UNIVERSIDAD TECNOLÓGICA DEL CENTRO DE VERACRUZ

Tecnologías de la Información y Comunicación

TSU en Tecnologías de la Información

M A N U A L D E A S I G N A T U R A

Estructura de datos

Cuitláhuac, Ver.
Diciembre de 2013

Av. Universidad No. 350 Carretera Federal Cuitláhuac – La Tinaja
Localidad Dos Caminos
(278) 732 2050 **C.P. 94910 Cuitláhuac Veracruz**

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

1 Conceptos básicos

1.1 Objetivo de la unidad

El alumno elaborará programas que integren el uso de recursividad y definir estructuras de datos para generar alternativas de programación.

1.2 Introducción de la unidad

En esta unidad clarificaremos la clasificación de los tipos de datos en un lenguaje de programación así como también la recursividad un tema complejo para muchos.

1.3 Tipos de datos abstractos

1.3.1 Objetivo particular

El objetivo de este tema es:

Saber: Describir una Estructura de datos, tipos de datos abstractos, tipos de datos abstractos genéricos.

Saber Hacer: Diferenciar los tipos de datos abstractos y una estructura de datos.

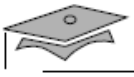
1.3.2 Introducción

Ejemplo en los lenguajes de programación se tienen el tipo entero, real y Booleano. A partir de eso se puede determinar, por ejemplo, que el 5 es un número entero, que true es un valor Booleano, etc. Además, los datos tienen ciertas operaciones y propiedades, por ejemplo sabemos que para números se tienen las operaciones aritméticas y para booleanos las operaciones lógicas. Entre las propiedades de estos datos se cuenta con que los datos forman conjuntos cerrados, por ejemplo una operación entre dos Booleanos devuelve un booleano. Al considerar tanto datos como operaciones se tiene un tipo abstracto de datos.

1.3.3 Tipos de datos primitivos

El lenguaje de programación Java define 8 tipos de datos primitivos

- Logical – boolean
- Textual – char
- Integral – byte, short, int, and long
- Floating – double and float



Logical – boolean

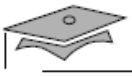
The boolean primitive has the following characteristics:

- The boolean data type has two literals, true and false.
- For example, the statement:

```
boolean truth = true;
```

declares the variable truth as boolean type and assigns it a value of true.

Ilustración 1 Tipo de dato booleano.



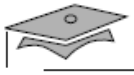
Textual – char

The textual char primitive has the following characteristics:

- Represents a 16-bit Unicode character
- Must have its literal enclosed in single quotes (' ')
- Uses the following notations:

'a'	The letter a
'\t'	The tab character
'\u????'	A specific Unicode character, ????, is replaced with exactly four hexadecimal digits . For example, '\u03A6' is the Greek letter phi [Φ].

Ilustración 2 Tipo de dato char.



Textual – `String`

The textual `String` type has the following characteristics:

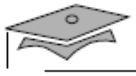
- Is not a primitive data type; it is a class
- Has its literal enclosed in double quotes (" ")

```
"The quick brown fox jumps over the lazy dog."
```

- Can be used as follows:

```
String greeting = "Good Morning !! \n";
String errorMessage = "Record Not Found !";
```

Ilustración 3 Clase String.



Integral – `byte`, `short`, `int`, and `long`

The integral primitives have the following characteristics:

- Integral primates use three forms: Decimal, octal, or hexadecimal

`2`	The decimal form for the integer `2`.
`077`	The leading `0` indicates an octal value.
`0xBAAC`	The leading `0x` indicates a hexadecimal value.

- Literals have a default type of `int`.
- Literals with the suffix `L` or `l` are of type `long`.

Ilustración 4 Tipos de datos numéricos enteros

Sun Educational Services

Integral – byte, short, int, and long

- Integral data types have the following ranges:

Integer Length	Name or Type	Range
8 bits	byte	-2^7 to 2^7-1
16 bits	short	-2^{15} to $2^{15}-1$
32 bits	int	-2^{31} to $2^{31}-1$
64 bits	long	-2^{63} to $2^{63}-1$

Ilustración 5 Rengos de tipos numericos.

Sun Educational Services

Floating Point – float and double

The floating point primitives have the following characteristics:

- Floating-point literal includes either a decimal point or one of the following:
  - E or e (add exponential value)
  - F or f (float)
  - D or d (double)

3.14	A simple floating-point value (a double)
6.02E23	A large floating-point value
2.718F	A simple float size value
123.4E+306D	A large double value with redundant D

Ilustración 6 Tipos de datos numéricos flotantes.

1.3.4 Tipos de datos abstractos

Un tipo abstracto de datos (TAD) es un conjunto de datos con sus operaciones. Se denominan TAD porque nada en la definición indica el cómo se implementan las operaciones (por cierto, puede haber más de una implementación), por tanto en Java para definir TAD se recomienda usar interfaces.

Para completar la descripción de un TAD se debe proporcionar además de la interfaz con las operaciones que se pueden realizar, una descripción de las propiedades que deben tener tales operaciones. Un mecanismo útil para describir estos aspectos no-sintácticos es el uso de pre/post condiciones. Una pre-condición es aquella propiedad que debe satisfacerse antes de la ejecución de un método. Un post-condición especifica propiedades que se deben esperar como consecuencia de la ejecución de un método.
Por su parte, las estructuras de datos tradicionales son mecanismos útiles para almacenar y recuperar (en/de memoria) colecciones de datos. En Java representan una implementación de un TAD particular.

1.3.5 El TAD conjunto

Un conjunto puede verse como un tipo de datos donde se puede almacenar cualquier cantidad de datos sin importar el orden y sin posibilidad de repetición de valores. Las operaciones básicas con conjuntos son: almacenar un dato en él, eliminar un dato, verificar si contiene algún dato particular, determinar si el conjunto está vacío o su tamaño. Además podrían estar las operaciones clásicas de intersección, unión y diferencia. Una forma bonita de definir un TAD es mediante una interfaz, así que
para los conjuntos se puede tener la siguiente:

```
public interface Conjuntable {
public void agregarElemento(Object elem);
public void eliminarElemento(Object elem);
public boolean contieneElemento (Object elem);
public boolean estaVacio();
public int tama~no();
... //Operaciones cl´asicas con conjuntos. Quedan como ejercicio
}
```

Propiedades que se esperan de un conjunto:

- **agregarElemento.** Si el elemento no existe en el conjunto lo agrega al conjunto, aunque no está definida la posición del nuevo elemento. Si el elemento ya se encuentra en el conjunto se dispara una excepción. En caso de ser exitosa la operación el tamaño del conjunto crece en una unidad.

- **contieneElemento.** Regresa true si el elemento está en el conjunto y false en otro caso. Este método no cambia el estado del conjunto.

- **eliminarElemento**. Si el elemento existe en el conjunto lo elimina, en caso contrario dispara la excepción NoSuchElementException. En caso de que la operación sea exitosa se reduce el tamaño del conjunto en una unidad.

- **estaVacio.** Devuelve true si el conjunto está vacío y false en otro caso.

- **tamaño**. Devuelve un entero que indica la cantidad de elementos en el conjunto.

