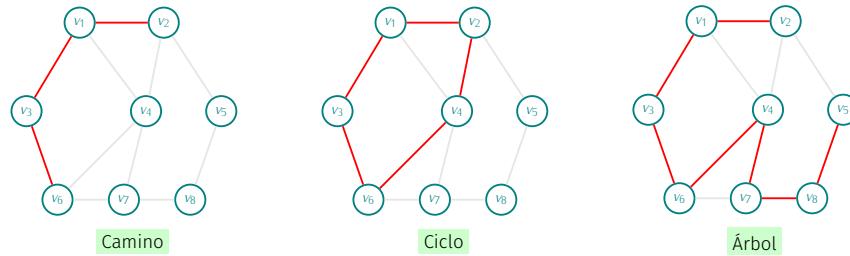


$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\},$$

$$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_4), (v_3, v_6), (v_4, v_7), (v_5, v_8), (v_6, v_1), (v_6, v_4), (v_6, v_7), (v_7, v_8), (v_7, v_4), (v_7, v_8), (v_8, v_5)\}.$$

Caminos, ciclos y árboles

Un camino es una secuencia de arcos en que el extremo final de cada arco coincide con el extremo inicial del siguiente en la secuencia. Un camino es simple si no se repiten vértices, excepto posiblemente el primero y el último. Un ciclo es un camino simple y cerrado. Un grafo es conexo si desde cualquier vértice existe un camino hasta cualquier otro vértice del grafo. Se dice que un grafo no dirigido es un árbol si es conexo y acíclico.



Recorridos

Algoritmo 14 Depth-first Search

```

1: procedure DFS( $G(V, E)$ ,  $v$ )
2:   PUSH( $S, v$ )
3:    $visited \leftarrow \{v\}$                                  $\triangleright v$  is visited
4:   while ! ISEMPTY( $S$ ) do
5:      $v \leftarrow \text{TOPPOP}(S)$ 
6:     for all  $w|(v, w) \in E$  do
7:       if  $w \notin visited$  then
8:         PUSH( $S, w$ )
9:          $visited \leftarrow visited \cup \{w\}$ 

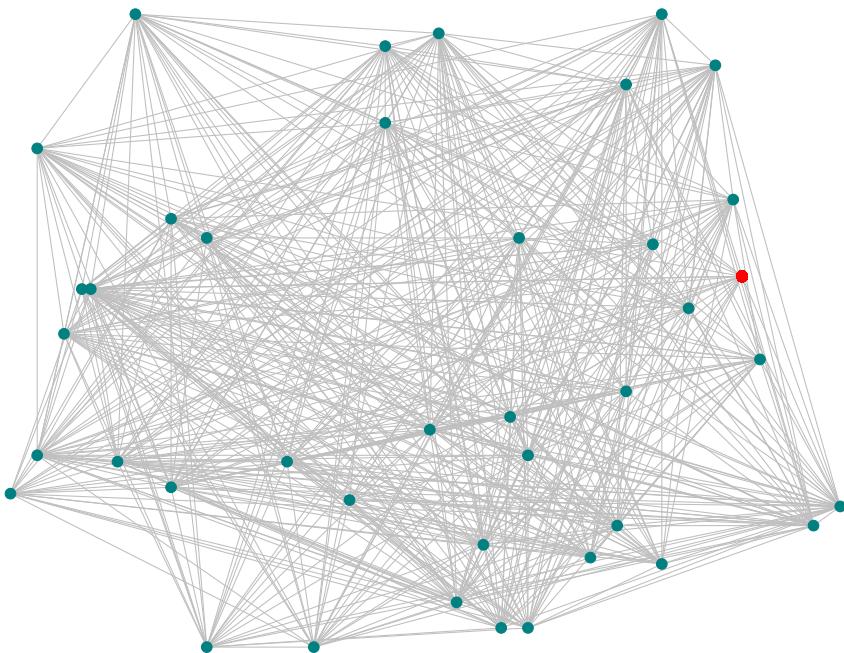
```

