ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/336316390

Data Structures and Algorithms with C/C++" (In Spanish: Algorithms and Data Structures with C/C++).

Book · January 2007

DOI: 10.6084/m9.figshare.9947357

CITATIONS
0

READS
1,740

1 author:

Mario José Diván
Intel

107 PUBLICATIONS **322** CITATIONS

SEE PROFILE

All content following this page was uploaded by Mario José Diván on 08 October 2019.

The user has requested enhancement of the downloaded file.

Reconocimientos y Agradecimientos

Como todos los aspectos de la vida y como correctamente se concibe en la teoría general de sistemas, ningún resultado es un resultado por el resultado mismo, sino que dicho resultado está en función de un conjunto de variables que inciden directa e indirectamente en las distintas etapas de nuestras vidas para sí, finalmente, lograr dicho resultado. Ustedes sabrán disculpar que en numerosas oportunidades acuda a los conceptos matemáticos para explicar más de un aspecto y concepto, pero me parece una forma universal de expresar una idea en un sentido relativamente puro.

Este material no hubiese sido posible sin el apoyo incondicional y la eterna paciencia de Laura, que día a día me motiva a seguir creciendo y progresando; A mi Mamá, que me ha enseñado lo que es la esperanza y la perseverancia; A mi Papá, que me ha enseñado el valor de las cosas; A mi Hermana Sandra, que su punto de vista es realmente impredecible y me sorprende día a día; Al apoyo incondicional de las autoridades de la Facultad de Ciencias Económicas y Jurídicas, por su pro actividad a la investigación y desarrollo de nuevos conocimientos e investigadores; al personal no docente de la Facultad de Ciencias Económicas y Jurídicas, que con su trabajo ayudan y facilitan el desarrollo del nuevo conocimiento.

Índice

Reconocimientos y Agradecimientos **1**
Índice **2**
Prólogo **7**

Capítulo 1: Introducción a C/C++

Introducción a C/C++ **9**
Introducción 10
Primeros Pasos en C/C++ 11
Mi Primer Programa 11
Bibliotecas 13
Archivos de Cabecera 14
Entrada/Salida por Pantalla 15
Comentarios 18
Variables 19
Concepto 19
Tipos de Datos 19
Caracteres 20
Enteros 20
Flotantes 21
Flotantes de Doble Precisión 22
Enumeraciones 23
Void 24
Introducción al Concepto de Punteros 25
Ciclo de Vida de una Variable 26
Conversión de Tipos 28
Modificadores de Tipo de Dato 30
Constantes 32
Operadores 34
Operadores Aritméticos 34
Operadores de Incremento y Decremento 35
Operadores de Relación y Lógicos 35
Operadores para Manejo de Bits 36
Operadores de Asignación y Expresiones 36
Pre Incremento y Post Incremento 37
Ejemplo Integrador de Operadores 38
Consideraciones sobre el Manejo a Nivel de Bits 40
Operación And Lógico 41
Operación Or Lógico 42
Operación Or Exclusivo Lógico 42
Operación Desplazamiento a la Izquierda 43
Operación Desplazamiento a la Derecha 43
Operación Complemento a Uno 44
Funciones 45
Concepto 45

Implementación de Funciones en C/C++ ... 46
Tipo de Retorno ... 47
Valores por Defecto en los Parámetros de Funciones ... 48
Lista de Parámetros Variables ... 49
Paso de Parámetros ... 50
Ejemplo Integrador ... 52
Cadenas de Caracteres ... 54
Concepto de Cadenas de Caracteres ... 54
Recuperación y Asignación de Caracteres Individuales ... 55
Inicialización de una Cadena de Caracteres ... 56
Copiar Cadenas de Caracteres ... 58
Comparar dos Cadenas de Caracteres ... 58
Tamaño de una Cadena de Caracteres ... 59
Concatenar Cadenas de Caracteres ... 60
Estructuras de Control ... 62
Estructuras Condicionales ... 62
Estructura Condicional de Bifurcación ... 62
Estructura Condicional de Asignación ... 65
Estructuras Repetitivas ... 67
Ciclo For ... 67
Ciclo While ... 71
Ciclo Do-While ... 73
Estructuras Selectivas ... 75
Switch ... 75
Tipos de Datos Definidos por el Usuario ... 76
Concepto de Tipo de Dato Definido por el Usuario ... 77
Definición de un Tipo de Datos definido por el usuario ... 78
Lectura y Asignación de Valores a un Tipo de Datos Definido por el Usuario ... 79
Tipos de Datos Definidos por el Usuario sin Límites ... 82
Tipos de Datos Definidos por el Usuario y Punteros ... 84
Inicialización de los Campos de un Tipo de Datos Definido por el Usuario ... 87
Gestión Dinámica de Memoria ... 88
Concepto ... 88
Implementación de Gestión Dinámica en C/C++ ... 90
Gestión Dinámica de Arreglos ... 92
Gestión Dinámica de Tipos de Datos Definidos por el Usuario ... 95
Referencias ... 97

Capítulo 2: Recursividad

Recursividad ... 98
Concepto de Recursividad ... 99
Problemas de la Recursividad ... 101
Análisis de la Condición de Corte Recursivo Mal Definida ... 102
Análisis de la Condición de Corte Bien Definida pero Mal Ubicada ... 104
Eliminación de la Recursión ... 105
Búsqueda Binaria Recursiva ... 106
Exponenciación Recursiva ... 109
Exponenciación No Recursiva ... 109

Mostrar una Cadena de Caracteres Mediante Aritmética de Punteros y Algoritmos Recursivos 110
Referencias 112

Capítulo 3: Análisis de Algoritmos

Análisis de Algoritmos 113
Marco de Referencia del Análisis de Algoritmos 114
Clasificación de los Algoritmos 115
Complejidad del Cálculo 116
Análisis del Caso Medio 117
Resultados Aproximados y Asintóticos 118
Análisis de Algoritmos Recursivos 119
Casos Prácticos con Análisis 119
Ordenamiento por Burbuja 119
Ordenamiento por Inserción 123
Referencias 126

Capítulo 4: Estructuras de Datos Simples

Estructuras de Datos Simples 127
Introducción a las Estructuras de Datos 128
Listas 128
Concepto 128
Implementación de Listas Simplemente Enlazadas 129
Implementación de una Lista Ordenadas Simplemente Enlazada 131
Implementación de una Lista Circular 134
Implementación de una Lista Doblemente Enlazada 138
Colas 141
Implementación de una Cola a través de Listas Simplemente Enlazadas 141
Pilas 145
Implementación de una Pila a través de Listas Simplemente Enlazadas 145
Árboles Binarios 148
Introducción 148
Implementación de un Árbol Binario 151
Análisis del Algoritmo de Inserción 156
Análisis del Algoritmo de Borrado 157
Análisis del Algoritmo de Muestra 159
Referencias 161

Capítulo 5: Algoritmos de Ordenamiento

Algoritmos de Ordenamiento 162
Introducción a los Algoritmos de Ordenamiento 164
Shellsort 164
Quicksort 169
Heapsort 178
Combsort 189
Intercambio de Residuos 193

Complemento a Uno ... 193
Operaciones Binarias Aplicables al Ordenamiento por Residuo ... 193
Referencias ... 202

Capítulo 6: Algoritmos y Estructuras para Búsqueda

Algoritmos y Estructuras para Búsqueda ... 203
Introducción a los Algoritmos y Estructuras para Búsqueda ... 204
Árboles AVL ... 205
Análisis de la Inserción de Valores en un árbol AVL ... 207
Análisis del Borrado de Valores en un árbol AVL ... 213
Implementación de un Árbol AVL en C/C++ Estructurado ... 216
Árboles Multidireccionales ... 224
Introducción ... 224
Conceptos Previos ... 224
Preferencia por el Orden ... 227
Estructura de Soporte al Nodo de un Árbol Multidireccional ... 229
Recorrido de un Árbol Multidireccional ... 230
Búsqueda en un Árbol Multidireccional ... 231
Algoritmo de Inserción ... 233
Algoritmo de Borrado ... 235
Implementación de un Árbol de Búsqueda Multidireccional ... 237
Algoritmo de Rotación en Árboles Multidireccionales ... 246
Árboles B ... 260
Algoritmo de Inserción en Árboles B ... 260
Borrado Lógico en Árboles B ... 263
Liberación de la Memoria del Árbol B ... 280
Árbol B^+ ... 280
Hashing ... 283
Introducción ... 283
Implementación de Hash Estático ... 284
Tratamiento de las Colisiones ... 287
Redispersión ... 289
Barrido Secuencial ... 293
Encadenamiento Externo ... 298
Algritmo de Inserción ... 298
Algoritmo de Borrado ... 301
Implementación del Encadenamiento Externo ... 303
Referencias ... 308

Capítulo 7: Búsqueda de Cadenas y Reconocimiento de Patrones

Búsqueda de Cadenas y Reconocimientos de Patrones ... 309
Introducción a la Búsqueda de Cadenas y Recnonocimientos de Patrones ... 310
Algoritmo de Fuerza Bruta ... 311
Algoritmo de Knuth-Morris-Pratt ... 314
Algoritmo de Boyer-Moore ... 318
Cálculo del Desplazamiento a Derecha del Patrón ... 318
Análisis del Funcionamiento Lógico del Algoritmo ... 320
Implementación del Algoritmo de Boyer-Moore en C/C++ ... 321

Introducción al Reconocimiento Teórico de Patrones 322
Máquinas de Estados para el Reconocimiento de Patrones 325
Aplicación de AFD al reconocimiento de Patrones 327
Referencias 332

Capítulo 8: Compresión de Datos

Compresión de Datos 333
Introducción a la Compresión de Datos 334
Compresión por Codificación de Longitud de Series 334
Implementación del Algoritmo de Compresión por Longitud de Series 336
Compresión de Longitud Variable 340
Algoritmo de Huffman 342
Implementación del Algoritmo de Huffman 345
Referencias 354

Capítulo 9: Criptología

Criptología 355
Introducción a la Criptología 356
Algoritmo de Encriptación: “Cifra del César” 357
Algoritmo de Vigenere 358
Algoritmo de Cifrado XOR 361
Algoritmo de Cifrado por Rotación de Bits 363
Algoritmo de Encriptación Por Tablas 366
Algoritmo de Encriptación Base64 369
Referencias 387

Capítulo 10: Introducción a la Teoría de Grafos

Introducción a la Teoría de Grafos 388
Introducción al Concepto de Grafos 389
Matriz de Adyacencia 392
Cierre Transitivo 393
Implementación del Cierre Transitivo en C/C++ 395
Aplicación de la Teoría de Grafos: Determinación de la Ruta Más Corta 398
Implementación del Algoritmo de Dijkstra 402
Referencias 404

Prólogo

Este libro surge con el objetivo de servir como soporte a la enseñanza inicial de los Algoritmos y Estructuras de Datos para el desarrollo de software. Como todo proceso de aprendizaje y más aún, en un área donde predomina lo abstracto, el alumno necesita visualizar el inicio y final del concepto. Esto es, desde el planteo conceptual de la idea o problemática, hasta la implementación concreta de alguna forma alternativa de solución al concepto o problemática. Esta idea, motivó a que el texto se plantee de dicho modo, esquematizando los conceptos en forma escrita y gráfica y continuando luego con la implementación en C/C++. Así, el alumno puede realizar la trazabilidad desde el mismo concepto hasta lo concreto, facilitándole no solo el proceso de aprendizaje sino el de puesta en práctica de lo aprendido.

Si bien el texto dispone de un capítulo completo destinado al repaso e interpretación de C/C++, se asume que el alumno posee el conocimiento básico de dicho lenguaje. El Capítulo inicial, está destinado al repaso del lenguaje C/C++ como herramienta de nivelación general de los alumnos, y de homologación-aprendizaje de la semántica implicada ante la definición de una instrucción determinada.

El Segundo Capítulo se destina al análisis del concepto de Recursividad. La idea del capítulo amén de plantear el concepto en sí, es esquematizar cómo se debe plantear la recursividad en términos de un lenguaje de programación formal y la factibilidad de existencia de soluciones no recursivas.

El Tercer Capítulo plantea como tema central al Análisis de Algoritmos. El objetivo del capítulo además de concretamente conceptualizar al Análisis de Algoritmos, es exponer cómo el alumno puede aproximar empíricamente sus propios análisis y con ello determinar el orden de un algoritmo, de cara a que pueda decidir con base empírica qué algoritmo utilizar en deteminado desarrollo de software.

El Cuarto Capítulo se denomina “Estructuras de Datos Sencillas”, en él se plantean las estructuras como las listas, pilas, colas y árboles binarios con sus correspondientes variables asociadas. Amén de conceptualizar cada estructura en sí, el capítulo orienta al usuario sobre la implicancia de cada operación realizada sobre las estructuras, la implementación concreta de cada una ellas y su asociación con la interpretación en memoria de las mismas.

El Quinto Capítulo introduce al alumno en los Algoritmos de Ordenamiento. Vistos los algoritmos tradicionales como el de Burbuja, Inserción y Selección, incursiona en algoritmos con mayor grado de sofisticación como el Quicksort, Shellsort, entre otros. Cada algoritmo dentro del capítulo, es analizado gráficamente mediante pruebas de escritorio de cara a interpretar su funcionamiento y, posteriormente, implementado en C/C++ para que el alumno pueda analizar la forma en que la lógica del algoritmo es concretada.

El Sexto Capítulo Aborda la Problemática de la Búsqueda. En este apartado se introduce al alumno en temas más avanzados, como las estructuras Hash con sus variantes, los árboles AVL, multicamino, B y B+. La idea es brindar al alumno de un panorama general pero actual, de las herramientas de búsqueda e indexación factibles de utilización en software comerciales.

El Séptimo Capítulo se enfoca en una introducción a la Búsqueda de Cadenas y al Reconocimiento de Patrones. En éste apartado se expondrán conceptual y pragmáticamente los algoritmos de fuerza bruta, Boyer-Moore y Knuth-Morris-Pratt y, finalmente, se realizará una breve introducción al reconocimiento teórico de patrones a través de autómatas finitos deterministas.

El Octavo Capítulo, aborda la introducción de los aspectos asociados a la compresión de datos. En éste, se parte desde los mecanismos más simples de compresión, como la codificación de datos por longitud de series, hasta algoritmos más elaborados como el de Huffman. En todos los casos, los algoritmos se encuentran implementados y disponibles para que el alumno pueda codificarlos y probarlos concretamente.

El Noveno Capítulo, aborda uno de los aspectos más delicados en cualquier software comercial, la criptología. El capítulo persigue exponer al alumno un panorama general de la criptología, en donde se enumeran, explican y codifican una serie de algoritmos de encriptación desde la Cifra del César hasta el Base64 según la RFC 3548.

El Décimo capítulo, introduce al alumno en la Teoría de Grafos exponiendo adicionalmente, el concepto de Matriz de Adyacencia, Cierre transitivo y concluye con una aplicación concreta de los grafos en la determinación de la ruta más corta mediante el algoritmo de Dijkstra.

Mi intención es simplemente que éste texto se transforme en una herramienta didáctica para el alumno que inicia su camino en el campo de los Algoritmos y Estructuras de Datos, permitiéndole obtener una comprensión acabada de los principales tópicos de la temática.

Mario Diván
Marzo 28 de 2005
Santa Rosa (La Pampa -Argentina-)

1 Introducción a C/C++

Introducción

Es indudable que el mundo de la computación ha sufrido cambios sustanciales en las últimas décadas. Hace no más de dos décadas se introducían las PC XT como un modelo prometedor en los entornos de oficina, rápidamente las arquitecturas de Pc fueron progresando hasta llegar hoy al Pentium IV, Athlon, Power PC G4, entre otros tantos fabricantes con sus modelos de vanguardia. Es sencillo observar que, independientemente del fabricante, el poder de procesamiento de datos en los computadores personales, ha crecido y sigue creciendo diariamente.

Los lenguajes de desarrollo han ido mutando, actualizándose y en otros casos desapareciendo, debido al adecuación de los mismos a los cambios de paradigmas. En un principio existían lenguajes no estructurados, con famosas sentencias "goto" que permitían el quiebre de la estructura de la rutina, complicando la depuración y mantenimiento de las aplicaciones.

La necesidad de un paradigma estructurado, llevo a que surjan herramientas que le brinden soporte. Dentro de tales herramientas se encontraban los lenguajes estructurados y con ello aparece el "C". Muchas de las ideas del C provienen del lenguaje BCPL, desarrollado por Martin Richards. La influencia del BCPL continuó indirectamente a través del lenguaje B, escrito por Ben Thompson en 1970 para el primer sistema UNIX de la DEC PDP-7. La dificultad con BCPL y B, es que son lenguajes carentes de tipos. Imaginemos la posibilidad de no poder definir tipos de datos, sino tan solo escribir rutinas y nunca declarar variables propias del tipo entero, caracter, flotante, entre otros. Esta dificultad, fue salvada en el lenguaje C escrito por Brian Kernigham y Dennos Ritchie, e incluso se introdujeron jerarquías de tipos de datos, apuntadores, referencias, estructuras, uniones, entre otros temas sobre los que volveremos en apartados posteriores.

Posteriormente, con el advenimiento del Paradigma Orientado a Objetos (POO), surgieron lenguajes comerciales que le brindaban soporte al mismo. Así es como surge Smalltalk, C++, entre otros. C++ puede verse como un súper conjunto del lenguaje C, que fue ampliado y mejorado para soportar al POO.

Tanto C como C++, permiten un acceso a los recursos del ordenador con relativa sencillez, posibilitando el desarrollo de programas de gran envergadura. Dentro de los programas destacados desarrollados con C/C++, se encuentran las versiones del Sistema Operativo Windows (95, 98, NT, entre otras) y Linux (En sus distintas distribuciones), los Motores de Bases de Datos MySQL y Postgresql, entre tantos otros.

La idea del presente capítulo, es introducir al lector en un lenguaje de programación ampliamente utilizado para el desarrollo de software de base y que además, le sirva como herramienta para la implementación de los distintos algoritmos a tratar en los capítulos subsiguientes.

Primeros Pasos en C/C++

Mi Primer Programa

La mejor forma de comenzar la explicación de un lenguaje de programación es programando, por lo que concentrémonos en el siguiente código:

Archivo: **primerpr.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        clrscr();
7        cout<<"Bienvenidos a Taller de Lenguajes II";
8       }
```

El símbolo "#", identifica el comienzo de una directiva de pre procesador, ¿qué significa esto? El programa que se empleó para escribir el texto del código es un editor de texto, podría ser un Bloc de Notas, el vi de Linux, el Edit del DOS o incluso el Word XP. Cuando le solicitamos al compilador que comience su función de compilación sobre nuestro archivo, lee el mismo de arriba hacia abajo y de izquierda a derecha, por lo que si al leer, encuentra el símbolo "#", sabrá que debe de trabajar con la directiva que se especifique *antes* que el código mismo del archivo. En el caso particular de la directiva "include", causa que se incluya el archivo cuyo nombre está delimitado por los símbolos: menor y mayor, en ese orden. Los archivos incluidos, son librerías del lenguaje que posibilitan el empleo de distintos servicios ofrecidos por las mismas. En el caso de iostream.h, posee distintas funciones para el flujo de entrada/salida, una función que pertenece a esta librería es "cout" en la línea 7. La Librería "conio.h", ofrece servicios vinculados a la entrada/salida por consola (Monitor y Teclado en forma estándar), la función "clrscr()" de la línea 6 es un servicio de esta librería. Finalmente, una vez cargadas las librerías "iostream.h" y "conio.h", se prosigue con la compilación del programa.

Todo programa estructurado, posee un cuerpo principal desde donde se llaman funciones, se definen variables, entre otras cosas, y al finalizar el mismo, culmina el programa. En nuestro ejemplo, y por convención del lenguaje, el cuerpo principal se define como la función "main" de la línea 4. El orden en que se realizan las acciones dentro de "main", será el orden en que las ejecutará el programa. En nuestro ejemplo, la primera acción que realizamos es invocar a la función clrscr(), la cual limpia la pantalla. Note que cada sentencia o llamada a función, lleva un ";" al final de su línea, este es el mecanismo utilizado en "C" para señalar el final de una sentencia e inicio de la próxima. Luego, se ejecuta la función "cout", la cual muestra en pantalla "Bienvenidos a Taller de Lenguaje II" y, como a continuación no existen mas sentencias, culmina el programa. Entonces, todo lo que hace nuestro programa es limpiar la pantalla y mostrarnos un mensaje.

La compilación no es lo mismo que la ejecución del programa, hasta el momento el compilador venía desarrollando su tarea de compilación, la cual tiene por objeto, verificar la validez y sintaxis del código escrito pero no la semántica. Para que el programa se convierta en un ejecutable (una aplicación primerpr.exe), debemos enlazarlo y luego ejecutarlo. La Tarea de enlace (o Linking), consiste en unir en un archivo simple (primerpr.exe), el código utilizado desde nuestro programa perteneciente a las librerías y el nuestro, en un mismo lugar para luego traducirlo al lenguaje de la computadora y generar el archivo ejecutable.

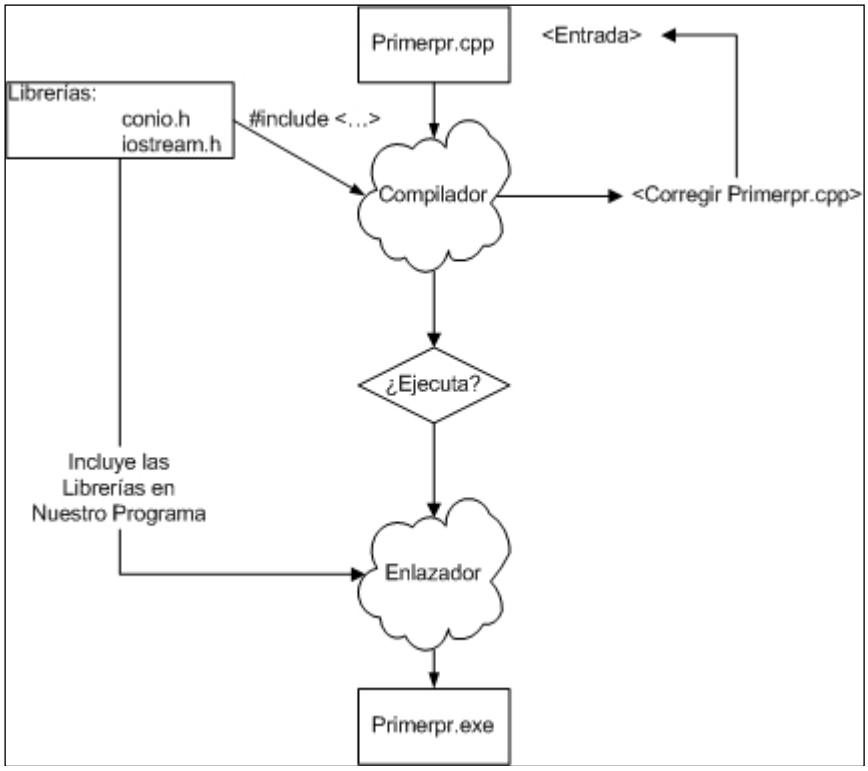


Figura 1: "Esquema de Compilación y Enlace"

Ahora que contamos con nuestro primer programa ejecutable, analicemos un detalle del código; en la línea 4, donde se inicia la función principal del programa, la misma se define como "*void* main(*void*)". El primer void representa el tipo de datos de retorno de la función; como void significa "ausente o sin retorno", no es necesario que al final del cuerpo de "main" exista un valor de retorno. Mientras que el segundo "void" significa, en este caso, que la función no tiene parámetros, "están ausentes"; de todas formas trataremos el tema funciones con posterioridad con más detalle.

Bibliotecas

Las bibliotecas son archivos especiales que poseen servicios a los cuales podemos acceder desde nuestro programa, previa inclusión de las mismas. Dentro de las bibliotecas interesantes, que podremos incluir *al inicio de nuestros archivos de programa*, se encuentran:

Biblioteca	Breve Descripción	Ejemplo de Inclusión
iostream.h	Funciones Asociadas a los Flujos de Entrada/Salida.	#include <iostream.h>
conio.h	Funciones Asociadas a la Entrada y Salida por Consola (Entrada por Defecto: Teclado, Salida por Defecto: Pantalla).	#include <conio.h>
time.h	Funciones de fecha y hora.	#include <time.h>
string.h	Funciones de Cadenas de Caracteres.	#include <string.h>
stdlib.h	Funciones de Utilería Estándar (Ejemplo: función atoi(const char*), convierte una cadena de caracteres a entero).	#include <stdlib.h>
math.h	Funciones Matemáticas.	#include <math.h>
Sodio.h	Funciones Asociadas a Entrada/Salida de Archivos y salidas con formato.	#include <stdio.h>

Tabla 1: “Bibliotecas Básicas“

Existe una segunda forma de inclusión de bibliotecas, y es empleando comillas dobles al inicio y al final de la sentencia. Veamos el programa “primerpr.cpp” reescrito con la nueva forma de inclusión:

Archivo: **primerpr.cpp** (Adaptado)