La interfaz es independiente de la implementación. A continuación se presenta una implementación usando arreglos.

```
/**
* Programa que implementa el tipo de datos abstracto conjunto.
* @author Estructura de datos
* @version Agosto 2013.
*/
public class Conjunto implements Conjuntable {
private Object[] datos;
private int indiceFinal;
/** Constructor por omisión, crea un conjunto con una capacidad máxima de
* 20 elementos
*/
public Conjunto() {
datos = new Object[20];
indiceFinal = 0;
}
/**
* Crea un arreglo con capacidad para un conjunto de un máximo determinado
* por el usuario o 20 en caso de proporcionar un número negativo.
* @param tam - entero positivo que determina el tamaño máximo del conjunto
*/

public Conjunto(int tam) {
datos = (tam <= 0) ? new Object [20] : new Object [tam] ;
indiceFinal = 0;
}
/**
* Método que determina si un conjunto no tiene elementos.
* @return boolean - true si el conjunto está vacío y false en otro caso.
*/
public boolean estáVacío(){
return (indiceFinal == 0);
}
/**
* Método que determina la cantidad de elementos en el conjunto
* @return int - cantidad de elementos en el conjunto.
```

```
*/
public int tamaño(){
return indiceFinal;
}
/**
* Método que determina si un elemento dado está en el conjunto
* @param elem - Objeto a buscar
* @return boolean - true si el objeto se encuentra en el conjunto y
* false en otro caso
*/
public boolean contieneElemento (Object elem){
if (!estaVacio())
for (int i = 0; i < indiceFinal; i++)
if (elem.equals(datos[i]))
return true;
return false;
}
/**
* Método que permite borrar un elemento del conjunto
* @param elem - Objeto a eliminar

**/

public void eliminarElemento(Object elem){
int i = 0;
boolean borrado = false;
while (i < indiceFinal && ! borrado)
if (elem.equals(datos[i])){ // recorre los elementos del arreglo
for (int j = i; j < indiceFinal -1; j++)
datos[j] = datos[j+1];
indiceFinal--;
borrado = true;
}
else
i++;
}
/**
* Método que permite agregar un elemento del conjunto, si aún no está
* en éll
* @param elem - Objeto a insertar en el conjunto.
*/
public void agregarElemento(Object elem){
if (! contieneElemento(elem) && indiceFinal < datos.length)
datos[indiceFinal++] = elem;
}
```

1.4 Recursividad

1.4.1 Objetivo particular

El objetivo de este tema es:

Saber: Explicar el concepto de recursividad.

Saber Hacer: Elaborar la codificación de aplicaciones que utilicen recursividad.

1.4.2 Introducción

Primero debemos decir que la recursividad no es una estructura de datos, sino que es una técnica de programación que nos permite que un bloque de instrucciones se ejecute n veces. Remplaza en ocasiones a estructuras repetitivas.

1.4.3 Concepto de recursividad

La recursividad es un concepto difícil de entender en principio, pero luego de analizar diferentes problemas aparecen puntos comunes.
En Java los métodos pueden llamarse a sí mismos. Si dentro de un método existe la llamada a sí mismo decimos que el método es recursivo.
Cuando un método se llama a sí mismo, se asigna espacio en la pila para las nuevas variables locales y parámetros.
Al volver de una llamada recursiva, se recuperan de la pila las variables locales y los parámetros antiguos y la ejecución se reanuda en el punto de la llamada al método.

```
public class Recursividad {

    void repetir() {
        repetir();
    }

    public static void main(String[] ar) {
        Recursividad re=new Recursividad();
        re.repetir();
    }
}
```

Ilustración 7 Ejemplo de un método recursivo.

1.4.4 Ejemplo de método recursivo

La recursividad es una técnica potente de programación que puede utilizarse en lugar de la iteración para resolver determinados tipos de problemas.

Por ejemplo, para escribir un método que calcule el factorial de un número entero no negativo, podemos hacerlo a partir de la definición de factorial:
0! = 1
n! = n * (n–1) * (n–2) * ... * 3 * 2 * 1
si n>0

Esto dará lugar a una solución iterativa en Java mediante un bucle for:

```
public double factorial(int n){
    double fact=1;
    int i;
    if (n==0)
        fact=1;
    else
        for(i=1;i<=n;i++)
            fact=fact*i;
 return fact;
}
```

Ilustración 8 Calculo de factorial mediante método iterativo.

```
public double factorial(int n){
    if (n==0)
        return 1;
    else
        return n*(factorial(n-1));
}
```

Ilustración 9 Calculo de factorial mediante método recursivo

1.4.5 Elementos de un método recursivo

Un método recursivo debe contener:

- Uno o más casos base: casos para los que existe una solución directa.
- Una o más llamadas recursivas: casos en los que se llama sí mismo

1.4.5.1 Caso base

Siempre ha de existir uno o más casos en los que los valores de los parámetros de entrada permitan al método devolver un resultado directo. Estos casos también se conocen como solución trivial del problema.

En el ejemplo del factorial el caso base es la condición:

```
if (n==0)
    return 1;
```

si n=0 el resultado directo es 1 No se produce llamada recursiva

Llamada recursiva: Si los valores de los parámetros de entrada no cumplen la condición del caso base se llama recursivamente al método. En las llamadas recursivas el valor del parámetro en la llamada se ha de modificar de forma que se aproxime cada vez más hasta alcanzar al valor del caso base.

En el ejemplo del factorial en cada llamada recursiva se utiliza n-1

```
return n * ( factorial(n-1) );
```

por lo que en cada llamada el valor de n se acerca más a 0 que es el caso base.

La recursividad es especialmente apropiada cuando el problema a resolver (por ejemplo cálculo del factorial de un número) o la estructura de datos a procesar (por ejemplo los árboles) tienen una clara definición recursiva.

No se debe utilizar la recursión cuando la iteración ofrece una solución obvia. Cuando el problema se pueda definir mejor de una forma recursiva que iterativa lo resolveremos utilizando recursividad.

Para medir la eficacia de un algoritmo recursivo se tienen en cuenta tres factores:

- Tiempo de ejecución
- Uso de memoria
- Legibilidad y facilidad de comprensión

Las soluciones recursivas suelen ser más lentas que las iterativas por el tiempo empleado en la gestión de las sucesivas llamadas a los métodos. Además consumen más memoria ya que se deben guardar los contextos de ejecución de cada método que se llama.

A pesar de estos inconvenientes, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa. En estos casos una mayor claridad del algoritmo puede compensar el coste en tiempo y en ocupación de memoria.

De todas maneras, numerosos problemas son difíciles de resolver con soluciones iterativas, y sólo la solución recursiva conduce a la resolución del problema (por ejemplo, Torres de Hanoi o recorrido de Árboles).

2 Arreglos

2.1 Objetivo de la unidad

El alumno elaborará programas que incluyan métodos de búsqueda y ordenamiento, usando arreglos unidimensionales y bidimensionales para manipular datos de forma organizada

2.2 Introducción de la unidad

Los arreglos se pueden definir como objetos en los que podemos guardar más de una variable, es decir, al tener un único arreglo, este puede guardar múltiples variables de acuerdo a su tamaño o capacidad, es importante recordar que las variables guardadas deben ser del mismo tipo, por ejemplo: Si tenemos un arreglo de tipo Numérico que puede almacenar 10 variables, solo podrá almacenar 10 números diferentes, no otras variables como caracteres o Strings.