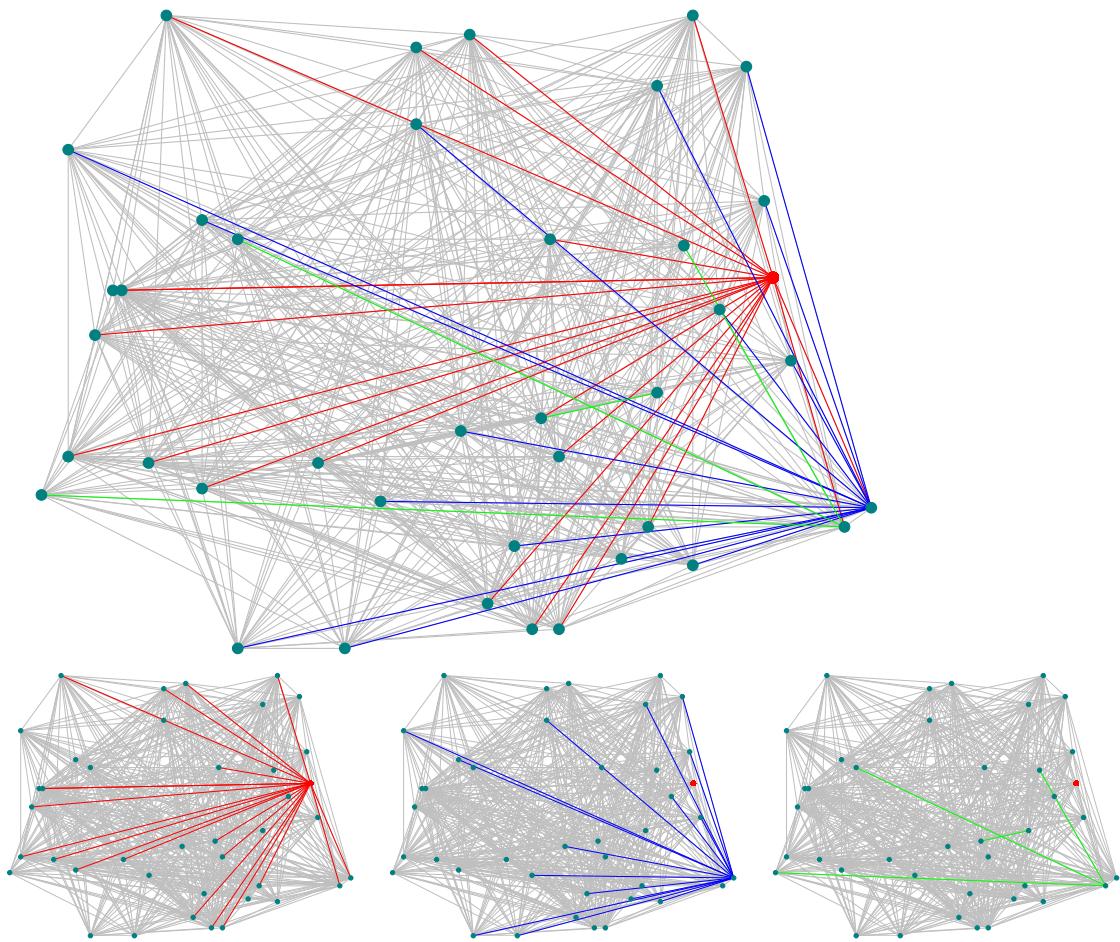
Algoritmo 15 Breadth-first Search

```

1: procedure BFS( $G(V, E)$ ,  $v$ )
2:   ENQUEUE( $Q, v$ )
3:    $visited \leftarrow \{v\}$                                  $\triangleright v$  is visited
4:   while ! ISEMPTY( $Q$ ) do
5:      $v \leftarrow \text{DEQUEUE}(Q)$ 
6:     for all  $w|(v, w) \in E$  do
7:       if  $w \notin visited$  then
8:         ENQUEUE( $Q, w$ )
9:          $visited \leftarrow visited \cup \{w\}$ 

```



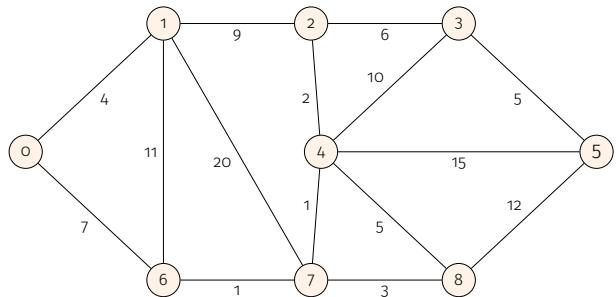


SUBSECCIÓN 20.2

Caminos mínimos**Algoritmo de dijkstra**

La idea del algoritmo es mantener un conjunto A de nodos "alcanzables" desde el nodo origen e ir extendiendo este conjunto en cada iteración. Los nodos alcanzables son aquellos para los cuales ya se ha encontrado su camino óptimo desde el nodo origen. Para esos nodos su distancia óptima al origen es conocida. Inicialmente $A = \{s\}$. Para los nodos que no están en A se puede conocer el camino óptimo desde s que pasa sólo por nodos de A . Esto es, caminos en que todos los nodos intermedios son nodos de A . Llámemos a esto su camino óptimo tentativo.