```
1       #include “iostream.h”
2       #include “conio.h”
3
4       void main(void)
5       {
6        clrscr();
7        cout<<"Bienvenidos a Taller de Lenguajes II";
8       }
```

Las dos formas de inclusiones tienen el mismo objetivo, incluir la librería especificada, pero tienen una sutil diferencia técnica. La Inclusión con los símbolos “<>”, implica que el compilador busca la librería en el directorio especificado por defecto, si allí no la encuentra, el compilador devuelve un error. Mientras que la inclusión mediante comillas dobles “”, buscará la librería en el directorio por defecto especificado al compilador y si no la encuentra, probará en el directorio donde reside el código fuente del archivo que estamos compilando. Por ejemplo, supongamos que el archivo “conio.h” no se encuentra donde debería estar inicialmente para el compilador, y se encuentra en:

c:\usuario\mislibrerias\conio.h (Windows) ó

/home/newuser/mislibrerias/conio.h (Linux/Unix)

si nuestro código fuente (primerpr.cpp) reside en “c:\usuario\mislibrerias\” en Windows, ó para los usuarios Linux/Unix, en “/home/newuser/mislibrerias/”, el compilador buscará en dicho directorio, la encontrará y compilará, para seguir con su

función como si nada hubiese pasado. Si en cambio, la librería “conio.h” residiese en estos directorios realmente, y hubiésemos utilizado la inclusión con “<>”, el compilador nos daría un mensaje de error, mencionándonos que no puede encontrar la biblioteca o librería solicitada.

Archivos de Cabecera

Además de las bibliotecas de las que dispone el lenguaje C o C++, podremos incorporar nuevas librerías y/o bibliotecas en un futuro, para facilitar nuestra tarea de desarrollo.

Los archivos de cabecera, por convención, poseen las extensiones “.h” o “.hpp”. Para poder incluir una librería en nuestro aplicativo, debemos introducir en la directiva “#include” el nombre del archivo “.h” o “.hpp”. Supongamos que hemos desarrollado nuestra propia librería para exponer información en pantalla, a la que denominamos “mipantalla”. La librería posee las definiciones de las funciones, constantes y estructuras necesarias para exponer la información en el monitor, en el archivo “mipantalla.h”. Mientras que el código fuente de las funciones se encuentra en el archivo “mipantalla.cpp”. Para utilizar el contenido de este archivo (“mipantalla.cpp”) en otra aplicación, debemos incluir el archivo cabecera correspondiente (“mipantalla.h”), en la aplicación que lo desee usar; por lo que éste último debe de contener la totalidad de las definiciones de las funciones contenidas en “mipantalla.cpp”. ¿Qué se quiere decir con esto último?, que si defino el prototipo y el cuerpo de la función “mostrar()” en el archivo “mipantalla.cpp”, cuando quiera utilizarlo en otro programa, al compilar el nuevo programa, el compilador me indicará que “La función mostrar no está definida”. En efecto, el mecanismo de inclusión en los programas “C” es a través de archivos de cabecera, y al no figurar el prototipo de “mostrar()” en “mipantalla.h” ocurre tal error. La solución no es incluir también la definición del prototipo en “mipantalla.h” además de la que pre existía en “mipantalla.cpp”, ya que esto último generará un nuevo mensaje de error similar a: “La función mostrar() ya fue definida”. La Resolución de esta incógnita pasa por definir los prototipos de funciones y tipos de datos definidos por el usuario en el archivo de cabecera (mipantalla.h), y el código de las funciones (sus cuerpos) en los archivos “.cpp” (sería “mipantalla.cpp”). De esta última forma, no existirá nada en algún arhivo “.cpp” que no esté definido en su correspondiente archivo “.h”, esto posibilitará una inclusión de la librería en otras aplicaciones sin mayor problema.

Archivo: **newaplic.cpp**

```
1       #include <mipantalla.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        clrscr();
7        mostrar(“Uso mi nueva función”);
8       }
```

Como se mencionó, en el nuevo programa denominado “newaplic.cpp”, se hace uso de la librería “mipantalla” incluyéndola a través de su archivo de cabeceras (ver

línea 1). La invocación de la función “mostrar()”, perteneciente a la librería “mipantalla”, se efectúa en la línea 7. Si al compilar el archivo “newaplic.cpp”, el compilador nota que la función “mostrar()” no está definida en “mipantalla.h”, entonces es aquí donde se devuelve el mensaje “La función mostrar no está definida”. Vale alcarar que, para que la librería “mipantalla” pueda ser utilizada en el programa “newaplic.cpp”, la misma, debe de haber sido compilada y enlazada con anterioridad.

Entrada/Salida por Pantalla

En todo programa es necesario exponer información en pantalla, como así también solicitarla al usuario. Para este objetivo, el C++ ofrece dos flujos predefinidos “cout” y “cin” a través de la librería “iostream.h”.

El flujo predefinido “cout” representa la salida por consola: **c**onsole-**out**, la consola de salida estándar es el monitor. Mientras que el flujo predefinido “cin” representa la entrada por consola estándar: **c**onsole-**in**, la consola estándar de entrada es el teclado. Para direccionar la entrada/salida se utilizan los operadores >> y <<.

Para mostrar un mensaje por pantalla, el flujo predefinido y el operador de salida se funden como sigue: “cout<<”. Las cadenas de caracteres a mostrar por pantalla, deben de estar encerradas entre comillas dobles, por ejemplo: “Bienvenidos a Taller de Lenguajes II” (ver el código del programa primerpr.cpp en el apartado “Bibliotecas”).

Para tomar un dato desde el teclado e introducirlo en alguna variable, el flujo predefinido y el operador de entrada se funden como “cin>>”.

Veamos el siguiente programa para clarificar lo dicho:

Archivo: **edad.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4
5       void main(void)
6       {
7        int edad=0;
8
9        clrscr();
10       cout<<"Ingrese su Edad: ";
11       cin>>edad;
12       cout<<"\n\n\tSu edad es "<<edad;
13      }
```

Este programa, denominado edad.cpp, al ser ejecutado, define una variable del tipo entero en la línea 7 y la inicializa en cero (Veremos en más detalles las variables en apartados posteriores), luego limpia la pantalla en la línea 9 para mostrar el mensaje por pantalla:”Ingrese su Edad: ” en la línea 10. Hasta el momento, lo único nuevo fue la definición e inicialización de la variable; en la línea 11, la sentencia “cin>>edad;” provoca que en la posición inmediatamente después del último caracter del mensaje mostrado por la línea 10, se quede titilando el cursor a la espera de que se ingrese algún valor por teclado y luego presionemos la tecla “Enter”. Al presionar “Enter”, el valor ingresado se asigna a la variable edad y luego, este valor es utilizado en la instrucción de la línea 12 para mostrar la edad cargada.

Aclaración: Tenga en cuenta que el valor ingresado por teclado será asignado a la variable del tipo "int", por lo que dicho valor debería ser un entero. Si intenta cargar un valor no entero como ser "hola" o "7.21", sencillamente dicho valor no será almacenado y el valor mostrado por la instrucción de la línea 11, puede ser lo que el programa interprete. En programas complejos, cada instrucción de carga de datos, tiene asociado una o más rutinas de validación.

Mostrar el valor de cualquier tipo de variable por pantalla, es relativamente sencillo como se muestra en la línea 12.La mecánica para exponer información de variables y mensajes es:

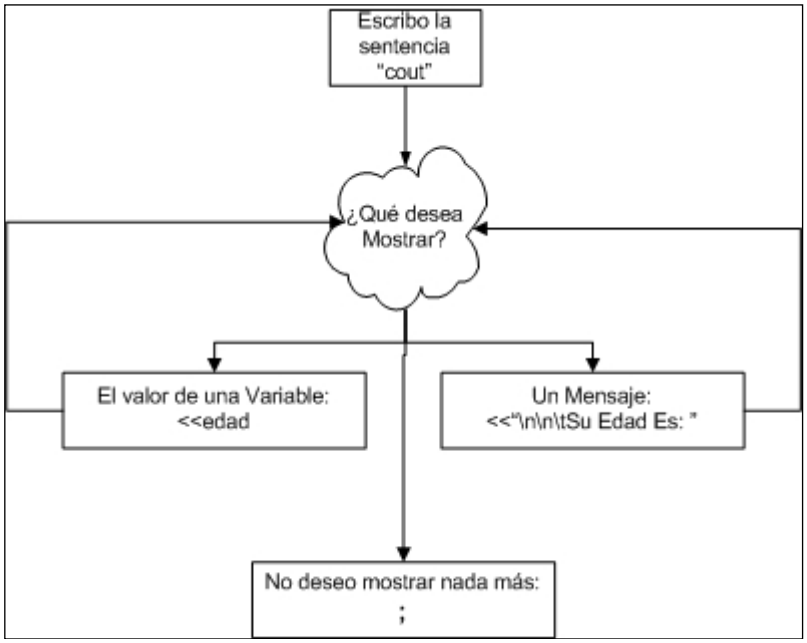


Figura 2: "Esquema de Exposición con cout"

Supongamos que aún no hemos escrito la línea 12 de nuestro programa edad.cpp y nos solicitan, que expongamos un mensaje y el valor de la variable edad por pantalla. Analicemos cómo escribiríamos la sentencia utilizado el Esquema de Exposición de la figura 2:

1. Como la información la expondremos por pantalla, escribimos *cout*. Hagamos de cuenta que disponemos de un buffer imaginario en donde vamos almacenando lo que escribiríamos en nuestro código, en él tendríamos solo "cout".
2. ¿Qué deseamos mostrar?: en primer momento el mensaje "\n\n\tSu Edad es: ", al ser una cadena de caracteres, debemos encerrarla entre comillas dobles. Los modificadores \t y \n los discutiremos con posterioridad. Así agregamos el mensaje inmediatamente después de "cout". Al agregar nuestro mensaje, el buffer imaginario quedaría como "cout<<"\n\n\tSu Edad es: "". Ahora volvemos a iniciar el ciclo de la figura 2.

3. ¿Qué deseamos mostrar?: Ahora necesitamos mostrar el valor cargado en la variable edad por pantalla, así agregamos a nuestro buffer imaginario "<<edad" quedando como sigue: "cout<<"\n\n\tSu Edad es: "<<edad". Ahora volvemos a iniciar el ciclo de la figura 2.
4. ¿Qué deseamos mostrar?: Nada más, entonces agregamos ";" al final de lo escrito en el buffer quedando como sigue: "cout<<"\n\n\tSu Edad es: "<<edad;". Ahora transcribimos el contenido del buffer imaginario a nuestro programa, y obtenemos al fin nuestra famosa línea 12.

En forma análoga para cualquier mensaje o variable que desee mostrar, veamos un nuevo ejemplo. Supongamos que tenemos que mostrar un mensaje que diga la altura y peso de una persona, algo como "Su Altura es 1,78 mts y su peso 90 kg" ¿Cómo lo haríamos si disponemos de una variable *altura* y otra variable *peso*?, como sigue:

1. colocamos *cout* en nuestro buffer imaginario quedando "cout".
2. Ingresamos nuestro operador "<<" y el mensaje sobre la altura al buffer imaginario como sigue: "cout***<<"Su Altura es "***".
3. Ingresamos el operador "<<" y la variable *altura* en nuestro buffer imaginario como sigue: "cout<<"Su Altura es "***<<altura***".
4. Ingresamos el operador "<<" y el mensaje sobre el peso en nuestro buffer imaginario como sigue: "cout<<"Su Altura es "<<altura***<<" y su peso "***".
5. Ingresamos el operador "<<" y la variable *peso* en nuestro buffer imaginario como sigue: "cout<<"Su Altura es "<<altura<<" y su peso "***<<peso***".
6. Como no deseamos mostrar nada más, agregamos el ";" al final del buffer imaginario como sigue: "cout<<"Su Altura es "<<altura<<" y su peso "<<peso**;**".

Así la sentencia queda definitivamente, como sigue:

```
cout<<"Su Altura es "<<altura<<" y su peso "<<peso;
```

¿Cómo hacer para tabular un Texto o escribir en una línea inferior en la pantalla?, es aquí donde aparecen los modificadores de texto. De los más utilizados veremos "\t" para tabular un texto y "\n" para escribir en una línea inferior. La mejor forma de interpretarlos es con ejemplos, así que veamos el cuadro a continuación:

Sentencia	Ejemplo en Pantalla
Texto sin formato:	Esto es una prueba
cout<<"Esto es una prueba";	
cout<<"\tEsto es una prueba";	_____Esto es una prueba
cout<<"\nEsto es una prueba";	
	Esto es una prueba
cout<<"\n\tEsto es una prueba";	
	_____Esto es una prueba
cout<<"\t\tEsto es una prueba";	_____Esto es una prueba
cout<<"\n\nEsto es una prueba";	
	Esto es una prueba

Tabla 2: "Ejemplos de Exposición de Texto con Modificadores"

El uso de los modificadores es relativamente simple, donde se desea el salto de línea o el espacio de tabulación, simplemente se coloca "\n" o "\t" respectivamente. En

caso de que en algún mensaje se necesite mostrar el caracter "\" se debe colocar "\\" dos barras invertidas seguidos, para que el compilador entienda de que se desea en realidad, mostrar el caracter "\" y no un modificador de texto. Veamos un ejemplo, donde deseemos mostrar nuestro directorio home en Linux:

```
cout<<"\\home\\miusuario";
```

En pantalla, aparecerá:

```
\home\miusuario
```

Comentarios

En un programa bien escrito, no pueden faltar los comentarios en el código fuente que faciliten la tarea de mantenimiento, depuración y actualización. C y C++ permiten la inserción de comentarios en sus programas de la siguiente forma:

Archivo: **Edad.cpp** (Adaptado)

```
1       #include "iostream.h" //incluye la librería iostream
2       #include "conio.h"
3       /*Este es un comentario que ocuparía
4       más de una línea*/
5       void main(void)
6       {
7       // int edad=0; Anulo la sentencia, convirtiéndola a comentario para
8       // el compilador, es un mecanismo de depuración útil.
9        clrscr();
10       cout<<"Ingrese su Edad: ";
11       cin>>edad;
12       cout<<"\n\n\tSu edad es "<<edad;
13      }
```

Los comentarios que no ocupen más de una línea pueden efectuarse con "//", como por ejemplo en la línea 1. Aquellos comentarios que ocupen más de una línea pueden efectuarse con "/*" indicando el inicio del comentario, y "*/" indicando la finalización del mismo, como en la línea 3 y 4. También podemos comentar varios renglones con "//", como en la línea 7 y 8.

Variables

Concepto

Una variable, técnicamente, es una cantidad de memoria reservada, dinámica ó estáticamente (La Gestión dinámica de memoria la veremos en apartados posteriores), en una posición precisa de la memoria global del sistema, que es asignada a un identificador con un nombre definido. La cantidad de memoria solicitada, depende del tipo de datos en la gestión estática, o del tipo de estructura en la gestión dinámica.

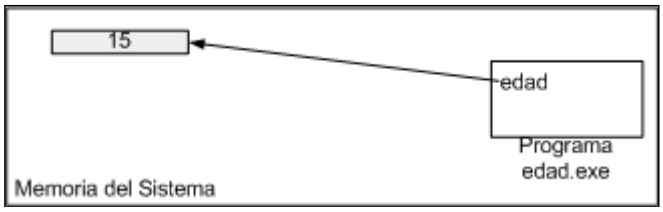


Figura 3: "Esquema de una Variable en Memoria"

Si recordamos nuestro programa "edad.cpp", cuando el mismo sea ejecutado, solicitará memoria para albergar un entero. Dicho espacio de memoria reservado, tendrá una longitud de 2 bytes (correspondiente a un entero), y se accederá a través del nombre de variable "edad", desde el programa "edad.cpp", para inicializar su valor o asignarle uno nuevo. En este ejemplo de la figura 3, se asume que el usuario, cuando se le solicito que ingrese una edad con "cin>>edad", cargo por teclado el valor "15" y presiono enter.

Aclaración: La longitud en bytes, de los distintos tipos de datos, van evolucionando con el avance de las arquitecturas de las computadoras y los lenguajes de programación. En una AT 386 con Borland Turbo C, un entero tenía un tamaño de 2 bytes, y en un Pentium III (o Equivalente en AMD) con Borland C++ Builder el tamaño de un entero era de 4 bytes. Existe una instrucción en C, que nos permite conocer el tamaño en bytes de los distintos tipos de datos, y es sizeof(<tipo de dato>), se utiliza de la siguiente forma:

```
cout<<"Tamaño de un entero "<< sizeof(int);
```

Tipos de Datos

En todo programa es necesario almacenar información en memoria, ya sea con fines temporales del programa o del usuario. Tenemos diversas formas de solicitar memoria, pero dentro de los mecanismos estáticos, podemos solicitar memoria de acuerdo al tipo de dato que deseemos cargar. Dentro de los tipos de datos nativos en C/C++ encontramos: char, int, float, double, enum, void y punteros.

Caracteres

Un texto se compone de caracteres sencillos del tipo “a”, “L”, “?” y de cadenas como “Esto es una cadena de texto”. Normalmente un caracter equivale a un byte y éste último se compone de 8 bits. Los 8 bits, permite que cada caracter pueda asumir 246 valores diferentes (2^8=256) y representarlos sin problemas.

El mecanismo de definición de una variable del tipo caracter es sencillo, se utiliza la palabra reservada “**char**” seguida del nombre de la variable y un punto y coma al final. Por ejemplo, el siguiente código define cinco variables caracteres:

```
1       char letra1;
2       char letra2,letra3,letra4;
3       char letra5='a';
4       letra2=letra3=letra1=letra4='b';
```

Note que se puede declarar las cinco variables línea por línea, o bien, al ser del mismo tipo de dato, se puede utilizar la “,” (coma) como mecanismo de síntesis en el nombramiento de las variables.

Los caracteres válidos para C y C++ a almacenar en una variable del tipo **char** son:

Tipo	Caracteres Permitidos
Letras Minúsculas	abcdefghijklmnopqrstuvwxyz
Letras Mayúsculas	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Dígitos	0123456789
Símbolos	+ - * / = , . _ : ; ? \ ” ´ | ! # % $ & ( ) [ ] { } ^ @
Combinaciones Válidas	++ -- == && || << >> >= <= += -= *= /= ¿: :: /* */ //

Tabla 3: “Caracteres Permitidos”

Recuerde lo mencionado sobre los caracteres “\n” y “\t”. Los mismos son considerados un carácter y para el C/C++ significan un retorno de carro en el caso de “\n” o un “tab” en el caso de “\t”.