2.3 Arreglos unidimensionales y bidimensionales

2.3.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar los diferentes tipos de arreglos y sus características. Identificar la sintaxis para la declaración y creación de arreglos (unidimensionales y bidimensionales).

Saber Hacer: Organizar conjuntos de datos mediante el uso de arreglos unidimensionales y bidimensionales realizando operaciones básicas (inicialización, acceso, impresión y eliminación).

2.3.2 Introducción

En este tema veremos cómo declarar, inicializar e implementar arreglos unidimensionales y bidimensionales de tipos de datos primitivos y abstractos para lo cual serán necesarios los conocimientos adquiridos en la primera unidad.

2.3.3 Declaración de arreglos

- Agrupa datos del mismo tipo
- Declaración de arreglos de tipos de datos primitivos o absatractos
  char s[];
  Point p[];
- Crea un espacio de referencia
- Un arreglo es un objeto por lo mismo es creado con new

2.3.4 Creación de arreglos

Se usa new para crear un arreglo

A continuación un ejemplo de la creación de un arreglo de tipo char.

```
public char[] createArray() {
char[] s;
s = new char[26];
for ( int i=0; i<26; i++ ) {
s[i] = (char) ('A' + i);
}

return s;
}
```

2.3.5 Inicialización de arreglos

La figura 10 muestra la forma en la que se puede inicializar un arreglo

```
String[] names;
names = new String[3];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";
```

```
String[] names = {
    "Georgianna",
    "Jen",
    "Simon"
};
```

```
MyDate[] dates;
dates = new MyDate[3];
dates[0] = new MyDate(22, 7, 1964);
dates[1] = new MyDate(1, 1, 2000);
dates[2] = new MyDate(22, 12, 1964);
```

```
MyDate[] dates = {
    new MyDate(22, 7, 1964),
    new MyDate(1, 1, 2000),
    new MyDate(22, 12, 1964)
};
```

Ilustración 10 Ejemplo de inicialización de arreglos

2.4 Métodos de ordenamiento y búsqueda

2.4.1 Objetivo particular

El objetivo de este tema es:

Saber: Explicar los algoritmos de los métodos de búsqueda (secuencial y binaria) y ordenamiento (burbuja, quick sort, shell, merge sort).

Saber Hacer: Elaborar la codificación de los algoritmos de búsqueda y ordenamiento para resolver casos en un lenguaje de POO.

2.4.2 Introducción

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar. En este caso, nos servirán para ordenar vectores o matrices con valores asignados aleatoriamente. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y revisando el código, escrito en Java, de cada algoritmo.

2.4.3 Métodos iterativos

Estos métodos son simples de entender y de programar ya que son iterativos, simples ciclos y sentencias que hacen que el vector pueda ser ordenado.
Dentro de los Algoritmos iterativos encontramos:
– Burbuja
– Inserción
– Selección
– Shellsort

2.4.3.1 Método burbuja

Como lo describimos en el item anterior, la burbuja más simple de todas es la que compara todos con todos, generando comparaciones extras, por ejemplo, no tiene sentido que se compare con sigo mismo o que se compare con los valores anteriores a él, ya que supuestamente, ya están ordenados.

```
for (i=1; i<LIMITE; i++)
    for j=0 ; j<LIMITE - 1; j++)
        if (vector[j] > vector[j+1])
            temp = vector[j];
            vector[j] = vector[j+1];
            vector[j+1] = temp;
```

Ilustración 11Burbuja simple

```
Bubblesort(int matriz[])
{
        int buffer;
        int i,j;
        for(i = 0; i < matriz.length; i++)
        {
                for(j = 0; j < i; j++)
                {
                        if(matriz[i] < matriz[j])
                        {
                                buffer = matriz[j];
                                matriz[j] = matriz[i];
                                matriz[i] = buffer;
                        }
                }
        }
}
```

Ilustración 12 Burbuja optimizada

2.4.3.2 Inserción y selección

El bucle principal de la ordenación por inserción va examinando sucesivamente todos los elementos de la matriz desde el segundo hasta el n-esimo, e inserta cada uno en el lugar adecuado entre sus predecesores dentro de la matriz.

La ordenación por selección funciona seleccionando el menor elemento de la matriz y llevándolo al principio; a continuación selecciona el siguiente menor y lo pone en la segunda posición de la matriz m y así sucesivamente.

```
Insercion(int matrix[])
{
	int i, temp, j;
	for (i = 1; i < matrix.length; i++)
	{
		temp = matrix[i];
		j = i - 1;
		while ( (matrix[j] > temp) && (j >= 0) )
		{
			matrix[j + 1] = matrix[j];
			j--;
		}
		matrix[j + 1] = temp;
	}
}
```

Ilustración 13 Método de inserción.

```
Seleccion(int[]matrix)
{
	int i, j, k, p, buffer, limit = matrix.length-1;
	for(k = 0; k < limit; k++)
	{
		p = k;
		for(i = k+1; i <= limit; i++)
			if(matrix[i] < matrix[p]) p = i;
		if(p != k)
		{
			buffer = matrix[p];
			matrix[p] = matrix[k];
			matrix[k] = buffer;
		}
	}
}
```

Ilustración 14 Método de selección

2.4.4 Métodos recursivos

Estos métodos son aún más complejos, requieren de mayor atención y conocimiento para ser entendidos. Son rápidos y efectivos, utilizan generalmente la técnica Divide y

vencerás, que consiste en dividir un problema grande en varios pequeños para que sea más fácil resolverlos.
Mediante llamadas recursivas a sí mismos, es posible que el tiempo de ejecución y de ordenación sea más óptimo.

Dentro de los algoritmos recursivos encontramos:

– Ordenamiento por Mezclas (merge)
– Ordenamiento Rapido (quick)

2.4.4.1 Ordenamiento por mezclas

Este algoritmo consiste básicamente en dividir en partes iguales la lista de números y luego mezclarlos comparándolos, dejándolos ordenados.

Si se piensa en este algoritmo recursivamente, podemos imaginar que dividirá la lista hasta tener un elemento en cada lista, luego lo compara con el que está a su lado y según corresponda, lo sitúa donde corresponde.

En la siguiente figura podemos ver cómo funciona:

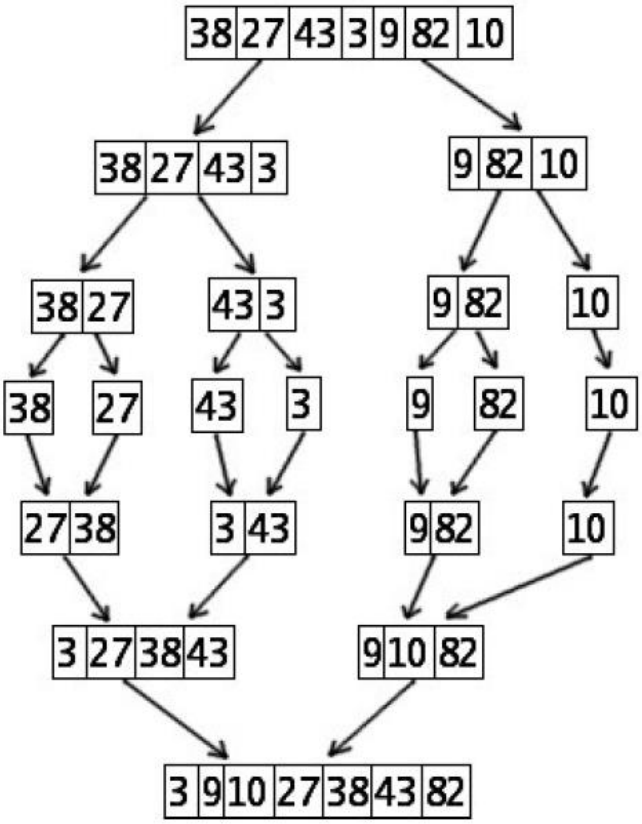


Ilustración 15 Ordenamiento por mezclas

El algoritmo de ordenamiento por mezcla (*Mergesort*) se divide en dos procesos, primero se divide en partes iguales la lista:

```
public static void mergesort(int[ ] matrix, int init, int n)
{
        int n1;
        int n2;
        if (n > 1)
        {
                n1 = n / 2;
                n2 = n - n1;
                mergesort(matrix, init, n1);
                mergesort(matrix, init + n1, n2);
                merge(matrix, init, n1, n2);
        }
}
```

Ilustración 16 Mergesort

Y el algoritmo que nos permite mezclar los elementos según corresponda:

```
private static void merge(int[ ] matrix, int init, int n1, int n2)
{
	int[ ] buffer = new int[n1+n2];
	int temp  = 0;
	int temp1 = 0;
	int temp2 = 0;
	int i;
	while ((temp1 < n1) && (temp2 < n2))
	{
		if (matrix[init + temp1] < matrix[init + n1 + temp2])
			buffer[temp++] = matrix[init + (temp1++)];
		else
			buffer[temp++] = matrix[init + n1 + (temp2++)];
	}
	while (temp1 < n1)
		buffer[temp++] = matrix[init + (temp1++)];
	while (temp2 < n2)
		buffer[temp++] = matrix[init + n1 + (temp2++)];
	for (i = 0; i < n1+n2; i++)
		matrix[init + i] = buffer[i];
}
```

Ilustración 17 Merge

2.4.5 Shellsort

Este método es una mejora del algoritmo de ordenamiento por Inserción (*Insertsort*). Si tenemos en cuenta que el ordenamiento por inserción es mucho más eficiente si nuestra lista de números esta semi-ordenada y que desplaza un valor una única posición a la vez.

Durante la ejecución de este algoritmo, los números de la lista se van casi-ordenando y finalmente, el último paso o función de este algoritmo es un simple método por inserción que, al estar casi-ordenados los números, es más eficiente.

```
public void shellSort(int[] matrix)
{
	for ( int increment = matrix.length / 2; increment > 0; increment =
	(increment == 2 ? 1 : (int) Math.round(increment / 2.2)))
	{
	    for (int i = increment; i < matrix.length; i++)
	  {
	        for (int j = i; j >= increment && matrix[j - increment] >
	         matrix[j]; j -= increment)
	      {
	            int temp = matrix[j];
	            matrix[j] = matrix[j - increment];
	            matrix[j - increment] = temp;
	        }
	    }
	}
}
```

Ilustración 18 Shellsort

2.4.6 Quicksort

Gracias a sus llamadas recursivas, 24teoría24se en la 24teoría de *divide y vencerás.* Lo que hace este algoritmo es dividir recursivamente el vector en partes iguales, indicando un elemento de inicio, fin y un pivote (o comodín) que nos permitirá segmentar nuestra lista. Una vez dividida, lo que hace, es dejar todos los mayores que el pivote a su derecha y todos los menores a su izq. Al finalizar el algoritmo, nuestros elementos están ordenados.

Por ejemplo, si tenemos 3 5 4 8 básicamente lo que hace el algoritmo es dividir la lista de 4 elementos en partes iguales, por un lado 3, por otro lado 4 8 y como comodín o pivote el 5. Luego pregunta, 3 es mayor o menor que el comodín? Es menor, entonces lo deja al lado izq. Y como se acabaron los elementos de ese lado, vamos al otro lado. 4 Es mayor o menor que el pivote? Menor, entonces lo tira a su izq. Luego pregunta por el 8, al ser mayor lo deja donde está, quedando algo así:

3 4 5 8

En esta figura se ilustra de mejor manera un vector con más elementos, usando como pivote el primer elemento:

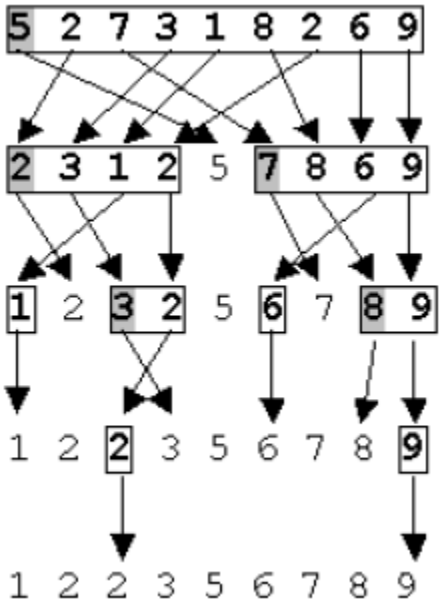


Ilustración 19 Quicksort

```
public void _Quicksort(int matrix[], int a, int b)
{
	this.matrix = new int[matrix.length];
	int buf;
	int from = a;
	int to = b;
	int pivot = matrix[(from+to)/2];
	do
	{
		while(matrix[from] < pivot)
		{
			from++;
		}
		while(matrix[to] > pivot)
		{
			to--;
		}
		if(from <= to)
		{
			buf = matrix[from];
			matrix[from] = matrix[to];
			matrix[to] = buf;
			from++; to--;
		}
	}while(from <= to);
	if(a < to)
	{
		_Quicksort(matrix, a, to);
	}
	if(from < b)
	{
		_Quicksort(matrix, from, b);
	}
	this.matrix = matrix;
}
```

Ilustración 20 Implementación quicksort.

3 Listas

3.1 Objetivo de la unidad

El alumno elaborará programas usando listas para manipular datos de forma organizada.

3.2 Introducción de la unidad

Una lista está formada por una serie de elementos llamados nodos los cuales son objetos que contiene como variable miembro un puntero asignado y variables de cualquier tipo para manejar datos.

El puntero sirve para enlazar cada nodo con el resto de nodos que conforman la lista. De esto podemos deducir que una lista enlazada (lista) es una secuencia de nodos en el que cada nodo esta enlazado o conectado con el siguiente (por medio del puntero mencionado anteriormente). El primer nodo de la lista se denomina cabeza de la lista y el último nodo cola de la lista. Este último nodo suele tener su puntero igualado a NULL Para indicar que es el fin de la lista.

3.3 Definición de listas

3.3.1 Objetivo particular

El objetivo de este tema es:

Saber: Explicar el concepto de lista, sus características y terminología.

Saber Hacer: Determinar el uso de la estructura de datos lista con respecto a un arreglo.

3.3.2 Introducción

La lista enlazada es una estructura de datos dinámica cuyos nodos suelen ser normalmente registros y que tienen un tamaño fijo. Ahora bien suelen llamarse estructuras dinámicas porque se crean y destruyen según se vayan necesitando. De este modo se solicita o libera memoria en tiempo de ejecución del programa.