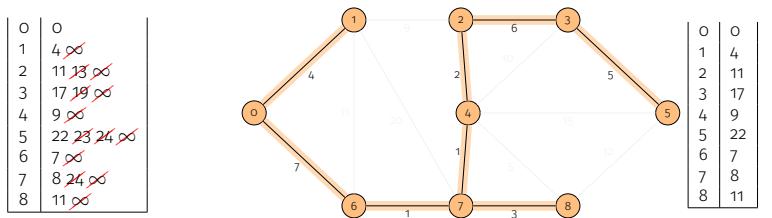
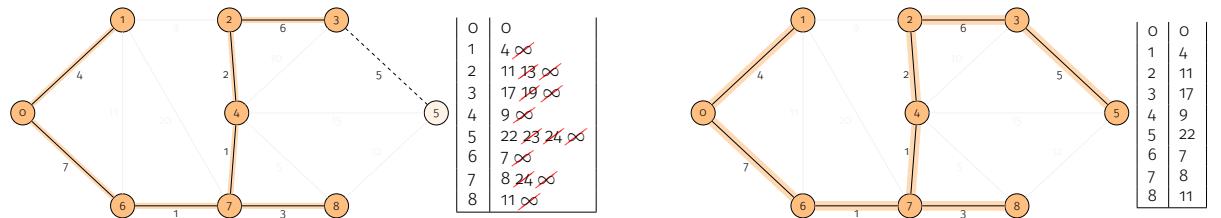
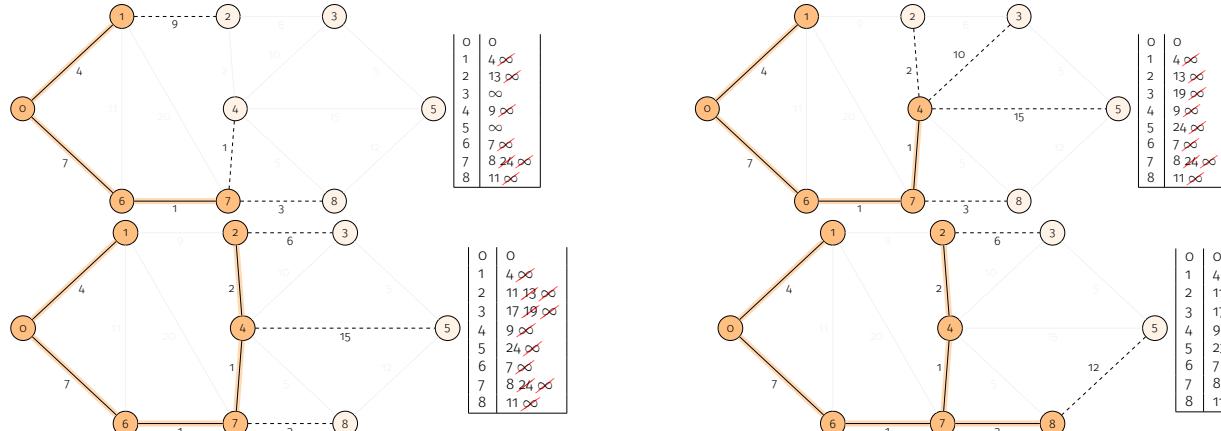
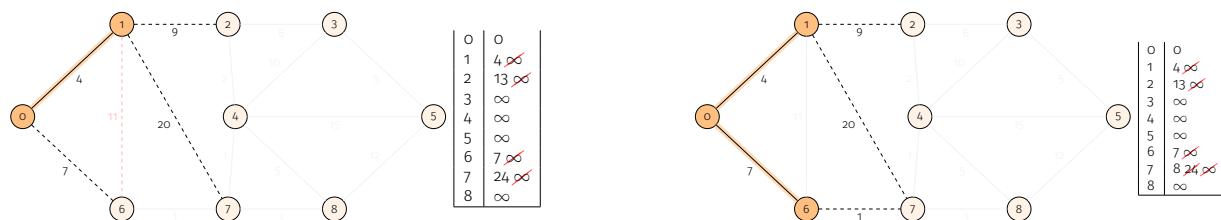
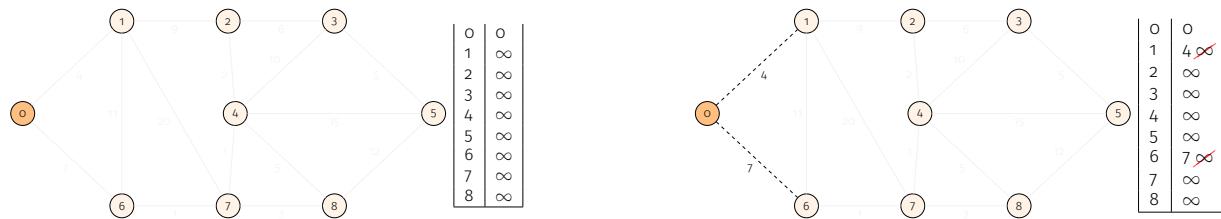
En cada iteración, el algoritmo encuentra el nodo que no está en A y cuyo camino óptimo tentativo tiene largo mínimo. Este nodo se agrega a A y su camino óptimo tentativo se convierte en su camino óptimo. Luego se actualizan los caminos óptimos tentativos para los demás nodos.