El signo “=”, se utiliza como operador de asignación y se emplea para inicializar los tipos de datos. En el tipo de datos caracter, al ser un caracter simple, se utiliza la letra a asignar a la variable encerrada entre *comillas simples*. En el extracto de código anterior, vemos que en la línea 3 se asigna la letra “a” a la variable letra 5, mientras que también es posible la asignación de un mismo valor a varias variables del mismo tipo, como se muestra en la línea 4.

Enteros

Los valores enteros se comprenden de valores negativos y positivos acotados en un rango. Este rango está definido por el tamaño del entero, como mencionamos anteriormente era de 2 bytes, lo que le permite a cada variable de este tipo, representar un valor entre -32768 y 32767. Supongamos que deseemos asignar el valor 40000 a una variable entera, la misma no lo representará, y en su lugar se almacenará el valor 40000-32767. Si deseamos almacenar valores más grandes a los de un entero, deberemos definir la variable como float o double, depende la precisión que necesitemos.

Aclaración: Recordemos que cuanto mayor precisión le exijamos a una variable, mayor espacio en bytes en memoria necesitaremos, por lo que un estudio de tipos de datos necesarios para el programa es recomendado.

Una variable del tipo entero se define colocando la palabra reservada "**int**" seguida del nombre de la variable. Veamos un ejemplo:

Archivo: **enteros.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        int entero1=8;//asigno el valor 8 al entero 1
7        int entero2,entero3;
8        entero2=entero3=9;//asgino el valor 9 a las variables entero2 y entero3
9
10       clrscr();//limpio pantalla
11       cout<<"Entero 1: "<<entero1<<"\nEntero 2: "<<entero2<<"\nEntero 3:
        "<<entero3;
12      }
```

El programa "enteros.cpp", define tres variables enteras, las inicializa asignándoles un valor, limpia pantalla y muestra el valor de cada variable. Note que la definición y asignación de valores es parecida al tipo de datos "**char**", y en general, en todos los tipos de datos es similar.

La Salida del programa "enteros.cpp" se vería como sigue:

```
Entero 1: 8
Entero 2: 9
Entero 3: 9
```

Figura 4: "Salida por pantalla del programa enteros.cpp"

Flotantes

Los valores flotantes se comprenden de valores negativos y positivos acotados en un rango, incluyendo los valores decimales. Este rango está definido por el tamaño del flotante, y éste posee un tamaño de 4 bytes (32 bits), lo que le permite a cada variable de este tipo, representar un valor entre $-3.4E^{-38}$ y $3.4E^{+38}$. Si deseamos almacenar valores más grandes a los de un flotante, deberemos definir la variable como double, depende la precisión que necesitemos.

Aclaración: Un tipo de datos *float* representa 4 bytes de memoria en un AT 386, 8 bytes en un Pentium III, por cada variable que necesitemos. Si necesitamos mayor capacidad que la enunciada, deberemos definir la variable como *double*, lo cual representa 8 bytes de memoria en una AT 386 y 16 bytes en un Pentium III.

Una variable del tipo flotante se define colocando la palabra reservada "**float**" seguida del nombre de la variable. Veamos un ejemplo:

Archivo: **flotante.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        float flot1=8.5;
7        float flot2,flot3;
8        flot2=flot3=10.456;
9
10       clrscr();//limpio pantalla
11       cout<<"flot1: "<<flot1<<"\nflot2: "<<flot2<<"\nflot3: "<<flot3;
12      }
```

El programa "flotante.cpp", define tres variables flotantes, las inicializa asignándoles un valor, limpia pantalla y muestra el valor de cada variable. Note que la definición y asignación de valores es parecida al tipo de datos "**int**". Preste atención a que el separador decimal en el lenguaje C es el punto ".", vea por ejemplo las líneas 6 y 8.

La Salida del programa "flotante.cpp" se vería como sigue:

```
flot1: 8.5
flot2: 10.456
flot3: 10.456
```

Figura 5: "Salida por pantalla del programa flotante.cpp"

Flotantes de Doble Precisión

Los valores flotantes de doble precisión, se comprenden de valores negativos y positivos acotados en un rango, incluyendo los valores decimales. Este rango está definido por el tamaño del flotante de doble precisión, y éste posee un tamaño de 8 bytes (64 bits), lo que le permite a cada variable de este tipo, representar un valor entre -$1.7E^{-308}$ y $1.7E^{+308}$. Si deseamos almacenar valores más grandes a los de un flotante de doble precisión, deberemos definir la variable como double y utilizar algún modificador de tipos como veremos en apartados posteriores.

Aclaración: Al utilizar un modificador de tipos para incrementar la precisión de un tipo de datos double, estamos intercambiando precisón por mayor espacio en memoria. El tipo de datos double representa 8 bytes de memoria en un AT 386 y 16 bytes en un Pentium III, por cada variable que necesitemos.

Una variable del tipo flotante de doble precisión, se define colocando la palabra reservada "**double**" seguida del nombre de la variable. Veamos un ejemplo:

Archivo: **double.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        double dob1=8.5;
7        double dob2,dob3;
8        dob2=dob3=10.456;
9
10       clrscr();//limpio pantalla
11       cout<<"dob1: "<<dob1<<"\ndob2: "<<dob2<<"\ndob3: "<<dob3;
12      }
```

El programa “double.cpp”, define tres variables flotantes de doble precisión, las inicializa asignándoles un valor, limpia pantalla y muestra el valor de cada variable. Note que la definición y asignación de valores es parecida al tipo de datos “**float**”. Preste atención a que el separador decimal en el lenguaje C es el punto “.”, vea por ejemplo las líneas 6 y 8.

La Salida del programa “double.cpp” se vería como sigue:

```
dob1: 8.5
dob2: 10.456
dob3: 10.456
```

Figura 6: “Salida por pantalla del programa double.cpp”

Enumeraciones

Los tipos de datos enumerados permiten la utilización de tipos definidos por el usuario. La idea de este tipo de definición para el programador, es asignarles nombres de una lista a valores enteros para que, dichos nombres actúen como sinónimo en los programas de los valores enteros (o cualquiera fuera su tipo de datos).

Tradicionalmente, si no se especifica el valor de inicio de una enumeración o lista, se asume por defecto que es cero. Para definir una enumeración se emplea la palabra reservada **enum** y se enuncia el contenido de la misma encerrado entre dos llaves. Veamos el código del programa “enum.cpp” como ejemplo:

Archivo: **enum.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        enum boolean{NO, YES};
7        enum meses{enero=1, febrero, marzo,abril,mayo,junio,julio,
8                   agosto,setiembre,octubre,noviembre,diciembre};
9
```

```
10      int varboolean=NO;
11      int varmes=setiembre;
12
13      clrscr();
14      cout<<"varboolean: "<<varboolean<<"\nvarmes: "<<varmes;
15     }
```

El programa define dos enumeraciones: “bolean” en la línea 7, y “meses” en las líneas 7-8. La enumeración *bolean* inicia con el valor por defecto 0 (cero) y se lo asigna a “NO”, asume un incremento unitario y a “YES” le asigna 1 (uno). En la enumeración *meses*, se especifica el valor de inicio para *enero* en “1”, lo que provoca que como la enumeración asume un incremento unitario, febrero asuma el valor de 2, marzo el de 3 y así sucesivamente.

Con las enumeraciones definidas, puedo asignarle un valor “NO” a una variable entera, la que indirectamente asumirá el valor 0 (cero) de “NO”. En forma análoga puedo asignarle el valor “setiembre” a una variable entera, la que indirectamente asumirá el valor 9 (nueve) de setiembre. Este mecanismo, es una forma útil de simplificar el manejo de constantes y etiquetas en un programa. La salida por pantalla del presente programa se vería como:

```
varboolean: 0
varmes: 9
```

Figura 7: “Salida por pantalla del programa enum.cpp”

Void

El tipo de datos “**void**”, se utiliza para significar valores que ocupan 0 bytes y que no tienen valor. Es frecuente encontrarlo en las definiciones de funciones para indicar que la función no tiene retorno. Un ejemplo conocido de esto último es nuestra función main. Veamos el código del ejemplo “enum.cpp”:

Archivo: **enum.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        enum boolean{NO, YES};
7        enum meses{enero=1, febrero, marzo,abril,mayo,junio,julio,
8                    agosto,setiembre,octubre,noviembre,diciembre};
9
10       int varboolean=NO;
11       int varmes=setiembre;
12
13       clrscr();
```

```
14      cout<<"varboolean: "<<varboolean<<"\nvarmes: "<<varmes;
15     }
```

En el programa "enum.cpp" el void a la izquierda de *main,* indica que la función no posee retorno, y el void entre paréntesis, a la derecha de *main*, representa que la función carece de argumentos.

Este tipo de datos, *void*, suele utilizarse para la definición de punteros genéricos. El tema de punteros genéricos lo veremos con posterioridad al tratamiento de los "Punteros".

Introducción al Concepto de Punteros

Este apartado tiene por intención dar una breve definición de lo que se interpreta en C como puntero, pero un tratamiento con mayor grado de profundidad se encarará en el apartado "Utilización de Punteros en C/C++".

Un puntero es una variable con un tipo definido (int, float, void, char, double, etc.) que almacena una dirección de memoria de una variable a la que apunta, la cual debe ser del mismo tipo definido para el puntero. No es la intención entrar en el tema de punteros actualmente, tan solo veamos un ejemplo gráfico de esta definición:

Figura 8: "Concepto de Puntero"

En este ejemplo, el puntero "p" (el cual se definió como *int *p=&edad;*) apunta a la posición de memoria de la variable edad. Por lo que en el interior del puntero "p", no se encuentra el entero de edad (15), sino la posición de memoria de la variable edad (0xFAFB).

Para definirán puntero a un tipo de datos en particular, se coloca:

<tipo de dato> *<nombre_puntero>

En nuestro ejemplo anterior, como nuestra variable p (el puntero), apuntaba a una variable entera (edad), éste debería definirse como entero también; por lo que la definición de nuestro puntero queda como sigue:

int p*=NULL;

Para asignarle a nuestro puntero p, la dirección de memoria de la variable "edad", se utiliza la siguiente sentencia:

int *p=&edad; //al momento de la definición del puntero 'p' ó
p=&edad; //posterior a la definición del puntero 'p'

Reitero, el tema de punteros y la aritmética de los mismos, se tratará en el apartado "Utilización de Punteros en C/C++" con mayor grado de profundidad. El presente inciso tiene por fin tan solo introducir el concepto.

Ciclo de Vida de una Variable

Por ciclo de vida de una variable, se entiende el período comprendido entre el momento mismo de su gestación en la definición, hasta su muerte con la finalización del bloque en el cual fue definida.

Figura 9: "Ciclo de Vida de Una Variable"

Se entiende por bloque al espacio delimitado por llaves en un programa en C/C++, veamos un ejemplo:

Archivo: **enum.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {//Inicio de Bloque
6        enum boolean{NO, YES};
7        enum meses{enero=1, febrero, marzo,abril,mayo,junio,julio,
8                    agosto,setiembre,octubre,noviembre,diciembre};
9
10       int varboolean=NO;
11       int varmes=setiembre;
12
13       clrscr();
14       cout<<"varboolean: "<<varboolean<<"\nvarmes: "<<varmes;
15      }//Finalización del Bloque => Mueren las variables bolean, meses,
        varboolean y varmes.
```

Aunque existan dos variables con el mismo nombre en bloques anidados, el compilador interpreta como válida aquella que se declaró en el bloque de mayor profundidad. Parece chino pero no es tan complicado, veamos como se comportan las variables en un ejemplo con una sentencia condicional básica:

Archivo: **cvida.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {//inicio del bloque 1
```

```
6        int edad=0;//inicializo la variable edad en 98
7
8        clrscr();
9        cout<<"Ingrese una edad: ";
10       cin>>edad;
11
12       if(edad>50)
13        {//inicio bloque 2
14         int edad=78;
15         cout<<"\nEdad: "<<edad;
16        }//fin bloque 2
17       else
18        {//inicio bloque 3
19         int edad=23;
20         cout<<"\nEdad: "<<edad;
21        }//fin bloque 3
22
23      }//fin del bloque 1
```

¿Cómo es esto?, ¿Definimos tres veces la variable *edad*?, ¿Funciona esto?, la respuesta es sí funciona, las tres definiciones son espacios de memoria independientes por estar definidos en bloques independientes. El programa solicita que el usuario cargue una edad, si la misma es superior a los 50 años ¿Mostrará el valor almacenado en la variable edad cargado por el usuario? no, porque tiene preferencia la variable edad (el mismo nombre) del bloque más profundo, en este caso el bloque 2, por lo tanto muestra "Edad: 78" en la pantalla. Análogamente, si el usuario carga una edad inferior a los 50, se mostrará por pantalla "Edad: 23", independientemente de la edad cargada por el usuario. Nuevamente, esto es debido a la preferencia de la variable definida en los bloques de mayor profundidad. Ahora, ¿Dónde muere la variable edad del bloque 2 y del bloque 3?, ¿Al final del programa?, No, la variable muere al finalizar el bloque en el cual fue definida. En el caso de la variable edad del bloque 2, morirá cuando la rutina de ejecución del programa abandone la línea 16. Mientras que la variable edad del bloque 3, morirá cuando la rutina de ejecución del programa abandone la línea 21.

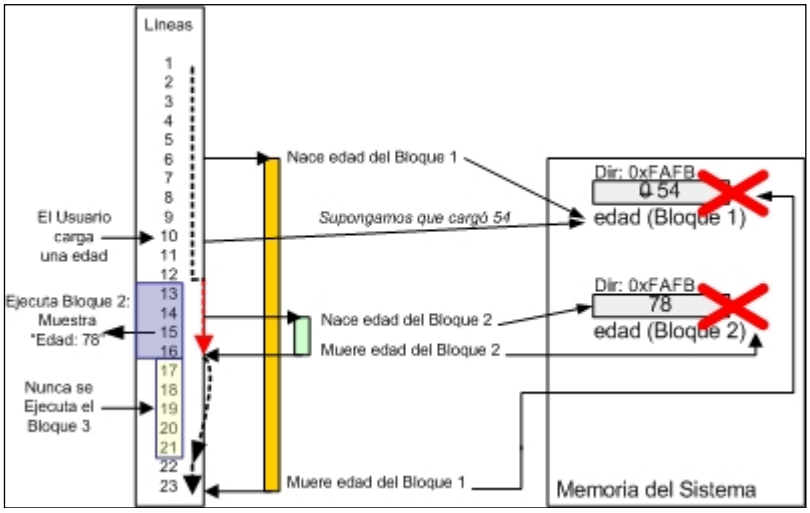


Figura 10: "Diagrama Conceptual del Ciclo de Vida de las variables en el Programa cvida.cpp"

En la figura 10, la línea punteada indica el flujo de control del programa, el cual inicia en la línea 1. En la línea 6 del bloque 1, es donde el usuario define la variable *edad* y se produce el nacimiento de la misma. El programa prosigue su ejecución, el usuario carga la edad solicitada, hasta la línea 12 donde se produce la primera bifurcación, si asumimos que el usuario cargo la edad "54", el flujo de control sigue en la línea 13, en la línea 14 se define la variable edad del bloque 2 e inicializa con el valor "78". En el bloque dos se muestra el contenido de la variable edad del bloque más profundo, 78, cuando llega a la línea 16 muere la variable edad del bloque 2 y el flujo de control pasa a la línea 22. ¿Y la variable edad del bloque 3? nunca se ejecutó porque el flujo de control no tuvo necesidad de pasar por allí; entonces el programa nunca creo ni destruyo la variable edad del bloque 3. El flujo de control, una vez en la línea 22, continúa hasta la finalización del bloque 1 (en la línea 23) y con ello se llega al final del programa, debido a que el bloque 1 era también el bloque principal del programa.

Conversión de Tipos

En un programa se trabaja con más de un tipo de datos, asignar valores entre las variables de un mismo tipo no es una complicación; ¿Pero que ocurriría si las variables entre las que se quiere intercambiar valores, son de distintos tipos? En algunos casos podremos efectuar lo que se denomina "conversión de tipos" y en otros casos no.

En los casos en que podremos efectuar una conversión de tipos, existen dos alternativas: la primera es que perdamos parte de los datos en la conversión y la segunda es que los mantengamos intactos.

Los casos en que **no se pierde información** son los siguientes:

- Convertir el contenido de una variable entera a:
  - Flotante,
  - Doble (Flotante con doble precisión),
  - Cadena de Caracteres.
- Convertir el contenido de una variable flotante a:
  - Doble (Flotante con doble precisión),
  - Cadena de Caracteres.
- Convertir el contenido de una variable flotante a:
  - Cadena de Caracteres.
- Convertir el contenido de una variable cadena de caracteres a:
  - Entero,
  - Flotante,
  - Doble (Flotante con doble precisión).

  (*) En este último caso, la cadena de caracteres debe poseer el formato del tipo al cual desea ser convertido, Ej.: Si en una cadena de caracteres tenemos "23" no va a ver problema de convertirlo a entero, flotante o doble; si por el contrario tenemos "23,456" solo podrá convertirse a flotante o doble.

Los Casos en que se produce la conversión a riesgo de perder información, son los siguientes:

- Convertir el contenido de un Doble a:
  - Flotante,
  - Entero,
- Convertir el contenido de un flotante a:

  - Entero.
- Convertir el contenido de una variable cadena de caracteres a:
  - Entero,
  - Flotante,
  - Doble.

  (*) Cuando el formato de la cadena de caracteres no es coincidente con el tipo de datos al cual se desea convertir.

Los Casos en los que no se pueden convertir los tipos son:

- Cadena de caracteres alfanuméricos a cualquier tipo de dato numérico.
- Contenido de punteros a tipo de datos numéricos.

Veamos cómo se hace lo dicho sobre las conversiones de tipo, en el siguiente ejemplo:

Archivo: **cnvtipos.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3       #include "stdlib.h"
4       #include "string.h"
5
6       void main(void)
7       {
8        //conversiones desde enteros;
9        int a=12;
10       float b=a;//entero a flotante
11       double c=a;//entero a doble
12       char string[20];
13       for(int i=0;i<20;i++) string[i]=' ';//inicializo arreglo
14       itoa(a,string,10);
15       clrscr();
16       cout<<"\nConversiones desde Entero";
17       cout<<"a: "<<a<<"\nb: "<<b<<"\nc: "<<c<<"\nstring: "<<string;
18       //conversiones desde flotantes;
19       b=12.234;
20       a=b; //flotante a entero, se pierde 0.234
21       c=b; //flotante a doble
22       int ndig,dec,sig;
23       ndig=5;//dígitos de precisión
24       strcpy(string,fcvt(b,ndig,&dec,&sig));//Flotante a Cadena, la cadena
               se formatea sin coma decimal.
25       cout<<"\nConversiones desde Flotante";
26       cout<<"a: "<<a<<"\nb: "<<b<<"\nc: "<<c<<"\nstring: "<<string;
27       //conversiones desde double
28       c=12.234;
29       a=c; //Doble a entero, se pierde 0.234
30       b=c; //Doble a flotante
31       ndig=5;//dígitos de precisión
```

```
32      strcpy(string,fcvt(b,ndig,&dec,&sig));//Doble a cadena, la cadena
              se formatea sin coma decimal.
33      cout<<"\nConversiones desde Doble";
34      cout<<"a: "<<a<<"\nb: "<<b<<"\nc: "<<c<<"\nstring: "<<string;
35      //Conversiones desde Cadena de Caracteres v lidas
36      char *stringtoint="23";
37      a=atoi(stringtoint); //de Cadena a Entero
38      char *stringtofloat="23.4568";
39      b=atof(stringtofloat);//de Cadena a Flotante
40      char *stringtodouble="34.325648";
41      c=atof(stringtodouble);//de Cadena a Doble
42      cout<<"a: "<<a<<"\nb: "<<b<<"\nc: "<<c<<"\nstringtoint: "<<stringtoint
43            <<"\nstringtofloat: "<<stringtofloat<<"\nstringtodouble"
44            <<stringtodouble;
45     }
```

Modificadores de Tipo de Dato

Los modificadores de tipo son palabras reservadas del Lenguaje C y C++, que permiten modificar la definición original de un tipo de dato. Esta modificación del tipo de dato, permite agregar precisión a una fracción del tipo de datos a costa de la quita de precisión en otra parte del mismo ó bien, disminuir la precisión de un tipo de datos y con ello la memoria del sistema requerida.

Los Modificadores de Tipos disponibles son los siguientes:

- **unsigned**: Le indica al compilador que utilice la sección negativa de la capacidad de almacenaje de un tipo de dato, para almacenar datos positivos y con ello, aumentar al doble la capacidad de almacenaje de datos positivos del tipo de dato a costa de anular la capacidad de almacenaje de datos negativos.

- **signed**: Este modificador es aplicado por defecto a todos los tipos de datos. Le indica al compilador debe de dedicar la mitad de su capacidad de almacenaje en bytes a números con signo (negativos).

- **short**: Reduce la precisión de un número a la mitad. Esto permite la utilización de la mitad de la memoria que se utilizaría normalmente para un tipo de dato.

- **long**: Permite duplicar la precisión de un número. Esto último incrementa la precisión y con ello el requerimiento de memoria.

Los Modificadores unsigned y signed, como así también short y long, son mutuamente exclusivos entre sí. Con esto se quiere decir que los pares no pueden ser utilizados conjuntamente en una definición. En cambio, unsigned y signed son complementarios con short y long. Esto implica que éstos últimos si pueden ser utilizados en forma conjunta en una declaración de variable.