3.3.3 Representación de una lista enlazada

Una lista enlazada se puede representar de la siguiente manera

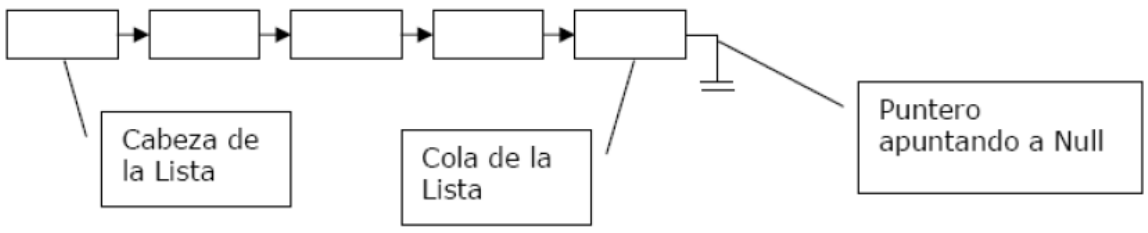


Ilustración 21 Lista enlazada

3.4 Tipos de listas

3.4.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar los diferentes tipos de listas y sus componentes.

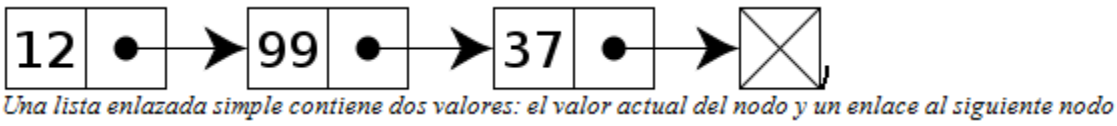
Saber Hacer: Determinar en qué casos es pertinente utilizar los diferentes tipos de listas

3.4.2 Introducción

En esta unidad explicaremos en que consiste las listas simplemente enlazadas, listas doblemente enlazadas, listas circulares simplemente enlazadas y las listas circulares doblemente enlazadas.

3.4.3 Listas simplemente enlazadas

La lista enlazada básica es la **lista enlazada simple** la cual tiene un enlace por nodo. Este enlace apunta al siguiente nodo en la lista, o al valor NULL o a la lista vacía, si es el último nodo.

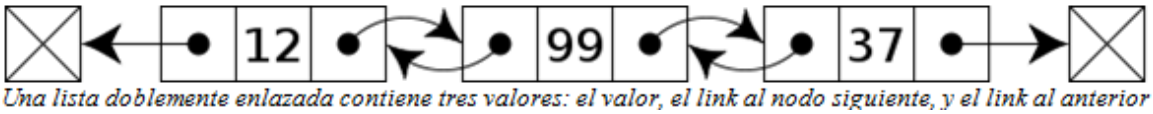


Una lista enlazada simple contiene dos valores: el valor actual del nodo y un enlace al siguiente nodo

Ilustración 22 Lista simplemente enlazada

3.4.4 Lista doblemente enlazadas

Un tipo de lista enlazada más sofisticado es la **lista doblemente enlazada** o **lista enlazadas de dos vías**. Cada nodo tiene dos enlaces: uno apunta al nodo anterior, o apunta al valor NULL si es el primer nodo; y otro que apunta al nodo siguiente, o apunta al valor NULL si es el último nodo.



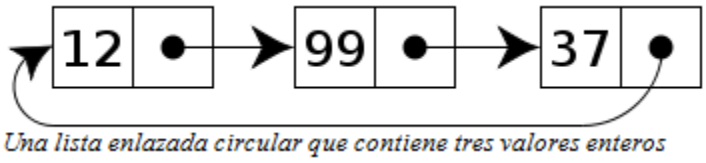
Una lista doblemente enlazada contiene tres valores: el valor, el link al nodo siguiente, y el link al anterior

Ilustración 23 Lista doblemente enlazada

3.4.5 Lista circulares

En una lista enlazada circular, el primer y el último nodo están unidos juntos. Esto se puede hacer tanto para listas enlazadas simples como para las doblemente enlazadas. Para recorrer una lista enlazada circular podemos empezar por cualquier nodo y seguir la

lista en cualquier dirección hasta que se regrese hasta el nodo original. Desde otro punto de vista, las listas enlazadas circulares pueden ser vistas como listas sin comienzo ni fin. Este tipo de listas es el más usado para dirigir buffers para “ingerir” datos, y para visitar todos los nodos de una lista a partir de uno dado.



Una lista enlazada circular que contiene tres valores enteros

Ilustración 24 Listas circulares

3.5 Construcción y operaciones con listas

3.5.1 Objetivo particular

El objetivo de este tema es:

Saber: Explicar la sintaxis para la creación de los distintos tipos de listas y sus elementos, utilizando el paradigma orientado a objetos.

Identificar las operaciones que se pueden realizar con listas (inserción, eliminación y, acceso).

Saber Hacer: Elaborar listas y sus operaciones desde un enfoque orientado a objetos

3.5.2 Introducción

En esta unidad explicaremos cada una de las operaciones que se deben realizar en las listas enlazadas

3.5.3 Construcción de listas

Las listas las tenemos en muchos casos en la vida real: lista de compras, lista de útiles escolares, lista de invitados, lista de pendientes, lista de tareas, lista de discos, etc. Pero, ¿qu´e es una lista? Una lista es una estructura de datos con la propiedad que se pueden agregar y suprimir datos en cualquier lugar.

3.5.4 Operaciones con listas

El tipo de datos Lista debe tener las siguientes propiedades:

• Determinar si está vacía la lista.

• Limpiar la lista.

• Agregar un elemento.

• Encontrar un elemento. (por valor).

• Actualizar un elemento.

• Eliminar un elemento.

Como características de las listas se tiene que de antemano se desconoce cuántos elementos tendrá ésta. Es irrelevante el orden de los elementos y el manejo de ellos, es decir las inserciones y supresiones pueden ser de cualquier dato no por posición.

La interfaz de una lista podría definirse como sigue:

```
interface Listable {

public boolean estaVacia();

public void limpiar();

public Object primerElemento();

public void agregar(Object x); //Inserta al inicio de la lista

public boolean contiene(Object x);

public void eliminar(Object x); //Borra el primer nodo que tenga valor x

public void sustituir(Object orig, Object nuevo);

public java.util.Iterator elementos();

}
```

• estaVacia. Devuelve true si la lista está vacía y false en otro caso.

• limpiar. Elimina todos los elementos de la lista, es decir el tamaño de la lista después de esta operación es cero.

• primerElemento. Devuelve el valor almacenado en el primer elemento de la lista. El estado de la lista no se ve alterado.

• agregar. Agrega al inicio de la lista el objeto especificado como el parámetro. El tamaño de la lista crece en una unidad.

• contiene. Regresa true si el elemento está en la lista y false en otro caso. Este método no cambia el estado de la lista.

• eliminar. Si el elemento existe en la lista lo elimina, en caso contrario dispara la excepción NoSuchElementException. En caso de que la operación sea exitosa se reduce el tamaño de la lista en una unidad.

• sustituir. Si el elemento a sustituir se encuentra en la lista lo sustituye por el segundo parámetro, en caso contrario dispara la excepción NoSuchElementException. Esta operación no altera el tamaño de la lista.