**Algoritmo 16** Camino Mínimo

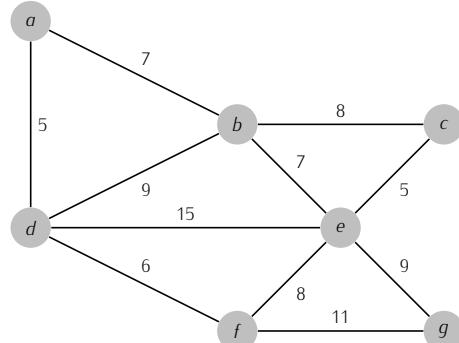
```

1: procedure DIJKSTRA( $G(V, E)$ )
2:   Todos los arcos tienen costo  $\infty$ 
3:    $A \leftarrow \{s\}$             $\triangleright s \in V, A$  conjunto alcanzable
4:    $D_s \leftarrow 0$ 
5:    $D_v \leftarrow \text{costo}(s, v)$             $\triangleright \forall v \in V - A$ 
6:   while  $A \neq V$  do
7:     Find  $v$  tal que  $D_v$  es mínimo            $\triangleright v \in V - A$ 
8:      $A \leftarrow A \cup \{v\}$ 
9:     for todo  $w$  tal que  $(v, w) \in E$  do
10:     $D_w \leftarrow \min(D_w, D_v + \text{cost}(v, w))$ 

```



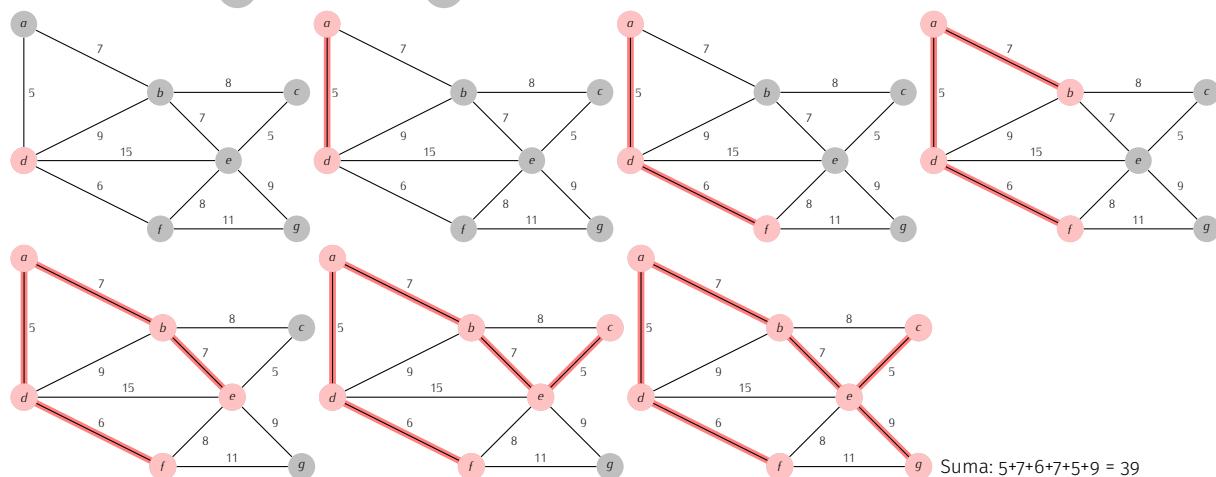
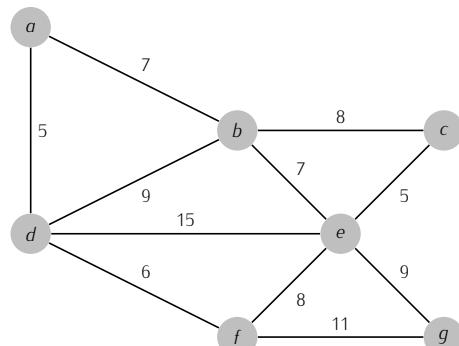
SUBSECCIÓN 20.3