El código a continuación, muestra algunos ejemplos sobre la aplicación de modificadores de tipo:

	Tipos Básicos de Dato			
	char	int	float	double
signed	**A**	**A**	**NA**	**NA**
unsigned	**A**	**A**	**NA**	**NA**
short	**A**	**A**	**A**	**A**
long	**A**	**A**	**A**	**A**

Tabla 4: "Aplicación de Modificadores de Tipos Permitidas"
(A= Aplicable NA= No Aplicable)

Archivo: **modtipos.cpp**

```
1       void main(void)
2       {
3        //Caracteres
4        unsigned char c1;
5        signed char c2;
6        short char c3;
7        long char c4;
8        unsigned short char c5;
9        unsigned long char c6;
10       signed short char c7;
11       signed long char c8;
12
13       //Enteros
14       unsigned int i1;
15       signed int i2;
16       short int i3;
17       long int i4;
18       unsigned short int i5;
19       unsigned long int i6;
20       signed short int i7;
21       signed long int i8;
22
23       //Flotantes
24       short float f3;
25       long float f4;
26
27       //Dobles
28       short double d1;
29       long double d2;
30      }
```

Prestemos atención a un detalle del código, el mismo posee solo una función *main*, pero no incluye ninguna biblioteca ¿Funcionará?, sí funciona, la inclusión de las bibliotecas es tan solo a los fines de la utilización de alguna función en ella incluida. Como aquí no utilizamos ninguna función que el compilador del C/C++ no interprete en forma nativa, no es necesario la inclusión de biblioteca alguna.

Constantes

Una constante, técnicamente, es un espacio de memoria referenciado a través de un identificador único en su ámbito (el nombre con el que se definió la constante), que no cambia su valor a lo largo de su existencia en el bloque donde fue definido.

El ciclo de vida de una constante tiene las mismas consideraciones que para el de una variable, por lo que lo dicho en el apartado "Ciclo de Vida de una Variable", es válido para este

Cualquier intento de querer asignar un valor a una constante produce un error en tiempo de compilación.

Si se desea declarar una constante global se puede utilizar la directiva de pre procesador "#define", tendría la siguiente forma:

```
#define nombre "José Ignacio Torres"
```

La Anterior línea definiría una constante global al *main* de nuestro programa denominada *nombre,* que contendría como valor constante la frase "José Ignacio Torres". Dicha definición en memoria se interpretaría como sigue:

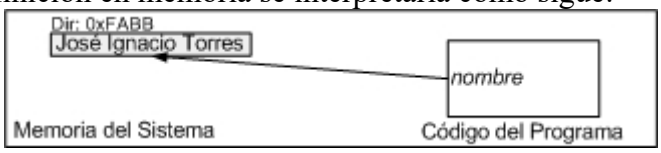


Figura 11: "Conceptualización del Almacenamiento de una Constante en Memoria"

La constante denominada "nombre", solo reservará memoria para almacenar "José Ignacio Torres", ni un byte más ni uno menos.

Es posible, en tiempo de compilación y puntualmente en la definición, indicarle al compilador que a una determinada variable la queremos convertir en constante. Esto no significa que la variable se define, se le asignan valores durante la ejecución del programa y luego se desea convertirla en constante; sino por el contrario, al momento de la definición se puede indicar que una variable de determinado tipo será constante con un valor que es asignado al momento de la definición.

El mecanismo para hacer constante a una variable, es a través de la palabra reservada "*const*" utilizada del siguiente modo:

```
const <tipo de variable> <nombre de variable> = <valor de la constante>
```

Esto implica, que si queremos definir una constante entera con un valor de 1234 sería:

```
const int mientero=1234;
```

De la misma forma se pueden declarar variables del tipo flotante simple, flotante de doble precisión y caracteres.

Una constante, definida mediante ***const***, puede ser utilizada en cualquier tipo de expresión u operación matemática, de concatenación de cadenas de caracteres, de exposición en pantalla, entre otras, siempre y cuando su rol, en dichas operaciones, sea el de exponer su valor almacenado y de ninguna forma el tratar de modificarlo.

Aclaración: Para definir una constante global, es válido utilizar tanto la directiva de pre procesador *#define*, como el modificador *const*. Ocurre que *#define*, solo puede ser utilizado fuera de la declaración de la función principal *main*, mientras que *const*, puede ser utilizado tanto dentro de *main* y/o cualquier otra función, como fuera de las mismas.

Téngase en cuanta que la directiva *#define*, es utilizada para definir macros, en este caso en particular la enfocamos desde el punto de vista de definición de constantes, en apartados posteriores volveremos sobre su capacidad de definir macros funcionales.

Veamos un ejemplo donde reunimos los conceptos de constantes y las operaciones factibles sobre ellas:

Archivo: **const.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       #define TERCERA_EDAD=2
5       #define SEGUNDA_EDAD=1
6       #define PRIMERA_EDAD=0
7
8       const short int R640_480=0;
9       const short int R800_600=1;
10
11      void main(void)
12      {
13       const short char interior[]="Mensaje de Texto";
14       const int nro=10;
15       int seleccion;
16
17       clrscr();//limpio pantalla
18
19       //seleccion=PRIMERA_EDAD/2;Esta l¡nea devuelve error
20
21       seleccion=nro/2;//Esta l¡nea es correcta
22       cout<<seleccion;//muestra un 5
23       cout<<"\nResolución Escogida: "<<R800_600;//muestra un 1
24
25       cout<<"\n"<<interior<<" en el interior del main";
26      }
```

El programa “const.cpp”, muestra en pantalla tres líneas:

- La Primera línea, es un 5 resultado de la expresión matemática en la cual se utilizó la constante (en la línea 21).
- La Segunda línea, utiliza el contenido de la constante global para mostrarla, junto a una cadena de caracteres, en pantalla.

- La Tercera línea, muestra el contenido de la constante "interior", la cual almacenaba una cadena de caracteres, concatenándola con la frase: " *en el interior del main*".

Operadores

Las sentencias de un lenguaje de programación emplean distintos operadores para efectuar operaciones lógicas y/o matemáticas. En C/C++ existen distintas categorías de operadores:

- *Operadores Aritméticos:* Utilizados en expresiones numéricas. Permiten efectuar diferentes operaciones matemáticas básicas.
- *Operadores de Incremento y Decremento:* Utilizados en expresiones numéricas para incrementar en una unidad simple el valor actual de las mismas.
- *Operadores de Relación y Lógicos:* Utilizados en expresiones condicionales y/o comparativas. Permiten efectuar operaciones comparativas y devolver un valor: verdadero ó falso.
- *Operadores para Manejo de Bits:* Utilizados sobre variables del tipo int, char, short int y short char. Permiten efectuar operaciones a nivel de bits sobre los valores almacenados en alguna de las variables de estos tipos.
- *Operadores de Asignación y Expresiones:* Utilizados para asignar valores entre variables y/o conformar estructuras de cálculos o evaluación.

Operadores Aritméticos

Dentro de los operadores aritméticos empleados en las expresiones matemáticas encontramos:

- "*": Operador Binario que permite multiplicar dos expresiones. La forma de utilización es la siguiente: <expresión 1>*<expresión 2>. Expresión 1 y 2 pueden ser constantes u otro tipo de operaciones aritméticas.
- "/": Operador Binario que permite dividir una expresión en otra. La forma de utilización es la siguiente: <expresión 1>/<expresión 2>. Expresiones 1 y 2 pueden ser constantes u otro tipo de operaciones aritméticas. Téngase en cuenta que el compilador efectúa un análisis sintáctico no semántico, por lo que no verificará si la expresión 2 puede en algún momento asumir el valor "0". En caso de que la expresión 2, en tiempo de ejecución, asuma el valor "0" se producirá un Error que puede afectar el correcto funcionamiento del programa.
- "+": Operador Binario que permite sumar dos expresiones. La forma de utilización es la siguiente: <expresión 1>+<expresión 2>. Expresión 1y 2 pueden ser constantes u otro tipo de expresiones aritméticas.
- "-": Operador Binario que permite restar dos expresiones. La forma de utilización es la siguiente: <expresión 1>-<expresión 2>. Expresión 1 y 2 pueden ser constantes u otras operaciones aritméticas.
- "%": Operador Binario que devuelve el módulo de dividir la expresión en la expresión 2. La forma de utilización es la siguiente: <expresión 1>%<expresión 2>. Expresión 1 y 2 pueden ser constantes u otras operaciones aritméticas.

Operadores de Incremento y Decremento

Dentro de los operadores de incremento empleados en las diferentes expresiones, encontramos:

- "++": Operador Unario de Incremento Unitario. Las formas de utilización son las siguientes:
  - Pre Incremento: ++<expresión 1>
  - Post Incremento: <expresión 1>++

  Vea el Apartado "Pre Incremento y Post Incremento", para un mayor detalle.
- "--": Operador Unario de Decremento Unitario. Las formas de utilización son las siguientes:
  - Pre Incremento: --<expresión 1>
  - Post Incremento: <expresión 1>--

  Vea el Apartado "Pre Incremento y Post Incremento", para un mayor detalle.

Operadores de Relación y Lógicos

- ">": Operador Binario de Comparación "Mayor que". La forma de utilización es la siguiente: <expresión 1> **>** <expresión 2>. Si la expresión 1 es mayor que la expresión 2, esta comparación retorna "verdadero", de lo contrario retorna "falso".
- "<": Operador Binario de Comparación "Menor que". La forma de utilización es la siguiente: <expresión 1> **<** <expresión 2>. Si la expresión 1 es menor que la expresión 2, esta comparación retorna "verdadero", de lo contrario retorna "falso".
- ">=": Operador Binario de Comparación "Mayor e igual que". La forma de utilización es la siguiente: <expresión 1> **>=** <expresión 2>. Si la expresión 1 es mayor o igual que la expresión 2, esta comparación retorna "verdadero", de lo contrario retorna "falso".
- "<=": Operador Binario de Comparación "Menor e igual que". La forma de utilización es la siguiente: <expresión 1> **<=** <expresión 2>. Si la expresión 1 es menor o igual que la expresión 2, esta comparación retorna "verdadero", de lo contrario retorna "falso".
- "==": Operador Binario de Comparación "Igual a". La forma de utilización es la siguiente: <expresión 1> == <expresión 2>. Si la expresión 1 es igual a la expresión 2, esta comparación retorna "verdadero", de lo contrario retorna "falso".
- "!=": Operador Binario de Comparación "Distinto de". La forma de utilización es la siguiente: <expresión 1> != <expresión 2>. Si la expresión 1 es distinta de la expresión 2, esta comparación retorna "verdadero", de lo contrario retorna "falso".
- "!": Operador Unario de Negación. La forma de utilización es la siguiente: !<expresión 1>. Si la expresión 1 es verdadera, el operador retorna falso, mientras que si la expresión 1 es falsa el operador Unario retorna verdadero.
- "&&": Operador Binario Lógico "And". La forma de utilización es la siguiente: <expresión 1> && <expresión 2>. Si expresión 1 y 2 son "verdaderas" entonces el resultado es verdadero, de lo contrario es falso.
- "||": Operador Binario Lógico "Or". La forma de utilización es la siguiente: <expresión 1> || <expresión 2>. Si expresión 1 o expresión 2 es verdadera, independientemente de que sea en forma simultánea, el resultado es

verdadero, de lo contrario es falso. Téngase en cuenta que en una operación "o lógica" entre dos expresiones, la única forma de que devuelva falso, es que ambas expresiones sean falsas.

Operadores para Manejo de Bits

- "&": Operador Binario "And" Lógico. La forma de utilización es la siguiente <Expresión 1> & <Expresión 2>. El retorno es el resultado de la operación "and" lógica entre la expresión 1 y 2.
- "|": Operador Binario "Or" Lógico. La forma de utilización es la siguiente <Expresión 1> | <Expresión 2>. El retorno es el resultado de la operación "or" lógica entre la expresión 1 y 2.
- "^": Operador Binario "Or Exclusivo" Lógico. La forma de utilización es la siguiente <Expresión 1> | <Expresión 2>. El retorno es el resultado de la operación "or exclusiva" lógica entre la expresión 1 y 2.
- "<<": Operador Binario "Desplazamiento Izquierdo de Bits". La forma de utilización es la siguiente <Expresión 1> << <Expresión 2>. El retorno es el resultado del desplazamiento de expresión 1 en "expresión 2" veces. Expresión 1 es sobre quien se desplazarán los bits, mientras que expresión 2 indica la cantidad de veces que se desplazarán.
- ">>": Operador Binario "Desplazamiento Derecho de Bits". La forma de utilización es la siguiente <Expresión 1> >> <Expresión 2>. El retorno es el resultado del desplazamiento de expresión 1 en "expresión 2" veces. Expresión 1 es sobre quien se desplazarán los bits, mientras que expresión 2 indica la cantidad de veces que se desplazarán.
- "~": Operador Unario "Complemento a Uno". La forma de utilización es la siguiente ~<expresión 1>. El retorno es el resultado de la operación complemento a uno sobre la expresión 1.

Operadores de Asignación y Expresiones

- "=": Operador Binario de Asignación. La forma de utilización es la siguiente <variable 1>=<expresión 2>. A la variable 1 se le asignará el resultado de la expresión 2. Tengamos en cuanta que el resultado de la expresión 2 debe ser de un tipo compatible con respecto al de la variable 1; para mayor información remítase al apartado "Conversión de Tipos".
- "**op**=": **op** representa a cualquier operador binario que hayamos visto hasta al momento, como ser: +, -, *, /, entre otros. Siempre que se emplea expresiones tales como <variable 1>=<variable 1> **op** <expresión 2>, es posible rescribirla como <variable 1>**op**=<expresión 2> optimizando de esta forma el tiempo de compilación y ejecución. Ejemplo: si tenemos la expresión mivar=mivar+8 es posible rescribirla como mivar+=8, lo que significa a la variable mivar súmele 8 y el resultado, asígneselo nuevamente a mivar.

Operadores	Asociatividad Desde	Hasta
() [] ->	**I**	**D**
! ~ ++ -- + - * *Indirección* & *Dirección* (Tipo) sizeof	**D**	**I**
* / %	**I**	**D**
+ - **(Operadores Aritméticos Binarios)**	**I**	**D**
<< >>	**I**	**D**
< <= > >=	**I**	**D**
== !=	**I**	**D**
& **Operador Binario**	**I**	**D**
^ **Operador Binario**	**I**	**D**
&&	**I**	**D**
||	**I**	**D**
?:	**D**	**I**
= += -= *= /= %= &= ^= |= <<= >>=	**D**	**I**
,	**I**	**D**

Tabla 5: "Precedencia y Asociatividad de Operadores" [1]
(I= Izquierda D= Derecha)

En la anterior tabla figuran operadores que aún no se han tratado con detenimiento como ser el operador de indirección "*", que no se interpreta de la misma forma que el operador producto "*", aunque utilicen la misma simbología el contexto de utilización es diferente. De todas formas la presente tabla, les será de mucha utilidad como referencia a medida que se avance en la temática.

Pre Incremento y Post Incremento

Como se expuso en el apartado "Operadores de Incremento y Decremento", era posible escribir el incremento unitario de una variable entera de la forma ++<variable> ó <variable>++; o bien el decremento unitario de la forma --<variable> ó <variable>--. La pregunta es ¿Es lo mismo escribir el incremento ó decremento en la forma pre incremental ó post incremental?, la respuesta es no.

Archivo: **prepost.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        int c=0;
7        int d=0;
8        int e=0;
9
10       d=c++;
11       c=0;
12       e=++c;
13
14       clrscr();
15       cout<<"\"d\" Post incrementado: "<<d;
```

```
16      cout<<"\n\"e\" Pre incrementado: "<<e;
17     }
```

Este Programa Muestra por pantalla:

```
Post incrementado: 0
Pre incrementado: 1
```

Figura 12: “Salida por pantalla del programa prepost.cpp”

En efecto, en la línea 10, el valor de c es utilizado antes de ser incrementado debido a que utilizamos el operador en el modo postincremento. Esto implica, que en primer momento el valor de “c” es asignado a “d” y luego se incrementa, por lo que “d” toma el valor 0 (cero). En la línea 12 el análisis es diferente, antes de ser utilizado el valor de “c”, éste es incrementado en 1, por lo que el valor asignado a la variable “e”, es el valor incrementado (que en nuestro ejemplo es 1, ya que reiniciamos el valor de “c” en cero en la línea 11); así trabaja el pre incremento.

Ejemplo Integrador de Operadores

En este apartado, trataremos un programa escrito en C/C++ que engloba a todos los operadores tratados anteriormente, en un único ejemplo.

Archivo: **operador.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        //Definición de las Variables a Utilizar
7        int entero1,entero2,entero3,entero4;
8        entero1=1;
9        entero2=2;
10       entero3=3;
11       entero4=4;
12
13       clrscr();
14       //Operadores Aritméticos: + - * / %
15       cout<<"\n\n\t\tOperadores Aritméticos: + - * / %";
16       cout<<"\nProducto: "<<entero1*entero3;
17       cout<<"\nDivisión: "<<(float)entero1/entero4;
18       cout<<"\nSuma: "<<entero1+entero3;
19       cout<<"\nResta: "<<entero1-entero3;
20       cout<<"\nMódulo: "<<entero1%entero3;
21
22       //Operadores de Incremento y Decremento: ++ --
23       cout<<"\n\n\t\tOperadores de Incremento y Decremento: ++ --";
```

```
24      cout<<"\nPre Incremento: "<<++entero1;
25      cout<<"\nPost Incremento: "<<entero2++;
26      cout<<"\nPre Decremento: "<<--entero1;
27      cout<<"\nPost Decremento: "<<entero2--;
28
29      //Operadores de Relación y Lógicos
30      cout<<"\n\n\t\tOperadores de Relación y Lógicos: && || > < >= <= == !=";
31      cout<<"\nRelación >: "<<(entero1 > entero2);
32      cout<<"\nRelación >=: "<<(entero1 >= entero2);
33      cout<<"\nRelación <: "<<(entero1 < entero2);
34      cout<<"\nRelación <=: "<<(entero1 <= entero2);
35      cout<<"\nRelación ==: "<<(entero1 = entero2);
36      cout<<"\nRelación !=: "<<(entero1 != entero2);
37      cout<<"\nRelación &&: "<<(entero1 && entero2);
38      cout<<"\nRelación ||: "<<(entero1 || entero2);
39
40      //Operadores de Manejo de Bits
41      cout<<"\n\n\t\tOperadores de Manejo de Bits: & | ^ ~ << >>";
42      cout<<"\nAnd Lógico a Nivel de Bits: "<<(entero1 & entero2);
43      cout<<"\nOr Lógico a Nivel de Bits: "<<(entero1 | entero2);
44      cout<<"\nOr Exclusivo Lógico a Nivel de Bits: "<<(entero1^entero2);
45      cout<<"\nDesplazamiento Izquierdo de Bits en 1: "<<(entero1<<1);
46      cout<<"\nDesplazamiento Derecho de Bits en 1: "<<(entero1>>1);
47      cout<<"\nComplemento a 1: "<<(~entero1);
48
49      //Operadores de Asignación y Expresiones;
50      entero3+=2;//suma a entero3 la constante 2 y asigna el resultado a
        entero3
51      entero4=10;//asigna el valor de 10 a entero4
52      cout<<"\n\n\t\tOperadores de Asignación y Expresiones";
53      cout<<"\n+=: "<<entero3;
54      cout<<"\n=: "<<entero4;
55      entero4/=2;//divide entero4 en 2 y asigna el resultado a entero4
56      cout<<"\n/=: "<<entero4;
57     }
```

El programa operador.cpp daría la siguiente salida por pantalla:

```
                    Operadores Aritméticos: + - * / %
Producto: 3
División: 0.25
Suma: 4
Resta: -2
Módulo: 1

                    Operadores de Incremento y Decremento: ++ --
Pre Incremento: 2
Post Incremento: 2
Pre Decremento: 1
Post Decremento: 3

                    Operadores de Relación y Lógicos: && || > < >= <= == !=
Relación >: 0
Relación >=: 0
Relación <: 1
Relación <=: 1
Relación ==: 2
Relación !=: 0
Relación &&: 1
Relación ||: 1

                    Operadores de Manejo de Bits: & | ^ ~ << >>
And Lógico a Nivel de Bits: 2
Or Lógico a Nivel de Bits: 2
Or Exclusivo Lógico a Nivel de Bits: 0
Desplazamiento Izquierdo de Bits en 1: 4
Desplazamiento Derecho de Bits en 1: 1
Complemento a 1: -3