4 Pilas

4.1 Objetivo de la unidad

El alumno elaborará programas usando pilas para manipular datos de forma organizada.

4.2 Introducción de la unidad

Una **estructura de datos** es un conjunto de variables, quizá de tipos distintos, que se relacionan entre sí y que se pueden operar como un todo.

Son fundamentales para el manejo de información y el desarrollo de sistemas.

Varían en la forma como permiten el acceso a los miembros del conjunto y algunas imponen restricciones.

4.3 Definición de pilas

4.3.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar el concepto de pila, sus características y su terminología.

Saber Hacer: Demostrar el uso de la estructura de datos pila.

4.3.2 Introducción

Una **pila** es una lista de elementos en la cual los elementos se insertan o se eliminan sólo por uno de los extremos.

Una pila (**stack**) es una estructura tipo LIFO (Last In First Out), así que el primero en entrar es el último en salir.

Por ejemplo, en una cafetería de autoservicio donde se tienen charolas, las cuales se encuentran apiladas.

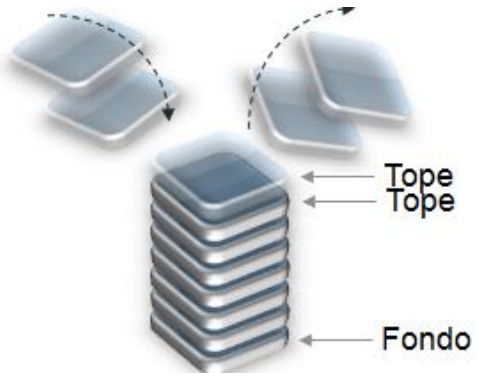


Ilustración 25 Ejemplo de pila

4.4 Tipos de implementación

4.4.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar las formas en que se puede implementar una pila (arreglos y listas).

Saber Hacer: Determinar en qué casos es pertinente utilizar la estructura de datos pila.

4.4.2 Introducción

Las pilas no están definidas como tales en los lenguajes de programación, se representan mediante el uso de Arreglos o Listas ligadas.

Para el manejo de la pila se requiere de un apuntador (**Tope**) al último elemento almacenado en la pila.

Este apuntador se mueve sobre el arreglo, hacia arriba o hacia abajo, según la pila crezca o decrezca.

Debido a que se maneja memoria estática, el **Tope** se mueve sobre el arreglo.

4.5 Operaciones con pilas

4.5.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar la sintaxis de las operaciones de una pila (push, pop, is_empty, full).

Saber Hacer: Elaborar pilas desde un enfoque orientado a objetos aplicando sus operaciones.

4.5.2 Introducción

En una pila se pueden hacer dos operaciones PUSH y POP las cuales en este tema se explican

4.5.3 Push

Consiste en ingresar un elemento a la pila las operaciones que se deben realizar al ingresar un elemento son las siguientes:

- Introducir el dato.

- Incrementar el **Tope**.
- ¿Qué pasa si la pila está llena y se intenta introducir algún dato?

4.5.4 Pop

Consiste en retirar un elemento de la pila, las operaciones que se deben realizar al retirar un elemento de la pila son las siguientes:

- Decrementar el **Tope**.
- Sacar el dato.
- ¿Qué pasa si la pila está vacía y se intenta sacar algún dato?

4.5.5 Ejercicios propuestos

1. Elabore un programa que permita simular el funcionamiento de una estructura tipo Pila que contenga los métodos: inserta_pila, sacar_pila, y mostrar_pila.

2. Lea una palabra por el teclado y determine si la palabra es palíndromo”

3. implemente el juego algoritmo iterativo para el juego de las torres de Hanoi (use pilas para simular la recursividad).

4. Leer una expresión aritmética en notación infija y conviértala a notación postfija y prefija.

5. Leer una expresión aritmética en notación post fija y obtenga su valor numérico. (Complemente le programa del ejercicio anterior).

6. Leer una frase y luego invierta el orden de las palabras en la frase. Por Ejemplo: “una imagen vale por mil palabras” debe convertirse en “palabras mil por vale imagen una”.

7. Simular la operación de n pilas operando simultáneamente y donde se saca y/o inserta elementos al azar a cualquiera de las pilas. Determine cuál es la pila de mayor trabajo y cuál es la pila de menos trabajo en un tiempo determinado de operación.

8. Usando pilas efectúe operaciones de suma y resta de dos números de más de 10 dígitos.

9. Mediante el uso de pilas efectúe la operación de división de dos números bastante grandes (más de 10 dígitos). Para ello utilice el método de restas sucesivas para efectuar la división.

10. En un almacén se guarda mercadería en contenedores. No es posible colocar más de n contenedores uno encima del otro y, no hay área para más de m pilas de contenedores. Cada contendor tiene un número y un nombre de la empresa propietaria. Elabore un programa que permita gestionar el ingreso y salida de contenedores. Note que para retirar

un contenedor es necesario retirar los contenedores que están encima de él y colocarlos en otra pila.

11. Elabore un programa que lea un archivo .CPP y determine si los símbolos { } , [ ] y ( ) estén correctamente balanceados. Si no se encuentra balanceado, que muestre el error indicando el símbolo faltante.

12. Se desea implementar dos pilas, y se dispone de un solo vector de N componentes. Implementar ambas pilas de manera que se pueda aprovechar al máximo el vector. Las operaciones de pila tendrán que llevar un parámetro adicional que indique sobre qué pila se quiere realizar la operación (pila 1 o pila 2).

Nota: Las dos pilas crecen partiendo de los extremos del arreglo

13. Implementar las mismas operaciones que se indican en el ejercicio 1 pero ahora utilice listas enlazadas.

14. Se tienen dos pilas (stacks) que contienen números enteros; la primera ordenada ascendentemente desde el tope hacia el fondo, y la segunda ordenada descendentemente desde el tope hacia el fondo. Si se cuenta con la clase CPila que contiene las operaciones básicas definidas para pilas, elabore un programa que fusione ambas pilas en una tercera ordenada descendentemente desde el tope hacia el fondo.

5 Colas

5.1 Objetivo de la unidad

El alumno elaborará programas usando colas para manipular datos de forma organizada.

5.2 Introducción de la unidad

Una **cola** es una lista de elementos en la cual los elementos se insertan por un extremo y se eliminan por otro.

5.3 Definición de colas

5.3.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar el concepto de cola, sus características y terminología.

Saber Hacer: Ilustrar el uso de la estructura de datos cola.

5.3.2 Introducción

Una cola (**queue**) es una estructura tipo FIFO (First In First Out), así que el primero en entrar es el primero en salir. Aquí los elementos salen en el mismo orden en que entraron.

Diariamente se tienen colas, como por ejemplo en supermercados, teatros, bancos, etc.

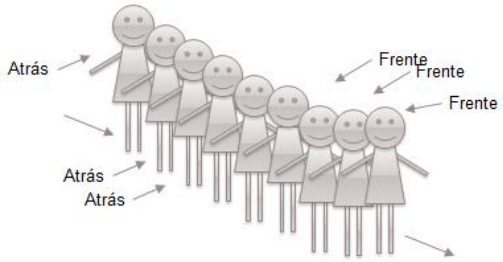


Ilustración 26 Ejemplo de cola

5.4 Tipos de implementación

5.4.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar las formas en que se puede implementar una cola (arreglos y listas).