Árbol cobertor mínimo**Prim****Algoritmo 17 Árbol Cobertor Mínimo**

```

1: function PRIM( $G(V, E)$ )
2:    $e \leftarrow \{v, w\}$                                  $\triangleright$  Arco de costo mínimo  $\in E$ 
3:    $T = \{e\}$ 
4:    $A \leftarrow \{v, w\}$                                  $\triangleright$  Conjunto alcanzable
5:   while  $A \neq V$  do
6:     Find  $e = \{v, w\}$                              $\triangleright v \in A, w \in V - A$ 
7:      $T = T \cup \{e\}$ 
8:      $A \leftarrow A + \{v, w\}$ 
return  $T$ 

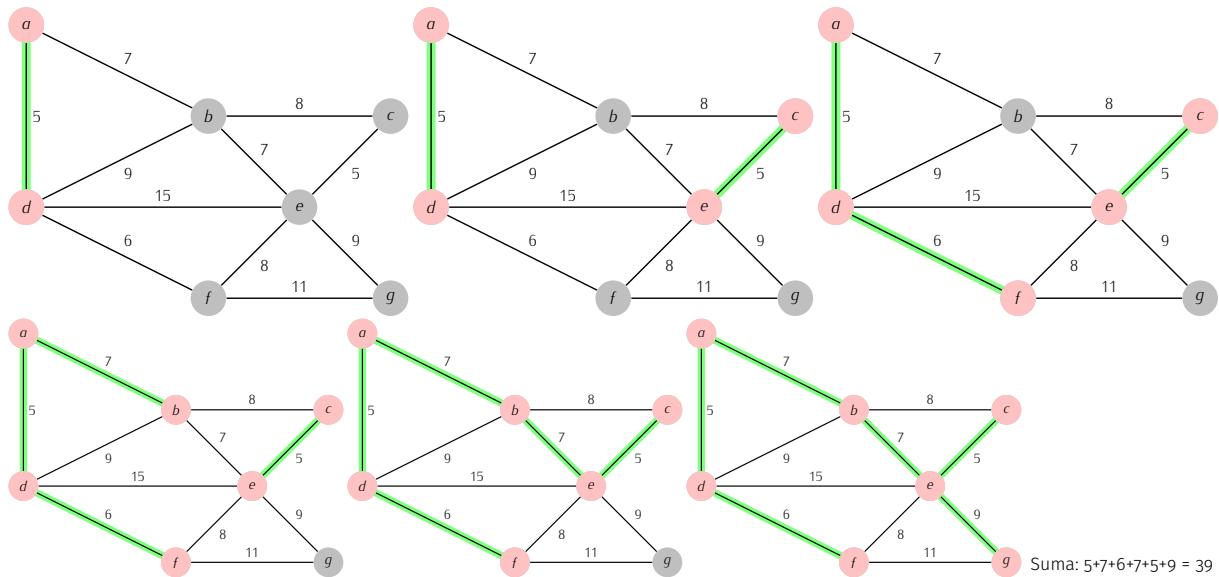
```

**Kruskal****Algoritmo 18 Árbol Cobertor Mínimo**

```

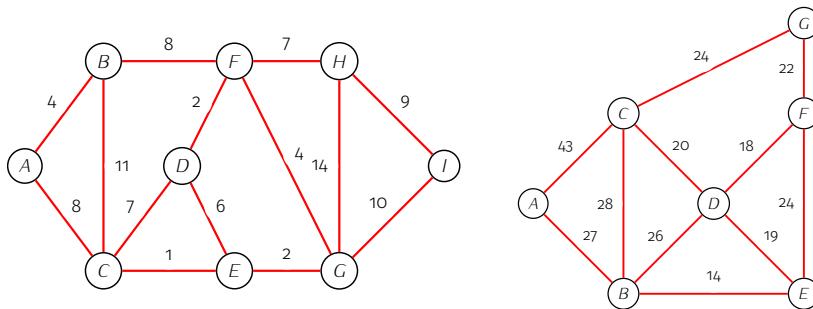
1: function KRUSKAL( $G(V, E)$ )
2:   SORT( $E$ )                                          $\triangleright$  Ordenar el conjunto de arcos
3:    $T = (V, \phi)$ 
4:    $C \leftarrow V$                                       $\triangleright v \in V, C$  componentes conexas
5:   while  $|C| > 1$  do
6:      $e \leftarrow \{v, w\}$                              $\triangleright e$  es el siguiente arco en orden de costo creciente
7:     if  $\text{FIND}(v) \neq \text{FIND}(w)$  then
8:        $T \leftarrow T \cup e$ 
9:        $a \leftarrow \text{FIND}(v)$ 
10:       $b \leftarrow \text{FIND}(w)$ 
11:      UNION( $a, b$ )
return  $T$ 