                    Operadores de Asignación y Expresiones
+=: 5
=: 10
/=: 5
```

Figura 13: “Salida por pantalla del programa operador..cpp”

Consideraciones sobre el Manejo a Nivel de Bits

Es importante que nos detengamos un momento, antes de seguir avanzando, para analizar la lógica de manejo de bits en un lenguaje de programación.

Un byte representa 8 bits, dependiendo del tipo de variable que definamos y de los modificadores de tipos que utilicemos, dispondremos de una cantidad de bits determinada.

Tipo de Datos	Disponibilidad Bytes	Disponibilidad Bits
char	**1**	**8**
int	**2**	**16**
float	**4**	**32**
double	**8**	**64**
unsigned int (Utilizo el bits de signo para almacenar datos)	**2**	**16**
unsigned char (Utilizo el bits de signo para almacenar datos)	**1**	**8**

Tabla 6: "Tabla de Tamaños de Bits para un AT 486"
(Los Tamaños en Bits varían entre plataformas y lenguajes para con los tipos de datos)

Entonces si una variable entera, llamada entero, almacena el valor 1 en su posición de memoria, su representación binaria será:

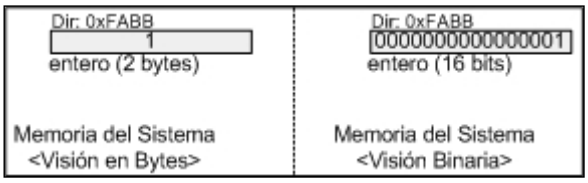


Figura 14: "Conceptualización del Manejo de Bits en Memoria"

A partir de dos variables caracter, más simple aún que las enteras ya que solo poseen 1 byte y por consiguiente 8 bits, analicemos las distintas operaciones binarias sobre ellas.

Operación And Lógico

Una tabla de Verdad es donde se conceptualizan los distintos resultados que pueden obtenerse a partir las distintas variaciones de bits. Veamos la tabla de verdad para la operación lógica "And":

Bits 1	Bits 2	Resultado
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

Tabla 7: "Tabla de Verdad de And"

Téngase en cuenta que C/C++ el 0 (cero) y el valor lógico "Falso" son sinónimos, como así también el valor numérico 1 (uno) y el valor lógico "Verdadero".

Analicemos gráficamente, cómo se llevaría a cabo una operación "And" lógica entre dos variables del tipo "char" en memoria:

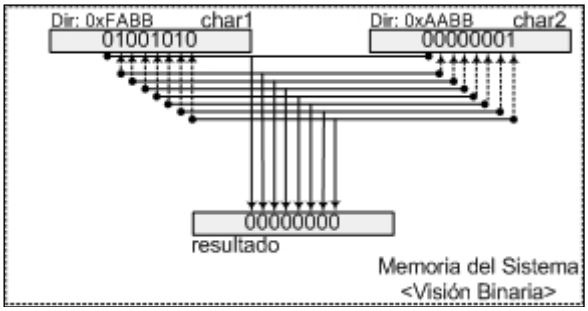


Figura 15: "Conceptualización de la Operación 'And' en Memoria"

Operación Or Lógico

Veamos la tabla de verdad para la operación lógica "Or":

Bits 1	Bits 2	Resultado
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 8: "Tabla de Verdad de Or"

Analicemos gráficamente, cómo se llevaría a cabo una operación "Or" lógica entre dos variables del tipo "char" en memoria:

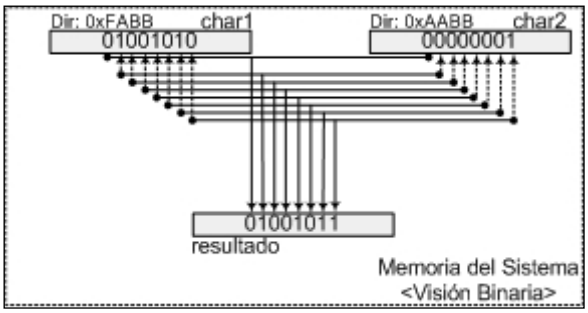


Figura 16: "Conceptualización de la Operación 'Or' en Memoria"

Operación Or Exclusivo Lógico

Veamos la tabla de verdad para la operación lógica "Or":

Bits 1	Bits 2	Resultado
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 9: "Tabla de Verdad de Or Exclusivo"

Analicemos gráficamente, cómo se llevaría a cabo una operación "Or" exclusivo lógico entre dos variables del tipo "char" en memoria:

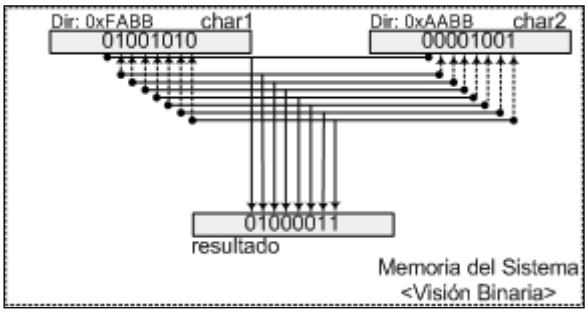


Figura 17: "Conceptualización de la Operación 'Or Exclusivo' en Memoria"

Operación Desplazamiento a la Izquierda

Como mencionamos anteriormente, esta operación desplaza los bits de una expresión y/o variable hacia la izquierda, tantas veces como la expresión a la derecha del operador lo indique.

Veamos la siguiente instrucción en memoria, char1<<1 (desplaza en 1 unidad los caracteres de char1 a la izquierda):

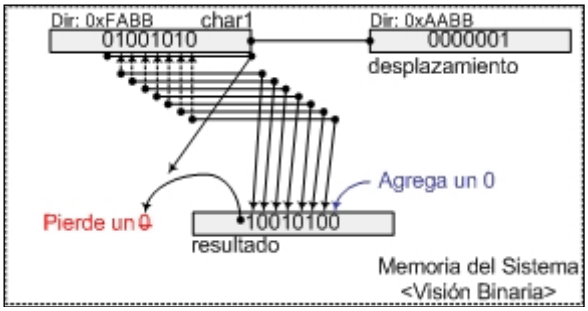


Figura 18: "Conceptualización de la Operación 'Desplazamiento Izquierdo' en 1, en Memoria"

Operación Desplazamiento a la Derecha

Como mencionamos anteriormente, esta operación desplaza los bits de una expresión y/o variable hacia la derecha, tantas veces como la expresión a la derecha del operador lo indique.

Veamos la siguiente instrucción en memoria, char1>>1 (desplaza en 1 unidad los caracteres de char1 a la derecha):

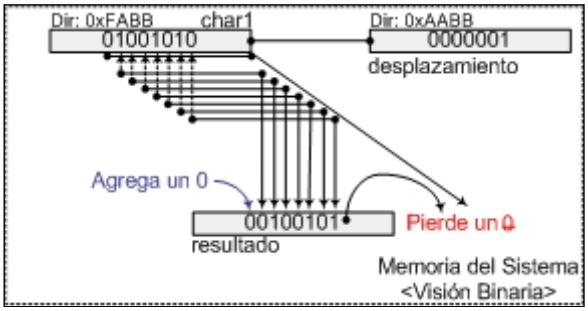


Figura 19: "Conceptualización de la Operación 'Desplazamiento Derecho' en 1, en Memoria"

Operación Complemento a Uno

Hasta el momento hemos analizado operaciones lógicas que involucraban dos variables y/o expresiones. El operador complemento a uno se aplica solo sobre una variable y/o expresión produciendo un efecto de "inversión"sobre los bits de la misma.

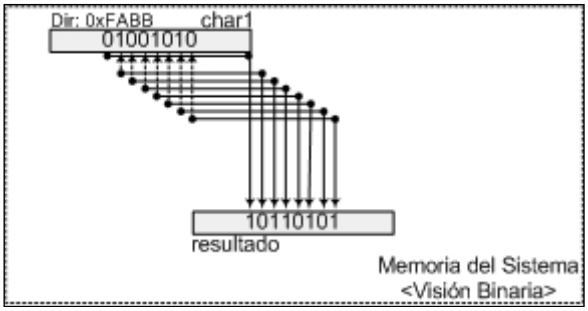


Figura 20: "Conceptualización de la Operación 'Complemento a Uno' en Memoria"

Funciones

Concepto

Una Función es un conjunto de sentencias estructuradas con un objetivo bien definido, la cual es identificada por un nombre único en un programa, a través del cual es invocada. Puede partir de ciertos parámetros de inicio y puede retornar un resultado. El tipo de retorno de la función podrá ser definido por el usuario ó nativo del lenguaje.

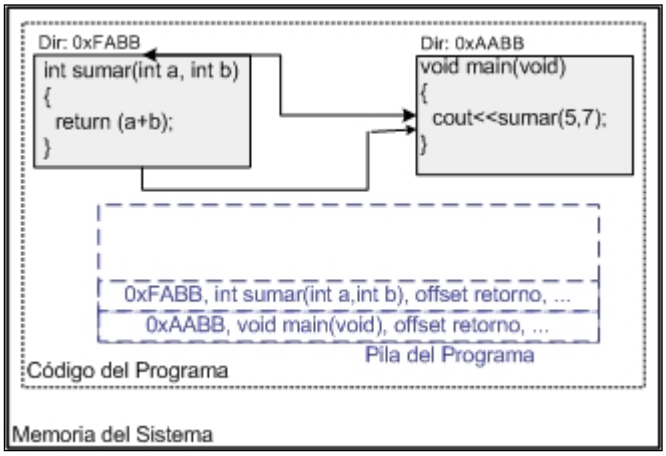


Figura 21: "Conceptualización de una Función en Memoria"

Una función, al ser invocada, deja cierta información almacenada en memoria que permita garantizar su retorno al sitio donde fue invocada. Este sitio de almacenamiento temporal, es conocido como la "Pila del Programa".

Es válido que desde una función, supongamos la función "sumar" del ejemplo, se llamen a otras funciones; estas nuevas llamadas son almacenadas en la parte superior de la pila, lo que implica que al momento de retornar valores, la primera función a retornar será la última función invocada, debido a que es ésta la que permanece en la parte superior de la pila. De esta forma, es totalmente válido el anidamiento de llamadas a funciones, lo que no tiene límite alguno en cuanto a cantidad de invocaciones, sino que su limitación será la cantidad de memoria disponible del ordenador.

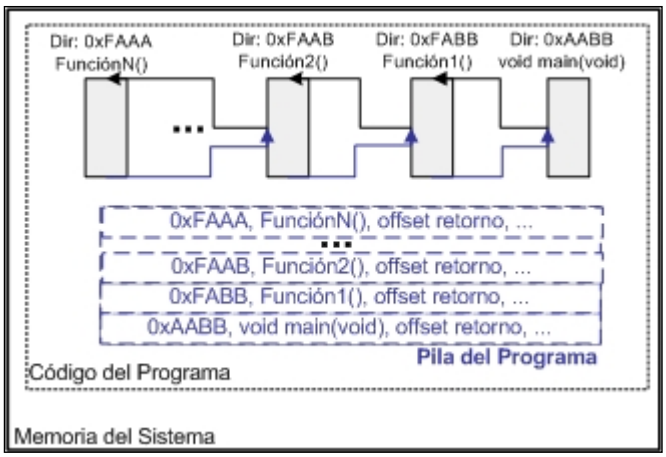


Figura 22: "Conceptualización del Anidamiento de Funciones en Memoria"

Cuando la función invocada retorna su valor, se libera el espacio utilizado en la pila del programa, y se continúa ejecutando la línea inmediatamente posterior de la función invocadora (En nuestro es el *main* la función invocadora).

Implementación de Funciones en C/C++

Una función en C/C++ queda establecida mediante la definición de su prototipo y su cuerpo. El prototipo de una función indica el tipo de retorno, el nombre y los parámetros de la función. El cuerpo de una función incluirá además, las sentencias o rutinas que cumplen con el objetivo de la misma. Por lo general el prototipo de una función es colocado en su correspondiente archivo de cabecera (Archivo ".h"), mientras que el cuerpo es incluido en el archivo fuente (Archivo ".cpp"). A los fines prácticos, y sin perjuicio del código del programa, definiremos los prototipos y cuerpos de las funciones en el archivo fuente, salvo indicación contraria.

Analicemos lo dicho con un ejemplo: Supongamos que desde nuestra función principal *main*, queremos mostrar la suma de dos números enteros cualesquiera.

Archivo: **funcion.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       //Defino el prototipo de la Función
5       int sumar(int a,int b);
6
7       void main(void)
8       {
9        int a,b;
10       a=b=0;
11       int recibe=0;
12
13       clrscr();
14       cout<<"\nIngrese a: ";
15       cin>>a;
16       cout<<"\nIngrese b: ";
17       cin>>b;
18
19       recibe=sumar(a,b);
20
21       cout<<"\n\n\tLa Suma de a+b es: "<<recibe;
22      }
23
24      //Defino el Cuerpo de la función
25      int sumar(int a,int b)
26      {
27       return a+b;
28      }
```

El programa "funcion.cpp", solicita al usuario que cargue dos número enteros por teclado y le devuelve la suma de los mismos. El prototipo de la función queda definido en la línea 5, con un tipo de retorno "int", un identificador único "sumar" y dos parámetros del tipo "int", a y b. La cantidad de parámetros que puede tener una función depende de la capacidad de la pila del programa y, ésta a su vez, de la memoria del ordenador. Los tipos de parámetros que pueden pasarse en una función no tienen porque ser del mismo tipo, pueden ser heterogéneos todos ellos pero con nombres distintos. Por ejemplo:

```
float promedio(int a, float b, double c);
```

Sigamos con el programa "function.cpp". Desde la línea 24 a 28 se define el cuerpo de la función, donde se cita nuevamente el encabezado, pero *sin el ";" al final*, a continuación se abre la llave para comenzar a escribir la rutina de suma, y una vez culminada la misma, se cierran las llaves. En este caso, como la función "sumar" tiene definido un tipo de retorno entero, debe utilizarse como última sentencia ejecutable, la palabra reservada *return* seguida del valor y/o expresión entera que contiene el resultado de la función.

Finalmente, para invocar nuestra función, debemos enunciar su nombre, abrir paréntesis y, separado por comas, enunciar las variables que serán sumadas de acuerdo al orden en que deseemos pasar los valores, luego cerramos el paréntesis. Tenga en cuenta que en una función, al invocarse, no es el mismo resultado funcion(a,b) que funcion(b,a). En el caso de la función sumar(a,b) sí es lo mismo porque la suma aritmética tiene la propiedad matemática de la conmutatividad; pero si deseamos efectuar lo mismo con la resta, recaeremos en un error: resta(5,3) es igual a 2 mientras que resta(3,5) es igual a -2.

En un programa podremos definir tantas funciones como nos hagan faltas para modularizar el comportamiento de nuestro programa.

Tipo de Retorno

Como mencionamos anteriormente, una función puede tener tipos de retorno nativos del lenguaje o definidos por el usuario.

En el caso los tipos de retornos nativos del lenguaje, tenemos:

Tipos de Retorno	Retorno de Punteros	Ejemplo: Definición de Prototipo
Int	*Int**	int resta(int a, int b);
short int	*short int**	short int sumar(int a, short int b);
long int	*long int**	long int aditivo(long a);
unsigned int	*unsigned int**	unsigned int funcionAND(unsigned a,undigned b);
char	*char**	char isCaracter(char* cadena, int posicion);
short char	*short char**	short char isALetter(short char letra);
long char	*long char**	long char isSpecialLetter(long char letra);
unsigned char	*unsigned char**	unsigned char funcionOR(unsigned char c1,undigned char c2);
float	*float**	float promedio(int a, float b, float c, float d);
unsigned float	*unsigned float**	unsigned float funcionOrEx(unsigned float f1, unsigned float f2);
double	*double**	double getSaldoCuenta(int nrocuenta);
unsigned double	*unsigned double**	unsigned double ValorAbsoluto(double a);

Tabla 10: "Tipos de Retorno Nativo"

Recuerde que, en caso de utilizar un tipo de retorno puntero, el valor devuelto será una dirección de memoria, por lo que la variable que reciba dicho valor, deberá ser de su mismo tipo.

En nuestro programa anterior, “funcion.cpp”, la variable que recibía el resultado de la función: “sumar(int a, int b)”, en la línea 19, era “recibe” del tipo “int”. En este caso, el valor de retorno será almacenado en “recibe” sin mayores problemas, ya que sus tipos son idénticos. En caso de que los tipos de retorno de la función y el de la variable de recepción fuesen distintos, existe la posibilidad de perder información. Para más detalles, remítase a las consideraciones del apartado “Conversión de Tipos” del presente capítulo.

Es posible indicar explícitamente, en el prototipo y cuerpo de la función, que el tipo de retorno será una estructura ó clase definida por el programador, pero este tema lo trataremos junto con el de estructuras.

Valores por Defecto en los Parámetros de Funciones

Se pueden establecer, valores por defectos en los parámetros de una función. Para hacerlo, *se agrega en el prototipo solamente*, seguido del parámetro deseado, un signo igual y el valor deseado. Veamos el ejemplo “funcion.cpp” adaptado para el uso de parámetros con valores por defecto.

Archivo: **funciond.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       //Defino el prototipo de la Funci¢n
5       int sumar(int a=7,int b=8);//Defino los valores por defecto
6
7       void main(void)
8       {
9        int a,b;
10       a=b=0;
11       int recibe=0;
12
13       clrscr();
14       cout<<"\nIngrese a: ";
15       cin>>a;
16       cout<<"\nIngrese b: ";
17       cin>>b;
18
19       //Utilizo los Valores por Defecto 7 y 8
20       recibe=sumar();//Note como varia la llamada a la función
21
22       cout<<"\n\n\tLa Suma de a+b es: "<<recibe;
23      }
24
25      //Defino el Cuerpo de la funcion
26      int sumar(int a,int b)
27      {
28       return a+b;
29      }
```

El cuerpo de la función no cambia. Veamos que en el prototipo, hemos agregado dos valores por defecto, 7 para el parámetro "a" y 8 para el parámetro "b". Los parámetros por defecto no implican que serán los únicos a ser utilizados, sino por el contrario, significan que si no se especifica en la llamada a la función, los valores correspondientes para los parámetros, *se utilizarán los valores por defecto para los parámetros ausentes de valor*. En el caso de la llamada de la línea 20 del programa "funciond.cpp", el usuario no introduce ninguno de los valores de los parámetros, por lo que el compilador interpreta que debe usar los parámetros por defecto 7 para "a" y 8 para "b". Veamos una tabla con la variación en la llamada de la función sumar:

Llamada	Valores por Defecto		Valores Utilizados por la Función		Resultado
	a	b	a	b	
sumar()	***7***	***8***	***7***	***8***	***15***
sumar(10)	***7***	***8***	***10***	***8***	***18***
Sumar(5,15)	***7***	***8***	***5***	***15***	***20***

Tabla 11: "Variación de los Valores de los Parámetros en las llamadas a Función"

De esta forma podemos ver lo dicho anteriormente, mientras se encuentre ausente el valor para un determinado parámetro, el compilador utilizará el definido por defecto para el mismo; si no existiese éste último, se producirá un error de compilación.

Lista de Parámetros Variables

Muchas veces debemos definir funciones de las cuales ignoramos la cantidad de parámetros que vamos a utilizar, entonces: ¿Cómo hace en estos casos? C/C++, a través de la librería "stdarg.h" permite dar solución a este inconveniente.

Para indicar que una función tendrá más parámetros de los enunciados, debemos utilizar luego del último parámetro "…". Como mínimo, la función debe contener un parámetro que indique la cantidad de parámetros finales pasados, al momento de la invocación. Veamos lo dicho con un ejemplo:

Archivo: **sparams.cpp**

```
1       #include "iostream.h"
2       #include "stdarg.h"
3
4       int sumar(int totalparametros,...);
5
6       void main(void)
7       {
8        int recibe=sumar(7,5,8,6,0,9,8,10);
9        cout<<"La Suma es: "<<recibe;
10      }
11
12      int sumar(int totalparametros,...)
13      {
14       va_list lista;
15       va_start(lista,totalparametros);
16       int acumulador=0;
17
18       for(int i=0;i<totalparametros;i++)
```

```
19        {
20         acumulador+=va_arg(lista,int);
21        }
22       va_end(lista);
23
24       return acumulador;
25      }
```

El primer "7" en la línea de invocación de la función "sumar" en la línea 8, indica la cantidad de parámetros a ser pasados a continuación. Cuando la función es invocada, se pasan los parámetros a la función sumar, ésta última para poder manejarlos debe definir una variable del tipo *va_list* (Macro definida de C++), en nuestro caso denominada lista en la línea 14. Una vez definida la variable lista, debemos decirle al compilador que en ella albergue los parámetros que hemos adicionado en nuestra llamada a la función; esto último lo hacemos mediante la llamada a la función va_start (de la librería stdarg.h), donde pasamos como primer parámetro a la variable lista y, como segundo parámetro a la variable "totalparametros", la cual contiene la cantidad de parámetros pasada en la invocación.

Para obtener el valor de cada parámetro, utilizamos la función va_arg de la librería "stdarg.h". El primer parámetro de esta función es la lista donde están almacenados los parámetros, mientras que el segundo parámetro indica el tipo de variable que será retornado (similar al tipo de retorno de una función). El tipo de retorno, en nuestro ejemplo "sparams.cpp", es entero y es acumulado en la variable acumulador. Una vez que finaliza el ciclo *for*, sobre el cual volveremos más adelante, se debe liberar el espacio ocupado por la lista, a través de la función va_end pasando como parámetro nuestra lista de parámetros "lista".

Paso de Parámetros

Al pasar una variable como parámetro a una función, existen tres alternativas:

- *Pasar la Variable por Valor:* Al llamar a la función, se le pasa una copia del valor de la variable, permaneciendo inalterable el valor de la primera por parte de la función.
- *Pasar la Variable por Referencia:* Al llamar a la función, se le pasa la dirección de memoria de la variable, siendo alterable el valor de la primera por parte de la función.
- *Pasar la Variable por Dirección:* Al llamar a la función, se le pasa una copia del valor del puntero, el cual contiene la dirección de memoria de la variable, siendo alterable el valor de la primera por parte de la función.

En términos de C/C++, el tipo de paso de parámetros, debe ser definido al momento de escribir el prototipo y cuerpo de la función. Una función definida por valor, es como la vista en el programa "funciond.cpp" (sumar):