Saber Hacer: Determinar en qué casos es pertinente utilizar la estructura de datos cola.

5.4.2 Introducción

Las colas no están definidas como tales en los lenguajes de programación, éstas se representan mediante el uso de Arreglos o Listas ligadas.

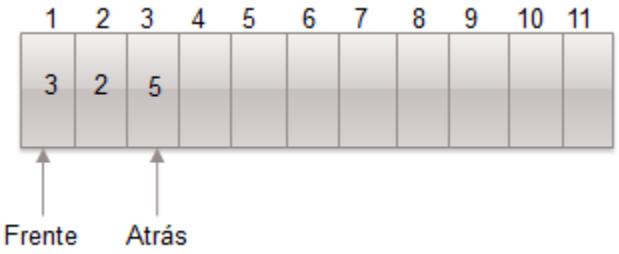


Ilustración 27 Representación de una cola haciendo uso de arreglos

5.5 Operaciones con colas

5.5.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar la sintaxis de las operaciones de una cola (Inserción y extracción).

Saber Hacer: Elaborar colas desde un enfoque orientado a objetos aplicando sus operaciones.

5.5.2 Introducción

En una cola se pueden llevar a cabo dos operaciones:

- Insertar: meter dato en la cola.
- Eliminar: sacar dato de la cola.

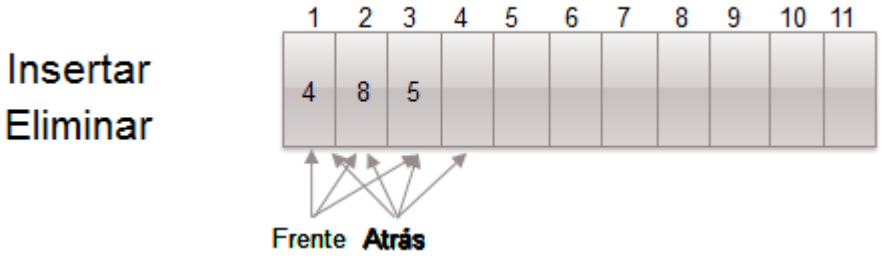


Ilustración 28 Operaciones de una cola

5.5.3 Insertar

Consiste en insertar un elemento al final de la cola

5.5.4 Eliminar

Consiste en sacar un elemento al principio de la cola.

5.5.5 Ejercicios propuestos

1. Se tiene una lista con los datos de los clientes de una compañía de telefonía celular, los cuales pueden aparecer repetidos en la lista, si tienen registrado más de un número telefónico. La compañía para su próximo aniversario desea enviar un regalo a sus clientes, sin repetir regalos a un mismo cliente. Los regalos se encuentran almacenados en una pila de regalos. Se desea elaborar un programa en C++ que permita generar una nueva estructura donde los clientes aparezcan sólo una vez con sus regalos asignados.

2. Escribir un programa que invierta el contenido de una cola. Usted puede utilizar estructuras de datos auxiliares para hacerlo.

3. Una matriz de N-filas puede ser vista como N-colas consecutivas, donde la operación de introducir un elemento en la cola, debería recibir el elemento a introducir y el identificador de la cola i donde se desea meter el elemento. Elabore un método que permita implementar la operación inserta_cola en una sucesión de N-colas en un objeto matriz NxM. M es la capacidad máxima de cada cola.

4. Implemente el objeto Cola en C++ de manera que reciba los datos de personas en una cola de un banco, esto es, nombre y el tipo de transacciones a realizar. Se requiere conocer el tiempo estimado de permanencia de cualquier persona en la cola, si se conocen los tiempos estimados para cada tipo de transacción:

Retiro	4 min	
Depósito		2min
Consulta		3.5min
Actualización	5 min	
Pagos	2 min	

5. Elabore un programa en C++ que simule el funcionamiento de una estructura de datos tipo Cola_Circulary otra estructura tipo Cola_Prioritaria. Para el primer caso use arreglos y para el segundo caso use listas enlazadas. Para el caso de las colas prioritarias asigne un nivel de prioridad de 1, 2 o 3. El nivel 1 indica mayor prioridad y 3 la menor prioridad. El programa debe contener los métodos Inserta_Circular, Elimina_Citrcular, Mostrar_Cola, Inserta_Priorit, Elimina_Priorit.

6. Unos vehículos blindados intentan pasar un puente defectuoso. Para ello forman un cola para atravesarlo y la probabilidad de éxito al momento de cruzar e puente es de 0.9 al inicio. Cada vez que un vehículo entra al puente, éste se deteriora más y la probabilidad de éxito se reduce en 0.06. Para un total de *n* vehículos blindados, ¿cuantos lograron atravesar el puente? ,¿Cuántos cayeron en el intento?

6 Arboles

6.1 Objetivo de la unidad

El alumno elaborará programas usando árboles para manipular datos de forma organizada.

6.2 Introducción de la unidad

6.3 Definición y tipo de arboles

6.3.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar el concepto de árbol binario, binario balanceado, de búsqueda y general, sus características y terminología.

Saber Hacer: Demostrar el uso de la estructura de datos árbol y sus tipos.

6.3.2 Introducción

6.4 Arboles binarios, balanceados y búsqueda

6.4.1 Objetivo particular

El objetivo de este tema es:

Saber: Identificar los casos en los que es pertinente utilizar los árboles binarios, binarios balanceados y de búsqueda.

Identificar las operaciones para arboles binarios, binarios balanceados, y de búsqueda: inserción, eliminación, búsqueda (profundidad, amplitud) y recorridos (preorden, inorden y postorden)

Saber Hacer: Elaborar árboles binarios, binarios balanceados y binarios de búsqueda, desde un enfoque orientado a objetos, usando las operaciones básicas, resolviendo problemas que utilicen este tipo de estructura.

6.4.2 Introducción

En ciencias de la computación, un árbol binario es una estructura de datos en la cual cada nodo siempre tiene un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre '**Binario**'). Si algún hijo tiene como referencia a **null**, es decir que no almacena ningún dato, entonces este es llamado un nodo externo. En el caso contrario el hijo es llamado un nodo interno.

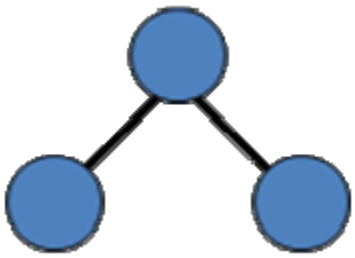


Ilustración 29 Representación de un árbol binario

6.4.3 Terminología

Nodo: Cada elemento en un árbol.

Nodo Raíz: Primer elemento agregado al árbol.

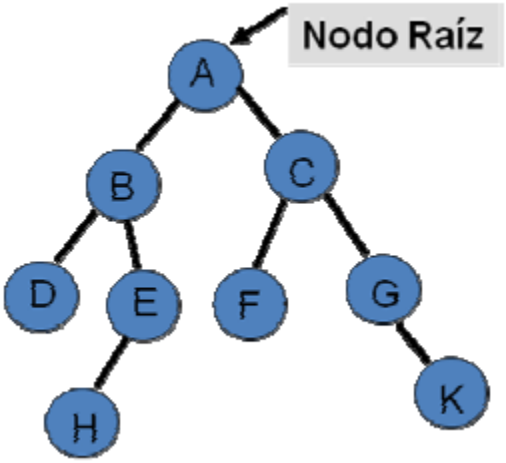


Ilustración 30 Árbol binario

Nodo Padre: Se le llama así al nodo predecesor de un elemento

Nodo Hijo: Es el nodo sucesor de un elemento.