```



Ejercicios

20.3.1 Para los siguientes grafos determinar el árbol cobertor mínimo y las distancias mínimas. De ser necesario use el nodo marcado como C como nodo origen.



20.3.2 Sea $G = (V, E)$ un Grafo Dirigido con Pesos. $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$, $E = \{(v_0, v_1, 2), (v_0, v_3, 1), (v_1, v_3, 3), (v_1, v_4, 10), (v_3, v_4, 2), (v_3, v_6, 4), (v_3, v_5, 8), (v_3, v_2, 2), (v_2, v_0, 4), (v_2, v_5, 5), (v_4, v_6, 6), (v_6, v_5, 1)\}$.

Se pide dibujar el grafo, encontrar el árbol cobertor y el camino mínimo a partir de v_1 .

SECCIÓN 21

Cuestiones y problemas

21.1 Se pide: Se ha implementado la función $ABB - delete(T, x)$ que permite borrar el elemento x del ABB T . Usando los criterios vistos en clases, use esta función para borrar lo siguientes elementos (asuma que el árbol de la figura se llama T):

$ABB - delete(T, 13)$
 $ABB - delete(T, 16)$
 $ABB - delete(T, 5)$

Suponga que los borrados se hacen en el orden indicado. Indique los criterios en cada borrado.

21.2 Dada la secuencia de claves enteras: $\{190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132, 80\}$ dibuje el árbol AVL.

21.3 Dada la implementación de un ABB. Se define el “árbol común” de dos árboles binarios de búsqueda A y B como aquel árbol C formado por todos los nodos comunes en A y en B que ocupen las mismas posiciones en ambos y sean alcanzables desde la raíz usando nodos comunes. Se pide realizar la implementación de “árbol común”.

21.4 Muestre paso a paso el efecto de insertar la siguiente secuencia comenzando con una estructura de datos inicialmente vacía: $S = \{9, 2, 8, 4, 5, 6, 7, 3, 1, 10\}$ dibuje el:

- árbol AVL
- árbol 2-3

La inserción es en el orden de la secuencia. Debe indicar paso a paso la construcción del árbol solicitado.

21.5 Implemente una función que permita determinar si un árbol ABB (original) es AVL.

21.6 Dados los dos recorridos siguientes, encontrar el árbol binario correspondiente:

- preorden: 1, 2, 4, 6, 3, 5, 7
- inorden: 4, 6, 2, 1, 5, 7, 3

21.7 Se define el valor de trayectoria pesada de una hoja de un árbol binario como la suma del contenido de todos los nodos desde la raíz hasta la hoja multiplicada por el nivel en el que se encuentra.

Implementar una función que, dado un árbol binario, devuelva el valor de trayectoria pesada de cada una de sus hojas.

21.8 ¿Cuál de las siguientes secuencias representa un heap?

- (a) 40, 33, 35, 22, 12, 16, 5, 7.
- (b) 44, 37, 20, 22, 16, 32, 12.
- (c) 15, 15, 15, 15, 15, 15

21.9 Dado un grafo $G(V,E)$ totalmente conectado no dirigido, determinar un algoritmo para encontrar todos los caminos posibles donde visite todos los nodos y vuelva al origen (*Salesman problem*).

Referencias y Recursos

SECCIÓN 22

Referencias

Código Apunte
Código pasado

SUBSECCIÓN 22.1

Figuras

Gayle Laakmann McDowell
Linus Torvalds
Niklaus Wirth
Dennis Ritchie
Internet 2003

SUBSECCIÓN 22.2

Páginas

Curso CC3oa UChile
TikZ
 $\text{\LaTeX}^{}Project$
NotesTex