```
int sumar(int a=7, int b=8)
```

Al ser invocada una función por valor, el compilador efectúa copias en memoria de los valores pasados como parámetros y la función, a lo largo ed su ejecución, utiliza éstas copias y *no los valores originales de las variables*, veamos lo dicho gráficamente:

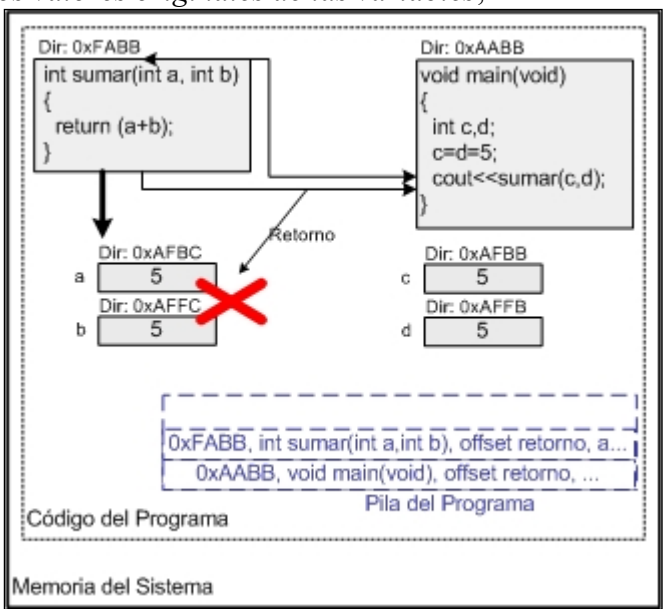


Figura 23: "Conceptualización del Paso de Parámetros por Valor"

Note que en el paso de parámetros por valor, como se mencionó, la función trabaja con copias de los valores pasados como parámetro, que al finalizar la función son destruidas (Al momento de retornar el valor objetivo de la función).

Una función se define por Referencia, como sigue:

int sumar(int &a, int &b);

Tanto "a" como "b" no representan variables comunes, sino referencias a las variables pasadas como parámetros. Las referencias podrían interpretarse como sinónimos del nombre original de la variable, lo que permite un acceso a la posición de memoria real de almacenamiento del contenido. Veamos lo dicho gráficamente:

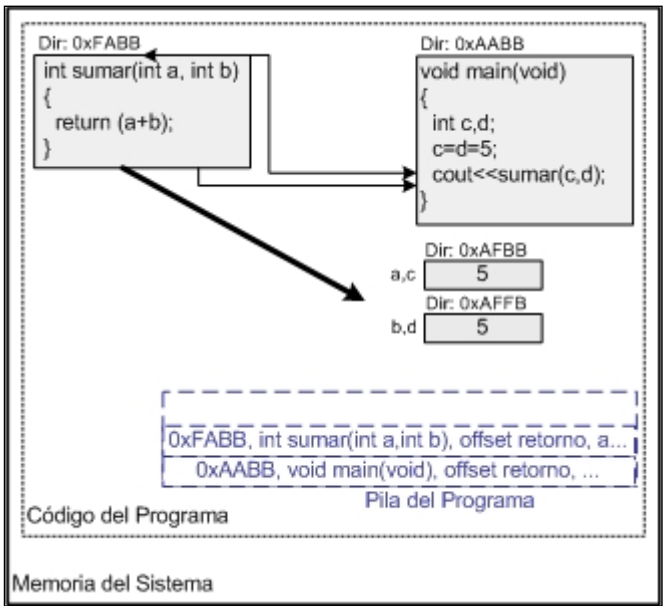


Figura 24: "Conceptualización del Paso de Parámetros por Referencia"

Note que en el paso de parámetros por referencia, como se mencionó, la función trabaja con los valores reales de los parámetros, mientras que cualquier modificación en "a" y "b" dentro de la función, modificará "c" y/o "d" respectivamente. Al finalizar la función son destruidos los sinónimos, no se libera memoria de la variable, ya que aún no culminó el bloque en el que fueron definidas "c" y "d".

Una función se define por Dirección, empleando punteros, como sigue:

*int sumar(int *a, int *b);*

Tanto "a" como "b" no representan variables comunes, sino punteros a las variables pasadas como parámetros. Los punteros son variables que almacenan la dirección de memoria de variables de su mismo tipo, lo que permite un acceso a la posición de memoria real de almacenamiento del contenido. Veamos lo dicho gráficamente:

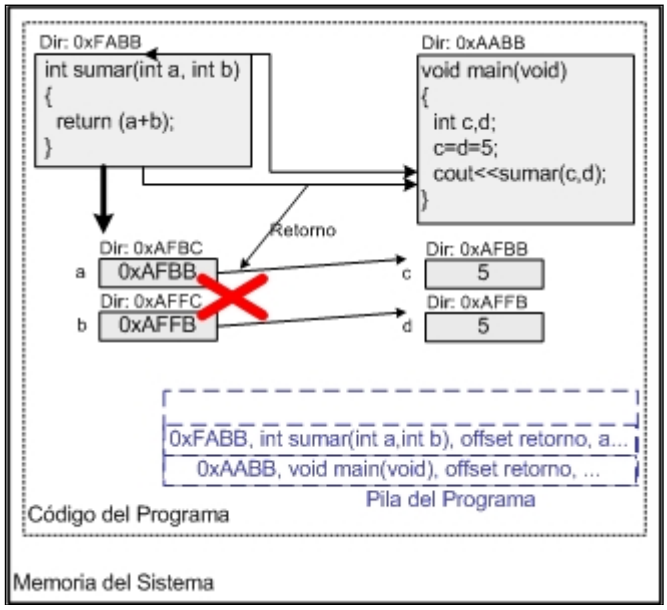


Figura 25: "Conceptualización del Paso de Parámetros por Dirección"

Note que en el paso de parámetros por dirección, como se mencionó, la función trabaja con los valores reales de los parámetros, mientras que cualquier modificación en "a" y "b" dentro de la función, modificará "c" y/o "d" respectivamente. Al finalizar la función, son destruidas las variables del tipo puntero: "a" y "b".

Ejemplo Integrador

El programa "varefdir.cpp", define tres funciones: sumar, resta y producto. La función sumar es la ya vista en programas anteriores, y pasa sus parámetros por valor. La función resta, efectúa la resta aritmética entre dos valores, pasando los parámetros por referencia. Finalmente, la función producto, efectúa la multiplicación aritmética entre dos número, pasando sus parámetros por dirección.

Archivo: **varefdir.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       int sumar(int a,int b);
5       int resta(int &a,int &b);
6       int producto(int *a,int *b);
7
8       void main(void)
9       {
10       int c,d;
11       c=9;
12       d=10;
13
14       clrscr();
15       cout<<"\nSuma: "<<sumar(c,d);//muestra "Suma: 19"
16       int recibe=resta(c,d);
17       cout<<"\nResta: "<<recibe<<" Valor c: "<<c;//muestra "Resta: 5 Valor c:
15"
18       recibe=producto(&c,&d);
19       cout<<"\nProducto: "<<recibe<<" Valor d: "<<d;//muestra "Producto: 15
         Valor d: 1"
20      }
21
22
23      int sumar(int a,int b)
24      {
25       return a+b;
26      }
27
28      int resta(int &a,int &b)
29      {
30       a+=6;//modifico "a" y transitivamente "c"
31
32       return a-b;
33      }
34
35      int producto(int *a,int *b)
36      {
37       *b=1;//modifico "b" y transitivamente "d"
38       return (*a)*(*b);
39      }
```

Note que tanto la función resta como producto, modifican el valor de “c” y “d” en forma transitiva, a partir de los parámetros de la función, mientras que esto no es posible desde la función *sumar*, debido a que posee copias de los parámetros pasados en la invocación y, no acceso directo a ellos.

Cadenas de Caracteres

Hasta ahora se ha visto como almacenar en memoria, datos de nuestros programas y/ cargados por el usuario del tipo numérico, pero ¿Cómo se hace cuando se desea almacenar un nombre cargado por el usuario?, la respuesta a esta pregunta y al manejo en general de caracteres, es el tema que toca abordar en este apartado.

Concepto de Cadenas de Caracteres

Las cadenas de caracteres, también conocidas como arreglos, son un conjunto de chars en memoria vinculados, donde cada uno representa un caracter de una palabra, los cuales son localizados a través de un puntero denominado con el nombre del arreglo.

La definición de un arreglo carácter del tipo estático, se da como sigue:

```
char <nombre_arreglo>[<cantidad_caracteres>];
```

El "<nombre_arreglo>" es un nombre de variable con el cual se apuntará en memoria a la cantidad de caracteres solicitada, especificado esto último por "<cantidad_caracter>". Al final de cada arreglo, el "C/C++" utiliza el carácter especial "\n" para indicar el final del mismo, por lo cual, además de la cantidad necesaria, debe tenerse en cuenta éste carácter adicional. En caso de que un programador omita con o sin intención esta última salvedad, corre el riesgo de una lectura/escritura en posiciones no permitidas en memoria.

Supóngase que deseamos almacenar en memoria la cadena de caracteres "Sandra Paola y María Laura", el "C/C++" la vería como:

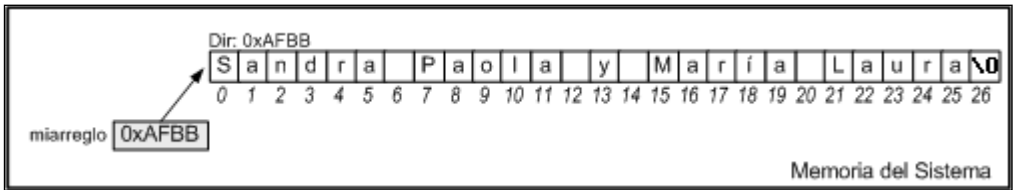


Figura 26: "Conceptualización de una Cadena de Caracteres en Memoria"

La posición inicial de una cadena de caracteres en "C/C++" es siempre 0 (cero). La definición de la cadena de caracteres para la figura 26 sería:

Caso 1:

```
char miarreglo[26];
strcpy(miarreglo,"Sandra Paola y María Laura");
```

ó

Caso 2:

```
char miarreglo[]="Sandra Paola y María Laura";
```

En el Caso 1, el "C/C++" reserva las 27 posiciones de memoria char solicitadas, no las inicializa, y define internamente el puntero "miarreglo" señalando a las posiciones reservadas, luego se utiliza la función "strcpy"de la librería "string.h", para

copiar en la secuencia de caracteres apuntada por miarreglo, la cadena de caracteres especificada la derecha agregando al final, luego de copiarla, el carácter especial "\0".

En el Caso 2, el "C/C++" reserva la cantidad de caracteres requeridos para almacenar la cadena de caracteres especificada a la derecha del signo igual, luego inicializa el puntero miarreglo para que señale a la memoria reservada y finalmente le asigna por orden de aparición y secuencialmente, cada caracter de la cadena a la derecha en cada uno de las posiciones respectivas. Con esto se quiere decir que, la "S" irá a la posición 0, la "a" irá a la posición 1, la "n" irá a la posición 2, la "d" irá a la posición 3, la "r" irá a la posición 4, la "a" irá a la posición 5, el espacio en blanco " " irá a la posición 6, y así sucesivamente hasta el último carácter luego del cual agregará el carácter especial "\0" para señalar el final de cadena. Téngase en cuenta que el carácter en blanco es tenido en cuenta como un carácter más, al igual que cualquier símbolo que se utilizase dentro de la cadena de caracteres.

El Caso solo puede utilizarse en caso de que conozcamos de antemano la cadena de caracteres a ser almacenada, sentencias como:

```
char miarreglo[];
char miarreglo[25]="Hola";
```

son consideradas inválidas por el compilador de C, o se utiliza el mecanismo de definición del Caso 1 ó el del Caso 2, son mutuamente excluyentes.

Notará que, si bien miarreglo es internamente un puntero que señala la posición de memoria reservada para la cadena de caracteres, no se define con "*" sino que en su lugar utiliza "[]". La utilización de "[]" le indica a "C/C++" que se está definiendo un arreglo y que deberá gestionar un puntero interno denominado con el nombre del arreglo, para que señale a la memoria reservada por el arreglo.

Es importante señalar que si solicitamos, como en el caso anterior, espacio de memoria para 27 caracteres (27 bytes), éstos serán alojados en espacios contiguos de memoria, como se muestra en la figura 26, no dispersos o separados de forma alguna. Entonces, los 27 bytes están todos juntos y en términos de variables del tipo "char", el espacio char de "S" es el primero, luego se encuentra el espacio de "a", luego se encuentra el espacio char de "n", y así sucesivamente hasta "\0", que también tiene su espacio char definido y representa la última posición.

Recuperación y Asignación de Caracteres Individuales

Una Cadena de caracteres en C/C++, es interpretada como una secuencia contigua de variables del tipo "char", señaladas por un puntero denominado como el nombre del arreglo. Entonces, no deberían existir mayores inconvenientes en acceder a cada uno de los caracteres en forma individual, en lugar de la frase completa. En efecto, C/C++ permite esto último sin mayores problemas a través del operador "[]". Este operador se utiliza como sigue:

```
cout<<miarreglo[0];// Muestra la 'S', según la figura 26.
cout<<miarreglo[1];// Muestra la 'a', según la figura 26.
cout<<miarreglo[2];// Muestra la 'n', según la figura 26.
```

Esta sentencia le dice a la macro 'cout', "recuperar el contenido de la posición 0 (cero) de la cadena de caracteres 'miarreglo' y mostrarlo por pantalla". Análogamente

con las sentencias posteriores, mostrarían la 'a' para la posición 1 y la 'n' para la posición 2.

Para la figura 26, las posiciones que van desde 0 a 26, se consideran el rango de variación válido de la cadena de caracteres, con esto se quiere decir, que es el rango válido. Nada nos impide acceder a la posición de memoria '55', aunque nada tengas que ver con nuestra cadena, por ejemplo:

```
cout<<miarreglo[55]; //Muestra algo, no se sabe qué.
```

Tenga en cuenta que, como puede acceder a leer las posiciones de memoria del rango válido y no válido (fuera del rango) del arreglo, también las puede escribir, ¿Cómo?, a través del operador de asignación "=", siempre que se asigne un carácter, esto es un valor constante entre comillas simples o una variable del tipo "*char*". Por ejemplo, supongamos que en la posición 0(cero) de miarreglo (según la figura 26), deseamos colocar una 's' en lugar de 'S', sería:

```
miarreglo[0]='s';//coloca s en la posición 0
```

Esta sentencia, coloca en la posición 0 (cero) del arreglo, el carácter 's' pisando lo que hubiese. Sí, leyó bien, *pisando lo que hubiese*. Allí está uno de los principales riesgos de escribir valores fuera del rango válido de un arreglo, se puede pisar una posición de memoria vital para un sistema operativo u otro software, haciéndolo inestable o incluso provocando un reinicio instantáneo del sistema. Note que el carácter es asignado, a la posición deseada del arreglo, mediante el uso de comillas simples, si intenta:

```
miarreglo[0]="s";//Provoca un error de Compilación
```

La sentencia anterior provocaría un error del tipo: "Está intentando asignar una cadena de caracteres a un carácter simple". El compilado de C/C++, interpreta lo encerrado entre comillas dobles como una cadena de carácter, en este caso, interpretaría una cadena de caracteres con longitud uno.

Inicialización de una Cadena de Caracteres

Inicializar una cadena de caracteres, significa asignarles algún valor a cada una de las posiciones de memoria reservadas para ella. Podemos inicializar una cadena de caracteres, colocando un asterisco ('*'), un numeral ('#'), un espacio en blanco (' ') o cualquier otro símbolo que pueda representarse en un caracter, en cada una de las posiciones de memoria del arreglo, en la figura 26 sería, desde la posición 0 a la 25, sin sobrescribir (se mantiene '\0') la posición 27 que indica el final de cadena.

Normalmente, para inicializar un arreglo se utiliza un ciclo repetitivo como el "for", no es la intención introducirnos en ciclos repetitivos ahora, ya que se tratan en apartados posteriores, pero veamos someramente como funciona, a los efectos de entender la inicialización de cadenas de caracteres.

Un ciclo for, tiene la siguiente estructura:

```
for(<sdv>;<cc>;<sefb>)
{
```

```
        …
}
```

sdv (Segmento de Definición de Variables): Es el segmento utilizado para definir las variables internas al ciclo for, esto quiere decir, que una vez culminado el mismo, la variable queda destruida.

cc (Condición de Corte): Es el segmento utilizado para especificar la situación que provocará que el ciclo *for* termine.

Sefb (Sentencia de Ejecución de Final de Bloque): Al ser un ciclo repetitivo, su bloque, delimitado por "{}", se ejecutará tantas veces lo permita la condición de corte. Al final de cada vuelta del ciclo, se ejecuta el segmento al que denominados *sefb¸* el cual puede tener diversas sentencias, separadas por ',' (coma).

Apliquemos lo visto someramente sobre el ciclo 'for' y lo referido a inicialización, en el siguiente ejemplo:

Archivo: **initcad.cpp**

```
1       void main(void)
2       {
3        char miarreglo[20];
4
5        for(int i=0;i<20;i++)
6        {
7         miarreglo[i]=' ';
8        }
9       }
```

El programa initcad.cpp lo único que hace, es definir e inicializar un arreglo. Note que en la definición del arreglo en la línea 3, no se asigna ningún valor, ya que utilizamos el caso 1 de definición.

El ciclo for, en su segmento *sdv*, define una variable interna 'i' y la inicializa en cero, lo que le servirá para ir recorriendo el arreglo. En el segmento *cc*, define que la condición de corte es que "i<20", en efecto, el recorrido del arreglo debe ser dentro de su rango válido que va, en este caso, desde 0 a 19. El último segmento del for, *sefb*, define que al finalizar cada vuelta del ciclo, incremente en uno el valor de la variable "i".

De esta forma, para la primera vuelta del ciclo, i=0, asignará a miarreglo[i] el carácter ' ', lo que es igual a decir que, en el primer ciclo del for, se asigna a miarreglo[0] el carácter ' ', ya que i=0. En la segunda vuelta, con i=1, se inicializa miarreglo[1] con el carácter ' '. En la tercera vuelta, con i=2, se inicializa miarreglo[2] con el carácter ' ', y de esta forma sucesivamente hasta que i=20, donde no cumplirá la condición de corte, 'i<20', lo que provocará la salida del ciclo "for".

Copiar Cadenas de Caracteres

Muchas veces es necesario copiar el contenido de una cadena de caracteres en otra. Supongamos dos cadenas de caracteres, una denominada origen (desde donde se extrae la información a copiar) y otra denominada destino (a donde se copia la información), la función se utiliza como sigue:

Archivo: **cpcad.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3       #include "string.h"
4
5       void main(void)
6       {
7        char origen[]="Cadena a Copiar";
8        char destino[16];
9
10       for(int i=0;i<16;i++)
11       {
12        destino[i]=' ';
13       }
14
15       strcpy(destino,origen);//Copia destino en origen
16       clrscr();
17       cout<<destino;//Muestra "Cadena a Copiar" en la pantalla
18      }
```

La función “strcpy” pertenece a la librería “string.h”. Vale aclarar que la cadena “destino”, debe ser del mismo tamaño o superior a la cadena origen, para no sobrescribir la memoria.

El programa “cpcad.cpp” define dos cadenas de caracteres, una origen y otra destino, pero con la salvedad de que tienen el mismo tamaño, lo que evita problemas de sobre escritura de memoria. Inicializa el arreglo destino entre las líneas 10 y 13, para luego copiar en ella el contenido de la cadena “origen” y mostrarlo en la línea 17.

También podría haberse utilizado la función “strcpy” con una cadena de carácter constante, reemplace la línea 15 por la sentencia:

```
strcpy(destino,”Cadena a Copiar”);
```

y notará que el resultado del programa no es afectado, ya que el C/C++, interpreta el contenido entre las comillas dobles, como cadenas de caracteres.

Comparar dos Cadenas de Caracteres

Para comparar el contenido de dos cadenas de caracteres, se utiliza la función strcmp, perteneciente a la librería “string.h”. La función strcmp tiene la siguiente estructura:

```
strcmp(destino,origen);
```

Esta función retorna un entero, luego de efectuar la comparación. Si el entero que retorna es:

- 0, si las cadenas son iguales,
- <0, si destino es menor que origen,
- >0, si destino es mayor a origen.

El hecho de que una cadena sea mayor o menor a otra, depende del código ASCII comparados. La comparación se hace carácter a carácter entre las dos cadenas, destino y origen, al encontrar una diferencia entre ambas, se fija si el código ASCII del carácter de destino es superior al código ASCII del carácter de origen, si es así, retorna un número mayor a cero sino, retornará un número menor a cero. El número retornado, ante una diferencia, es la resta de los valores ASCII de los caracteres comparados.

Veamos un ejemplo:
Archivo: cadcmp.cpp

```
Archivo: cadcmp.cpp

1       #include "iostream.h"
2       #include "conio.h"
3       #include "string.h"
4
5       void main(void)
6       {
7        char origen[]="hola";
8        char destino[]="hOla";
9
10       clrscr();
11       cout<<strcmp(destino,origen);
12       cout<<strcmp(destino,"hola");
13      }
```

La Primera comparación retorna “-32”, que es la diferencia entre ‘o’ y ‘O’ en valores ASCII, mientras que la segunda comparación retorna 0, ya que las cadenas que se comparan son iguales. Note que es factible, hacer la comparación con cadenas de caracteres constantes, como se ve en la línea 12.

Tamaño de una Cadena de Caracteres

En numerosas situaciones es necesario conocer el tamaño de un arreglo, por ejemplo cuando deseamos copiar valores entre ellos, para lo cual existe una función denominada “strlen”, perteneciente a la librería “string.h”, que nos permite realizar lo.

La forma de utilización es la siguiente:

```
strlen(miarreglo);
```

Esta función retorna un entero, que indica el tamaño del arreglo *sin contar el carácter '\0'*, tenga muy en cuenta esto último. Entonces, el tamaño real necesario para copiar una cadena de caracteres como miarreglo, es strlen(miarreglo)+1.

Veamos un ejemplo:

Archivo: cadlen.cpp

```
1       #include "iostream.h"
2       #include "conio.h"
3       #include "string.h"
4
5       void main(void)
6       {
7        char miarreglo[]="hola";
8
9        clrscr();
10       cout<<strlen(miarreglo);
11      }
```

El programa "cadlen.cpp", define un arreglo y lo inicializa con la cadena "hola" en la línea 7. En la línea 10, se calcula el tamaño del arreglo y se lo retorna para que lo muestre *cout*. Cout muestra 4 no 5, recuerde que esta función no incluye el carácter especial de fin de cadena '\0'.

Concatenar Cadenas de Caracteres

La concatenación de cadenas, consiste en que a partir de dos cadenas de caracteres, se agregue el contenido de una al final del contenido de la otra.

La forma de utilización de la función es:

```
strcat(destino,origen);
```

La sentencia anterior, agregará el contenido de "origen" al final de la cadena de caracteres "destino". Esto implica, que la cadena de destino debe tener el tamaño suficiente para almacenar su información, más la de la cadena origen. Veamos un ejemplo:

Archivo: **cadcat.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3       #include "string.h"
4
5       void main(void)
6       {
7        char destino[12];
8        char origen[]="Amigos";
9
10       clrscr();
```

```
11      strcpy(destino,"Hola ");
12      strcat(destino,origen);
13      cout<<destino;//Muestra "Hola Amigos"
14     }
```

El programa anterior, copia el contenido de ‘origen’ (“Amigos”), al final del contenido de ‘destino’(“Hola “), por lo que al mostrar la cadena de destino, mostrará “Hola Amigos”.