Nodo Hermano: Nodos que tienen el mismo nodo padre.

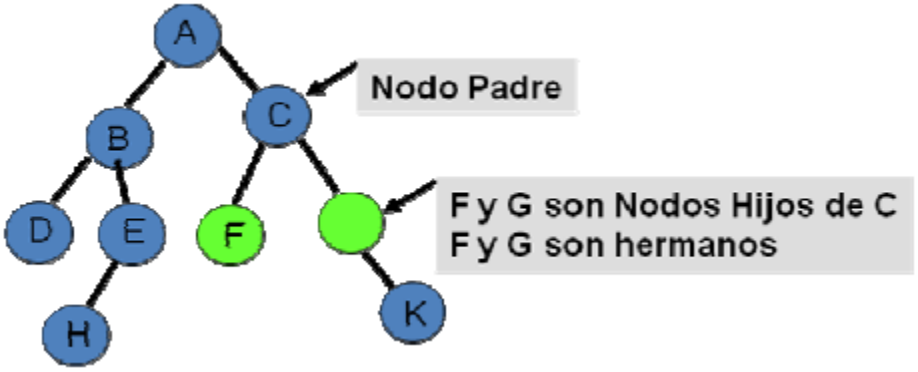


Ilustración 31 Nodo padre, hijos, hermanos.

Nodo Hoja: Aquel nodo que no tiene hijos.

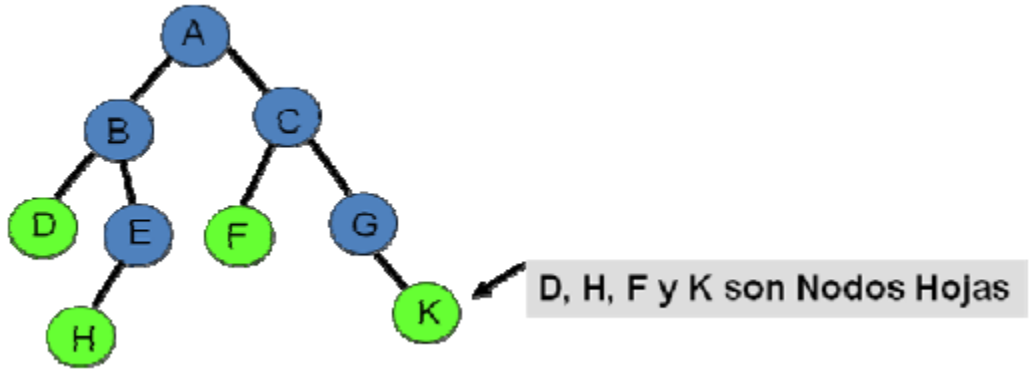


Ilustración 32 Nodos hojas

6.4.4 Altura y niveles

La altura es la cantidad de niveles.

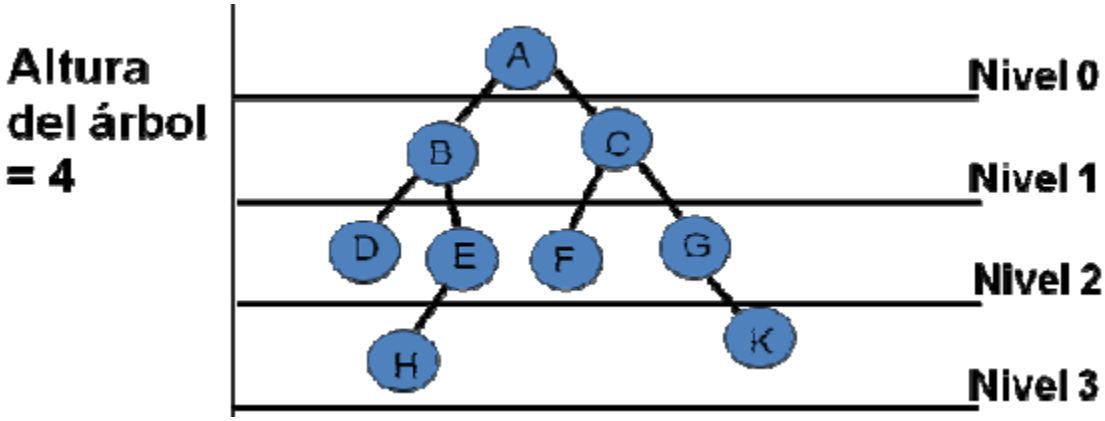


Ilustración 33 Altura y niveles.

6.4.5 Recorrido de un árbol

Recorrer el árbol significa que cada nodo sea procesado una vez en una secuencia determinada.

Existen dos enfoques generales:

- **Recorrido en Profundidad**: El proceso exigen alcanzar las profundidades de un camino desde la raíz hacia el descendiente más lejano del primer hijo, antes de proseguir con el segundo.

- **Recorrido en Anchura**: El proceso se realiza horizontalmente desde la raíz a todos sus hijos antes de pasar con la descendencia de algunos de ellos.

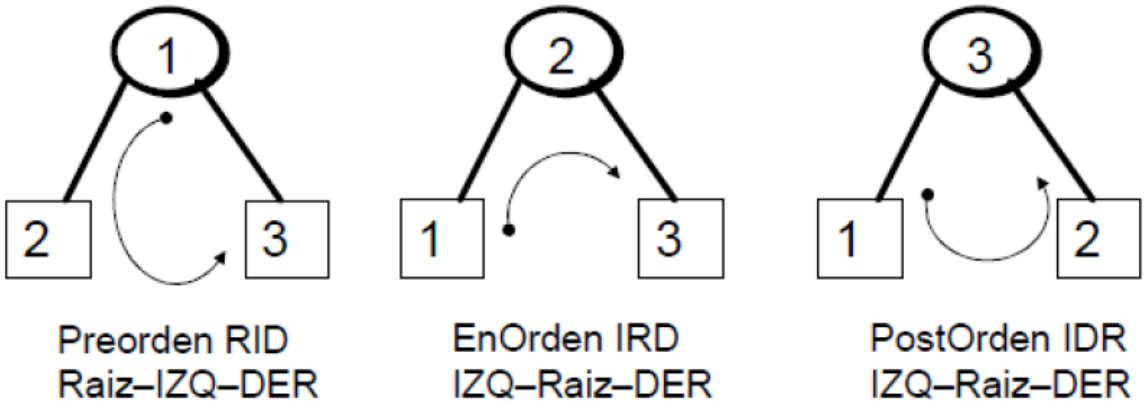


Ilustración 34 Recorrido de un árbol.

Bibliografía

Aho, A. H. (1988). *Estructuras de datos y algoritmos.* Delaware: Adison wesley Iberoamerica.

McMillan, M. (2007). *Data Structures and Algorithms.* Nueva york: Cambridge University Press.

Weiss, M. (2000). *Estructuras de datos en java.* Madrid: Addison Wesley.