Estructuras de Control

Numerosas veces, en nuestros programas debemos expresar formas de decidir ante distintos eventos, la forma de decidir en C/C++ para tomar el camino de un flujo de sentencias, es a través de sus estructuras de control. Dentro de las mismas, tenemos:

- Estructuras Condicionales de Asignación y Bifurcación.
- Estructuras Repetitivas.
- Estructuras Selectivas

Dentro de las Estructuras Condicionales de Asignación, se encuentra el operador "?:" y, en las Estructuras Condicionales de Bifurcación aparece la sentencia "if".

Dentro de las Estructuras Repetitivas, también conocidas como estructuras controladoras de ciclos, encontramos los ciclos "for", "while" y "do while".

Finalmente, y no por ello menos importante, encontramos las estructuras selectivas, implementadas a través de la sentencia "switch".

Estas estructuras, son las que nos mantendrán ocupados en el desarrollo del presente apartado "Estructuras de Control", se irá tratando cada una de ellas con su correspondiente análisis y aplicación práctica.

Estructuras Condicionales

Como mencionamos, disponemos de las estructuras condicionales de Asignación (?:) y la de Bifurcación (if). El uso de ambas, en absoluto son mutuamente exclusivas, es más, existen reiteradas situaciones en que será necesario un uso conjunto para evitar solicitar más memoria, ya veremos este tema en el siguiente apartado.

Estructura Condicional de Bifurcación

La Estructura Condicional de Bifurcación, **if**, somete a evaluación una expresión encerrada entre paréntesis, luego del if que la define, y si el resultado es "verdadero o distinto de cero" ejecuta la rama verdadera, de lo contrario ejecuta la rama falsa, que es contenida entre llaves, luego de la palabra reservada "else".

```
#include "iostream.h"
#include "conio.h"

void main(void)
{
 int y=5;

 if(y>=5)
  {//Inicio Rama Verdadera
   cout<<"`y` es Mayor o igual a Cinco";
  }//Fin Rama Verdadera
  else
  {//inicio Rama Falsa
   cout<<"`y` es Menor a Cinco";
  }//Fin Rama Falsa
}
```

Expresión de Control

Figura 27: "Comportamiento de una Estructura Condicional de Bifurcación Simple"

En la figura 27, la expression de control es “y>=5”, y al dar verdadero la evaluación de la expresión, debido a que y asume el valor de 5 por definición, ejecuta la rama verdadera mostrando “`y` es Mayor o Igual a Cinco”.

Para Definir una estructura condicional, primero se debe colocar la palabra reservada “**if**”, seguido de ella se abre un paréntesis “(” siguiéndole la expresión de control de la estructura de control (En nuestro ejemplo “y>=5”). Una vez escrita la expresión de control, se cierra el paréntesis “)” y, abriendo la llave “{” se señala el inicio de la rama verdadera del “if”. Seguido de la apretura de la llave, se colocan todas las sentencias que se desean ejecutar por la rama verdadera, una vez escritas todas las sentencias, se cierra la rama verdadera, con la llave “}”.

Si se desea indicar una serie de acciones por la rama falsa, en caso de que la expresión de control de igual a cero o falsa, se debe colocar, luego de la llave “}” que cierra la rama verdadera, la palabra reservada “**else**” seguida de una llave “{”, que indica el inicio de la rama falsa. Nuevamente, se escriben todas las sentencias que desean ejecutarse por dicha rama, y luego de la última sentencia se cierra el bloque falso con la llave “}”.

De esta forma se define una estructura condicional de bifurcación completa, tal y como se mostró en la figura 27. Supóngase ahora que en la rama falsa no desea escribir ninguna instrucción, entonces tiene dos alternativas:

- Escribir luego de la llave de cierre de la rama verdadera: “else{}” u,
- Omitir escritura alguna.

```
#include “iostream.h”
#include “conio.h”

void main(void)
{
 int y=5;

 if(y>=5)
  {//Inicio Rama Verdadera
   cout<<"`y` es Mayor o igual a Cinco";
  }//Fin Rama Verdadera
  else
  {//inicio Rama Falsa
  }//Fin Rama Falsa
}
```

```
#include “iostream.h”
#include “conio.h”

void main(void)
{
 int y=5;

 if(y>=5)
  {//Inicio Rama Verdadera
   cout<<"`y` es Mayor o igual a Cinco";
  }//Fin Rama Verdadera
}
```

Figura 28: “Estructura Condicional Completa vs. Abreviada”

¿Cuándo aplicamos una estructura condicional abreviada?, generalmente en los casos en los que tan solo desea evaluar el valor de una expresión y/ o variable, por ejemplo:

- Deseo saber si la persona es mayor de edad: *if(edad>=21){…}*.
- Deseo saber si la antigüedad del empleado es superior a cinco años: *if(antigüedad>5){…}*.

- Deseo Conocer el sexo de una persona: *if(sexo==`M`){…}*.

Aquellos casos en los que solamente le interesa aplicar una serie de sentencias ante la comprobación de una condición (si es verdadera), sin importar lo que ocurra si es falsa la expresión, también son una buena oportunidad de aplicación.

Es posible anidar las estructuras condicionales de bifuración, ¿Cómo?, supónga el caso en que debe evaluar una serie de condiciones pero solo después de evaluar otras condiciones previas como por ejemplo:

"Si es de sexo masculino y su edad es superior a 20, mostrar un 1 en pantalla, sino, si es femenino y su edad superior a 25, mostrar un 0 en pantalla".

Archivo: **if.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3       #include "string.h"
4
5       void main(void)
6       {
7        char sexo='M';
8        int edad=20;
9
10       clrscr();
11       if(sexo=='M')//Lo denominamos if0
12        {//inicio rama verdadera de if0
13
14         if(edad>=20)//Lo denominamos if1
15          {//inicio rama verdadera de  if1
16            cout<<"1";
17          }//fin rama verdadera de if1
18
19        }//fin rama verdadera de if0
20       else
21        {//inicio rama falsa de if0
22         if(sexo=='F')//Lo denominamos if2
23          {//inicio rama verdadera de if2
24           if(edad>=25)//Lo denominamos if3
25           {//inicio rama verdadera de if3
26            cout<<"0";
27           }//fin rama verdadera de if3
28           else
29           {//inicio rama falsa de if3
30            cout<<"Mujer menor de 25";
31           }//fin rama falsa de if3
32          }//fin rama verdadera de if2
33        }//fin rama falsa de if0
34      }
```

El programa “if.cpp” es una excelente oportunidad de analizar los anidamientos de “if´s” en un programa C/C++. Este programa, define dos variables, sexo (la cual inicializa con `M`) y edad (la cual inicializa con 20), luego limpia pantalla e iniciamos la secuencia de estructuras condicionales. La primera nos pregunta si el valor contenido en la variable sexo es igual al carácter “M” (tenga en cuenta que en C/C++, `M` es distinto de `m`), al ser verdadera esta evaluación, ingresa en la rama verdadera de “if0” donde se pregunta si la variable edad es igual o superior a 20, al ser verdadera esta evaluación, muestra un 1 por pantalla. Al encontrar la llave de cierre de if1, sigue ejecutando lo que siga en la rama verdadera de if0, como no existen más sentencias en ésta última, encuentra la llave de cierre de if0. Al ser if0, la estructura condicional de bifurcación principal del programa, continúa con las sentencias del main que sigan a la estructura if0, salteando la rama falsa del if0, debido a que originalmente la expresión para if0 dio verdadera no falsa. Así, continúa ejecutando lo que exista en el main, como encuentra a continuación la llave de cierre del main, concluye el programa.

Vea en este programa, la facilidad de anidar estructuras condicionales dentro de otras estructuras condicionales, sin mayores problemas en C/C++. Lo que se recomienda fuertemente en el anidamiento de esas estructuras, es que se mantenga un orden indentado desde la estructura condicional principal o primera, hasta la más profunda, sobre todo por la legibilidad del programa.

Estructura Condicional de Asignación

Es posible utilizar una estructura condicional de bifurcación en aquellos casos que, ante el resultado de la evaluación de una expresión, se deba asignar un valor a una variable, por ejemplo:

Archivo: **ifasig.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {
6        int valor=0;
7        char sexo='F';
8
9        clrscr();
10
11       if(sexo=='F')
12        {
13         valor=20;
14        }
15       else
16       {
17         valor=25;
18       }
19      }
```

El programa “ifasig.cpp” en primer momento define una variable valor (la que se inicializa en cero) y otra sexo (la que inicializa con ‘F’), luego limpia pantalla y, si

sexo es ‘F’ le asigna a la variable valor, la constante 20, sino es ‘F’ sexo, se le asigna a valor la constante 25. Este programa cumple Eficazmente con su cometido, pero no es lo suficientemente eficiente como se esperaría. Veamos este programa reescrito utilizando el operador “?:”:

Archivo: ifasig_s.cpp

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {
6        int valor=0;
7        char sexo='F';
8
9        clrscr();
10
11       valor=(sexo=='F')?20:25;
12       cout<<valor;
13      }
```

Veamos como trabaja la línea 11. Al ser la operación principal una asignación se lee de derecha a izquierda. Al encontrar el operador “?”, lee la expresión contenida entre los paréntesis, si es verdadera le devuelve la constante 20 para que lo asigne a “valor”, mientras que si es falsa, devuelve la constante 25 para que lo asigne a “valor”. Podemos destacar que es una solución más elegante y eficiente que la planteada en el programa “ifasig.cpp” pero sirve para asignaciones simples solamente. Esto último quiere decir, que si se desea asignar distintos valores a más de una variable para una expresión evaluada dada, deberemos utilizar la solución ifasig.cpp. Vea ésta última salvedad en el siguiente ejemplo:

Archivo: **ifasig_v.cpp**

```
1       #include "conio.h"
2
3       void main(void)
4       {
5        int valor1,valor2,valor3,valor4;
6        char sexo='F';
7
8        clrscr();
9
10       if(sexo=='M')
11        {
12         valor1=10;
13         valor2=20;
14         valor3=30;
15         valor4=40;
16        }
```

```
17      else
18       {
19       valor1=-10;
20       valor2=-20;
21       valor3=-30;
22       valor4=-40;
23       }
24     }
```

Este tipo de asignaciones no podrían haber sido hechas en una línea como en el programa “ifasig_s.cpp”, sino que o utilizan la solución enunciada en “ifasig_v.cpp”, o bien utiliza una línea al estilo de la línea 11 del programa “ifasig_s.cpp” para cada valor, pero sería menos eficiente debido a que efectúa cuatro comparaciones (y asignaciones) seguidas, contra 1 comparación y cuatro asignaciones (ifasig_v.cpp).

Estructuras Repetitivas

Ciclo For

Algo se ha comentado sobre el ciclo for en apartado posteriores. El ciclo for permite por su propia definición, definir variables de ciclo en su primera sección, la lógica de control de ciclo en su segunda sección y, en su tercera sección, las sentencias a ejecutar por cada final de ciclo. Veamos lo dicho con el ejemplo del apartado “Inicialización de Cadenas de Caracteres”:

Archivo: **initcad.cpp**

```
1      void main(void)
2      {
3       char miarreglo[20];
4
5       for(int i=0;i<20;i++)
6       {
7        miarreglo[i]=' ';
8       }
9      }
```

En el programa “initcad.cpp”, luego de definir el arreglo, se define el ciclo “for” de la siguiente manera:

- Solo por única vez, definirá la variable ‘i’, cuyo bloque de acción estará delimitado por el main, si fuese el for principal (Sin Anidamiento), esta es una de las virtudes del “for”. Tal definición se da en el primer bloque del “for”, el cual finaliza con el primer “;” desde izquierda a derecha.
- La lógica de control del ciclo queda definida en el segundo bloque, empleando la expresión “i<20”. El for, interpreta la expresión de la siguiente forma (i<20)?1:0, en otras palabras, si la expresión evaluada se verifica retorne verdadero y continúe el ciclo, de lo contrario, si la expresión no se verifica, corte el ciclo y salga del mismo. La expresión lógica de control del ciclo es definida por cada programador, lo único que este debe tener en cuenta, es que dicha expresión lógica, en algún momento, produzca un

resultado “falso” que permita salir del ciclo, de lo contrario caería en un bucle infinito que implicaría que el programa quede “colgado”.

- Las sentencias a ejecutar por cada final de ciclo, en el ejemplo, se define como “i++”. Esto último implica que, que por cada finalización de una iteración del ciclo, ejecutará un post incremento unitario en ‘i’.

Si por error un programador, omitiese la expresión lógica de control o bien la escribiese mal, caería en un bucle infinito con consecuencia directa en el programa, se “colgaría”, veamos un ejemplo:

Archivo: **for.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        //For 1
7        for(int i=0;i<20;)//Bucle Infinito, nunca incrementa "i"
8        {
9         cout<<"hola";
10       }
11
12       //For 2
13       for(;;)//funciona: Ciclo Infinito, No existe condici¢n de corte
14       {
15       }
16
17       //For 3
18       i=0;
19       for(int j=0;i<5;j++)
20       {
21        cout<<"chau";
22       }
23      //For 4
24       for(i=0;i<5;i++);//Error Frecuente de Programación: Colocar ";" luego
          del paréntesis de cierre del FOR, nunca muestra Hola TLII
25       {
26        cout<<"Hola TLII";
27       }
28      }
```

En el programa “for.cpp” se pueden apreciar una serie de definiciones, correctas desde el punto de vista sintáctico, pero incorrectas desde el punto de vista funcional del programa.

El For 1, define correctamente una variable “i”, cuyo alcance de bloque es el main (ciclo de vida), correctamente la expresión lógica de control, pero al no

incrementar nunca “i”, la expresión lógica siempre será “verdadera”, lo que provocará un ciclo infinito.

El For 2, es una definición válida desde el punto de vista sintáctico, pero que carece de condición de corte, por lo que C/C++ asume que siempre será verdadera la condición (por defecto), con lo que se ingresa en un ciclo infinito.

El For 3, es correcta desde el punto de vista sintáctico, ya que la variable “j” se define en el primer bloque del presente for, y la variable “i” había sido inicializada en el for 1, con la salvedad de que antes de ingresar en el For 3, se inicializa a cero (línea 18). Este ciclo tiene mal definida la expresión de control, ya que define e incrementa una variable “j”, pero controla con una variable “i” la cual nunca incrementa y, de acuerdo a la inicialización (ver nuevamente línea 18, i=0), la expresión de control del ciclo siempre será verdadera lo que provoca un ciclo infinito.

El for 5, es un ciclo con un error frecuente de programación complicado de localizar. A primera vista, pareciera estar bien definido, pero note que luego del paréntesis de cierre de su definición se colocó un “;”. Este punto y coma, provoca que el ciclo interprete que su cuerpo de acción es {}, en lugar del especificado entre la línea 25 a 27. La consecuencia está a la vista, recorrerá el ciclo sin ejecutar absolutamente ninguna sentencia, ya que interpreta que su cuerpo está vacío ({}).

La ejecución del programa “for.cpp” nunca llega a ejecutar siquiera el For 2, ya que queda atrapado en el bucle infinito del For 1.

Profundicemos un poco más lo dicho sobre la definición de variables en el primer bloque del for. Como se mencionó, definir la variable en este bloque es igual a definirla en el mismo nivel del for, por lo que su ciclo de vida estará determinado por el nivel del for. Veamos un ejemplo:

Archivo: **for2.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {
6        int i=0;
7
8        clrscr();
9
10       for(;i<5;i++)
11        {
12         cout<<i<<"\t";
13        }
14
15       //La Definici¢n del For anterior es equivalente a la
16       //siguiente definici¢n:
17       //for(int i=0;i<5;i++)
18       //{
19       // cout<<i<<"\t";
20       //}
21      }
```

En el programa “for2.cpp” podemos apreciar una de las virtudes de utilizar el ciclo for, dentro del primer bloque del mismo podemos definir variables con alcance al nivel del for (o incluso hasta ejecutar sentencias). Preste atención a la línea 17 (está comentada actualmente), y a las líneas 6 y 10; la línea 17 es equivalente a la línea 6 y 10 juntas, lo que hace más eficiente un programa, de hecho, si llegase a descomentar desde la línea 17 a la 20, el compilador de C/C++ retornaría un error del tipo: “Múltiple Definición para la Variable i”, lo que pone de manifiesto la definición de las variables ‘i’ en un mismo bloque. En el presente programa, la variable “i” dispone del ciclo de vida donde su correspondiente “for” es definido, entonces al definirse el for en la línea 17 (si estuviese descomentada), la variable i, tendría el ciclo de vida correspondiente al bloque que abarca la línea 17, y no es más que el main en este caso.

Veamos ahora, la capacidad de anidar ciclos for:

Archivo: **foranid.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {//Inicio del Bloque 1
6        clrscr();
7
8        for(int i=0;i<5;i++)
9         {//Inicio del Bloque 2
10         for(int j=0;j<5;j++)
11         {//Inicio del Bloque 3
12
13         }//Fin del Bloque 3
14         cout<<i<<"\t";
15        }//Fin del Bloque 2
16
17      }//Fin del Bloque 1
```

Veamos con la simplicidad con la que se pueden anidar bucles for entre sí, o cualquier otro tipo de bucles, tan solo basta con definir el ciclo dentro del cuerpo de otro ciclo, indistintamente de cual fuese, para que se produzca el anidamiento (Vea las líneas 8 y 10). ¿Cómo trabaja el anidamiento?, Primero define las variables para el ciclo de la línea 8, e inicia la ejecución del mismo. Al iniciar la ejecución, encuentra un nuevo ciclo, entonces define las variables y expresiones para éste, e inicia la ejecución de este último. Recién cuando finalice la ejecución del ciclo de la línea 10, continúa con la segunda iteración del bucle de la línea 8. En la segunda iteración, vuelve a ejecutar el ciclo de la línea 10 con sus variables inicializadas, al culminar el mismo, ejecuta la tercera iteración del ciclo de la línea 8, así sucesivamente hasta que se cumpla la condición de control del bucle superior (línea 8). Veamos la salida por pantalla producida por el programa “foranid.cpp”:

[0;0]	[0;1]	[0;2]	[0;3]	[0;4]
[1;0]	[1;1]	[1;2]	[1;3]	[1;4]
[2;0]	[2;1]	[2;2]	[2;3]	[2;4]
[3;0]	[3;1]	[3;2]	[3;3]	[3;4]
[4;0]	[4;1]	[4;2]	[4;3]	[4;4]

Figura 29: "Salida por pantalla del programa foranid.cpp"

La salida por pantalla pone de manifiesto el comportamiento de los ciclos, en la primera vuelta del ciclo de la línea 8, i=0, y al ingresar en el ciclo de la línea 10, imprimirá desde [0;1] hasta [0;4], luego ingresará en la segunda vuelta del ciclo de la línea 8 con i valiendo 1 y así sucesivamente.

Sinteticemos ahora el comportamiento del ciclo for, a través de la siguiente figura conceptual:

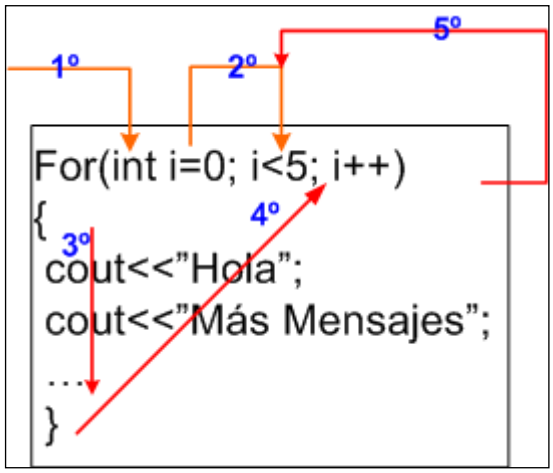


Figura 30: "Conceptualización del Comportamiento del Ciclo For"

En primer momento, se evalúa el primer bloque del for, donde generalmente se inicializan variables. En segundo lugar, se evalúa la expresión de control para ver si cumple, en caso de que no cumpla se corta el ciclo aquí; mientras que si la expresión de control cumple, se ejecuta el bloque del for (3º). Una vez culminada la ejecución del bloque del for, prosigue a ejecutar las sentencias del tercer bloque, en el cual se pueden colocar separado por comas, incrementos de variables, llamadas a funciones, o cualquier otra sentencia que se desee ejecutar al final de la ejecución del cuerpo por cada iteración. Una vez ejecutadas las sentencias post cuerpo (4º), se vuelve a evaluar la expresión de control para ver si es factible comenzar la segunda iteración del mismo, y de esta forma sucesivamente hasta que la expresión de control de "falso".

Ciclo While

El ciclo while, se define mediante el uso de la palabra reservada "while" seguida de un juego de paréntesis "()", dentro de estos paréntesis se coloca la condición de control al igual que en el ciclo for. Este ciclo ejecutará las rutinas de su cuerpo, tantas veces como se verifique la expresión de control, si en algún momento la condición de control es "falsa", se sale automáticamente del ciclo.

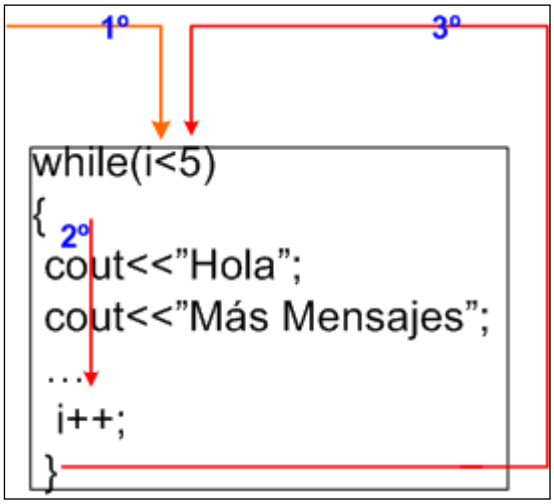


Figura 31: "Conceptualización del Comportamiento del Ciclo While"

En primer momento, se evalúa la expresión de control del ciclo encerrada entre paréntesis, luego de la palabra reservada "while". Si la expresión da falso, sale del bucle sin ejecutar nada, si por el contrario da verdadero, ejecuta el cuerpo del while (2º). Al finalizar la ejecución del cuerpo del while, vuelve a evaluar la expresión de control, si la cumple ejecuta nuevamente el cuerpo, de lo contrario sale del bucle, y así sucesivamente hasta que en algún momento la expresión de control de "falso".

Vea una de las diferencias vitales con el for, el while solo posee expresión de control, por lo que las variables vinculadas a ésta última, deben modificarse de alguna forma en el ciclo para que en algún momento la expresión de "falsa", de lo contrario se caería en un ciclo infinito, vea la salvedad de la sentencia "i++" que precede al cierre de llaves. Ésta última sentencia, asumiendo que 'i' se inicializó en cero, es la que provocará que la expresión de control devuelva "falso" en algún momento, y corte el ciclo. Veamos un ejemplo de ciclos while:

Archivo: **whileani.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {//Inicio del Bloque 1
6        clrscr();
7        int i=0;
8
9        while(i<5)
10        {//Inicio del Bloque 2
11         int j=0;
12
13         while(j<5)
14         {//Inicio del Bloque 3
15          cout<<"["<<i<<";"<<j<<"] \t";
16          j++;
17         }//Fin del Bloque 3
18
19         i++;
20         cout<<"\n";
21        }//Fin del Bloque 2
22      }//Fin del Bloque 1
```

Note que la forma de anidamiento de los ciclos while´s es similar a la de los ciclos for. El programa "whileanid.cpp", muestra en pantalla exactamente el mismo resultado que "foranid.cpp" (Figura 29), solo que implementado mediante ciclos while.

Inicialmente se evalúa la expresión del while de la línea 9, al dar un resultado verdadero, se inicia la ejecución de su bloque (Bloque 2), en él existe otro while cuya expresión de control da verdadero también (debido a que j se inicializa con cero en el bloque 2), entonces ejecuta su cuerpo (el bloque 3). Una vez que culmina la ejecución del ciclo de la línea 13, continúa ejecutando el anterior while (el de la línea 9), incrementa la variable i y vuelve a evaluar la expresión de control de la línea 9, si da verdadero reingresa en el cuerpo (bloque 2) y lo ejecuta nuevamente, de lo contrario corta y sale del ciclo.

Ciclo Do-While

Hasta ahora, todos los ciclos que hemos vistos se denominan 0-N, debido a que si los mismos no satisfacen la expresión de control en la primera oportunidad, nunca se ejecuta el bloque del ciclo. El Ciclo Do-while, se lo conoce como ciclo 1-N, debido a que al menos una vez ejecuta el cuerpo sin evaluar la expresión de control, al finalizar la ejecución del cuerpo recién evalúa la expresión, si da verdadero re ejecuta el cuerpo, de lo contrario sale del ciclo. Veamos la Conceptualización del comportamiento del mismo:

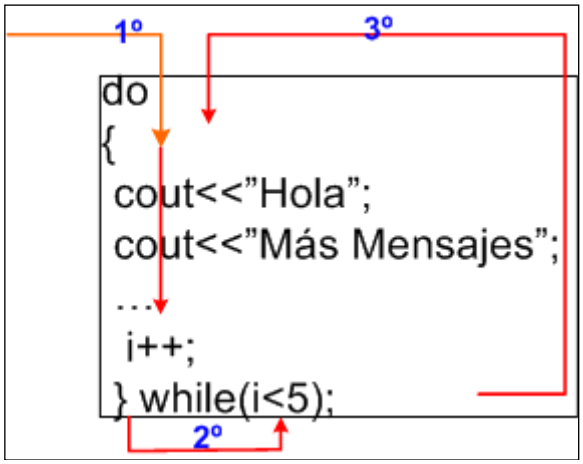


Figura 32: "Conceptualización del Comportamiento del Ciclo Do While"

En primer momento, se ejecuta el cuerpo del ciclo (1º), una vez culminada la ejecución, se evalúa la expresión de control (2º), si es verdadera, vuelve a ejecutar el cuerpo (3º), de lo contrario sale del ciclo.

La definición de un ciclo do-while consiste de la palabra reservada *do*, seguido por la llave de apertura ("{") que indica el inicio del bloque de sentencias, a continuación se colocan las sentencias que se desean ejecutar en el ciclo. Una vez escritas todas las sentencias, se cierra el cuerpo con la llave de cierre ("}"), y a continuación se coloca la palabra reservada *while*, seguida por un juego de paréntesis ("()"). Entre los paréntesis se coloca la expresión de control del ciclo, y una vez escrita la misma, luego del paréntesis de cierre posterior a la expresión de control, se coloca un punto y coma ";". Es el único ciclo, que por definición, lleva el punto y coma al finalizar su declaración.

A continuación re escribiremos el programa "whileani.cpp" con ciclos do-while, para que dé el mismo resultado:

Archivo: **dowhilea.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {//Inicio del Bloque 1
6        clrscr();
7        int i=0;
8
9        while(i<5)
10        {//Inicio del Bloque 2
11         int j=0;
12
13         while(j<5)
14         {//Inicio del Bloque 3
15          cout<<"["<<i<<";"<<j<<"] \t";
16          j++;
17         }//Fin del Bloque 3
```

```
18
19        i++;
20        cout<<"\n";
21       }//Fin del Bloque 2
22     }//Fin del Bloque 1
```

Estructuras Selectivas

Cuando en programa, debemos preguntar a una determinada variable, *entera o carácter*, si posee determinados valores, podemos emplear múltiples "if" como una alternativa eficaz, pero en absoluto eficiente. Existe una estructura capaz de responder la mismas necesidad con mayor eficiencia, y es la estructura switch.

Switch

La estructura switch, se define colocando la palabra reservada "switch" y un juego de paréntesis, dentro de los cuales se coloca la variable, carácter o entera, que será comparada contra cada uno de los valores expresados a continuación en su cuerpo. Luego del paréntesis de cierre, posterior a la variable a evaluar, se coloca un juego de llaves "{}", dentro de los mismos, se expresará qué valores se comparará con el asumido por la variable, y en caso de coincidencia, qué acciones se tomarán. Al final de cada acción por opción, se coloca la sentencia reservada "break", indicando que la anterior fue la última acción de la opción escogida. Veamos lo dicho con un ejemplo:

Archivo: **switch.cpp**

```
Archivo: switch.cpp

1       #include "iostream.h"
2       #include "conio.h"
3
4       void main(void)
5       {//Inicio del Bloque 1
6        clrscr();
7        int i=0;
8
9        switch(i)
10       {
11        case 0:
12              cout<<"Ingreso 0";
13              break;//si no coloco este break, sigue ejecutando el
14              //siguiente cout
15        case 1:
16              cout<<"Ingreso 1";
17              break;
18        default://palabra reservada para ejecutar acciones cuando
19              //el valor comparado no coincide con ninguno de los
20              //listados anteriormente
21              cout<<"Ingreso "<<i;
22              break;
23       }
```

```
24      }//Fin del Bloque 1
```

Como puede verse, la variable que se comparará contra cada valor enunciado en el cuerpo del switch es 'i' (línea 9). El formato para los valores a comparar está dado por: **case** <valor entero/caracter>**:** serie de acciones **break;**. Existe una palabra reservada, para indicar una serie de acciones en caso de que el valor comparado no coincida con ninguno de los enunciados, y es default, su formato es: **default:** serie de acciones **break;**. Recuerde que el *break;* es obligatorio al final de cada serie de acciones, si lo omite, se seguirá ejecutando acciones hasta encontrar un *break;* o el final del cuerpo con "}".

Veamos como trabaja el sistema de comparación del switch, suponiendo que nuestra variable 'i' es igual a 1:

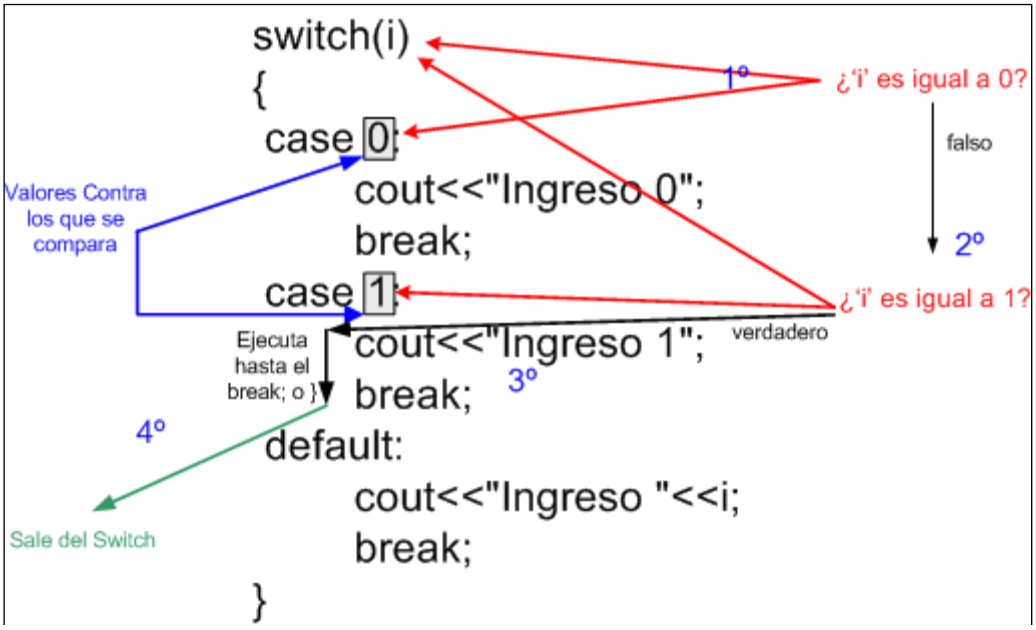


Figura 33: "Conceptualización del Comportamiento del Switch"

En primer lugar, compara el valor asumido por 'i' con el primer valor enunciado '0' (1º), como difieren, continúa la evaluación con el siguiente valor y pregunta 'i es igual a 1' (2º), al ser afirmativa la respuesta, inicia la ejecución de la serie de acciones (3º) hasta encontrar un break o "}". Al encontrar un break, sale del switch (4º). Tenga en cuenta que si los valores enunciados en el switch, no coinciden con el valor de i, se ejecutan las acciones de *default* por defecto.

Tipos de Datos Definidos por el Usuario

Hasta el momento se ha venido trabajando con tipos de datos nativos de C/C++, pero es lógico, que en un programa, existan estructuras de datos cuya estructura sea particular a ella. Podríamos querer definir un tipo de datos denominado fecha para almacenar día, mes y año en forma conjunta, pero ¿Cómo se hace? La respuesta a esta inquietud la enfocaremos desde el punto de vista de las estructuras definidas por el usuario, tema que compete al presente apartado.

Concepto de Tipo de Dato Definido por el Usuario

Un desarrollador, desde el punto de vista del lenguaje un usuario, puede definir sus propias estructuras de datos y emplearlas, posterior a su definición, en cualquier parte de sus programas como si fuese un tipo de datos más.

El Tipo de Dato definido por el usuario se puede interpretar como un conjunto que almacena otros datos, gráficamente podría verse como:

Figura 34: "Conceptualización de la estructura definida por el usuario: 'Fecha'"

Luego de la definición de la estructura, cada vez que se le dice al compilador que se reserve memoria para una variable del *tipo fecha* (según la estructura de ejemplo de la Figura 34), éste reservará la cantidad suficiente para almacenar todos los contenidos de esta estructura o conjunto (como prefiera interpretarlo), por lo que en nuestro ejemplo deberá reservar espacio para tres short int. Ahora, ¿Cómo definimos en C/C++ dicha estructura?, esto se hace con la palabra reservada *struct* seguido de un juego de llaves "{}", dentro de los cuales se colocará la forma en que desea armar la estructura (En nuestro ejemplo son tres short int).

Entonces al definir una variable, denominada var del *tipo fecha* (según la figura 34), en la memoria ocurre lo siguiente:

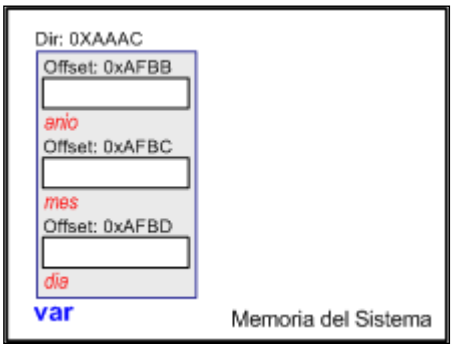


Figura 35: "Conceptualización en Memoria de la estructura definida por el usuario: 'Fecha', ante la definición de una variable 'var' de éste tipo"

La variable 'var', como toda variable, posee su propia dirección de memoria para acceder a su contenido dada por '0XAAAC', según el ejemplo de la figura 35. En su interior aparece un nuevo concepto: 'offset', esto significa el desplazamiento a partir de la dirección de la varable 'var' para el campo deseado. Entonces, si queremos acceder al contenido de mes de la variable 'var', será necesario tanto la dirección de var, como el offset de mes. Retornaremos sobre este tema en detalle, en el apartado "Lectura y Asignación de Valores a un Tipo de Datos Definido por el Usuario".

Lo importante de este apartado, es que pueda comprenderse el concepto mismo de la estructura como mecanismo de definición de tipos de datos definidos por el usuario, y el efecto en memoria, de emplearlo con posterioridad en los programas.

Definición de un Tipo de Datos definido por el usuario

Como se mencionó, el tipo de datos definido por el usuario se implementa a través de la palabra reservada *struct* de C/C++, seguida de un juego de llaves dentro de los cuales, se especificaba la estructura deseada para el tipo de datos.

Veamos un ejemplo de definición del tipo de datos fecha, y la declaración de una variable de este tipo:

Archivo: **struct1.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
6               short int mes;
7               short int dia;
8               };
9
10      void main(void)
11      {
12       fecha var;
13      }
```

El programa "struct1.cpp" no muestra nada por pantalla, sino que define el tipo definido por el usuario fecha (desde la línea 4 a 8), y declara una variable de éste tipo en el cuerpo principal de nuestro programa (línea 12). Note que al finalizar la definición del tipo de datos 'fecha', posterior a la llave de cierre '}', se coloca obligadamente un punto y coma ';'.

Por definición de C, lo que exista entre el paréntesis de cierre de la definición '}' (línea 8) y el ';' (línea 8), se consideran definiciones de variables cuyo ámbito de acción será el mismo que el de la definición del tipo de dato, en este caso sería global. Si queremos definir más de una variable global, posterior a la llave de cierre '}' (línea 8), deberemos separar los nombres de las variables por 'comas'. Ahora, si omitimos el ';' de la línea 8, todas las líneas posteriores hasta encontrar un ';' se interpretarán como intento de definir variables de este tipo de datos, lo que nos provocará una serie de errores de compilación del tipo: (1) ¿usted quiere definir una variable llamada *void* del tipo de datos fecha? (2) void es un nombre de variable no permitido, es una palabra reservada (3) ¿usted quiere definir una variable llamada *main* del tipo de datos fecha? (4) y continúa la colección de errores.

Veamos un ejemplo de definición de variables globales, en la misma línea 8:

Archivo: **struct1.cpp (Modificado)**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
```

```
6               short int mes;
7               short int dia;
8               }var1, var2, var3;
9
10      void main(void)
11      {
12       fecha var;
13      }
```

Vea que simplemente, se agrego una lista separada por comas de las variables globales, que se deseaban definir. Obviamente, estas últimas variables poseen un tiempo de vida superior a la variable var de la línea 12, ya que ésta última es local al main, mientras que las primeras son locales al programa completo. Veamos como se vería en memoria, la situación hasta la línea 12 inclusive:

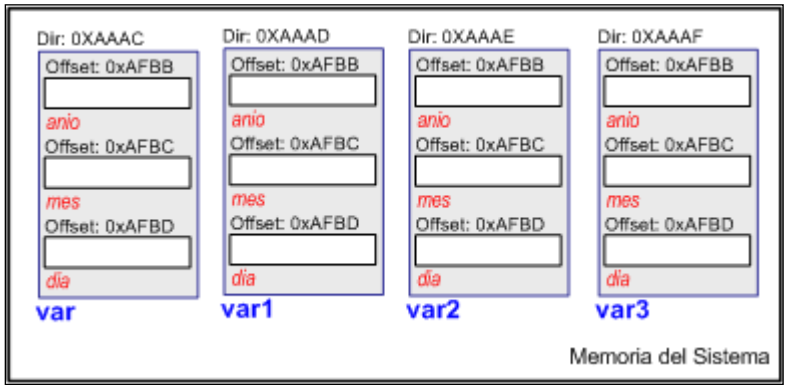


Figura 36: "Conceptualización en Memoria de la estructura definida por el usuario del programa 'struct1.cpp (Modificado)' "

¿Pero todos los campos datos tienen el mismo offset? Es una pregunta lógica, y puede ocurrir, la respuesta es sí. ¿Cómo es eso?, el compilador para acceder a anio de la variable var emplea la dirección de var más el offset de anio, lo que nos devuelve una dirección 'd1'; mientras que para acceder a anio de var1, emplea la dirección de var1 más el offset de anio de var1, lo que nos da una dirección 'd2', por consiguiente d1 es distinto de d2, ya que las direcciones de var y var1 son distintas. Esto es parecido a decir, si tenemos dos casa exactamente iguales, una en Av. Luro al 1000 y otra en San Luis 500, el comedor de ambas está a 10 metros de la entrada principal, pero esto no quiere decir que una casa esté sobre la otra, sino que si la memoria de nuestro sistema fuese la ciudad de Santa Rosa, desde la dirección Av. Luto al 1000 más estos 10 metros, llego al comedor de una casa y, desde la dirección San Luis 500 más 10 metros, llego al comedor de la segunda casa. En este último ejemplo las direcciones de las casas en la ciudad representarían las direcciones de memoria, y los 10 metros para llegar al comedor desde la entrada principal, el offset.

Lectura y Asignación de Valores a un Tipo de Datos Definido por el Usuario

El tipo de datos definidor por el usuario, almacena su contenido en cada uno de los campos que lo componen, esto es:

- Para almacenar el año de una fecha, lo hace dentro de su campo anio,

- Para almacenar el mes de una fecha, lo hace dentro de su campo mes y,
- Para almacenar el día de una fecha, lo hace dentro de su campo dia.

Para acceder al contenido de los campos integrantes del tipo de datos definidos por el usuario, se utiliza el nombre de la variable del tipo de datos definido por el usuario (por ejemplo, según el programa 'struct1.cpp', 'var'), un '.', y el nombre del campo al que se desea acceder. Es importante que no deje espacios entre el '.' y el nombre de la variable y el campo al que se desea acceder, por ejemplo: "var.anio". Veamos un programa de ejemplo:

Archivo: **struct2.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
6               short int mes;
7               short int dia;
8               };
9
10      void main(void)
11      {
12       fecha var;
13
14       var.anio=2004;
15       var.dia=10;
16       var.mes=3;
17
18       cout<<"Año: "<<var.anio;
19       cout<<"Mes: "<<var.mes;
20       cout<<"Dia: "<<var.dia;
21      }
```

Note que asignar valores a un campo de la estructura, o bien hacer uso de su contenido, sea para mostrarlo o lo que fuese, es relativamente simple. Veamos ahora, si utilizásemos más de una variable del tipo y se le asignan distintos valores a sus campos, a ver que ocurre:

Archivo: **struct3.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
6               short int mes;
7               short int dia;
8               }var1, var2, var3;
```

```
9
10      void main(void)
11      {
12       fecha var;
13
14       var.anio=2004;
15       var.dia=10;
16       var.mes=3;
17       var1.anio=2004;
18       var2.anio=var3.anio=var1.anio;
19       var1.mes=4;
20       var2.mes=5;
21       var3.mes=6;
22       var1.dia=11;
23       var2.dia=12;
24       var3.dia=13;
25
26       clrscr();
27
28       cout<<"A¤o var: "<<var.anio;
29       cout<<"\nMes var: "<<var.mes;
30       cout<<"\nDia var: "<<var.dia;
31       cout<<"\n\nA¤o var1: "<<var1.anio;
32       cout<<"\nMes var1: "<<var1.mes;
33       cout<<"\nDia var1: "<<var1.dia;
34       cout<<"\n\nA¤o var2: "<<var2.anio;
35       cout<<"\nMes var2: "<<var2.mes;
36       cout<<"\nDia var2: "<<var2.dia;
37       cout<<"\n\nA¤o var3: "<<var3.anio;
38       cout<<"\nMes var3: "<<var3.mes;
39       cout<<"\nDia var3: "<<var3.dia;
40      }
```

El programa “struct3.cpp”, provoca la siguiente salida por pantalla:

```
Año var: 2004
Mes var: 3
Dia var: 10

Año var1: 2004
Mes var1: 4
Dia var1: 11

Año var2: 2004
Mes var2: 5
Dia var2: 12

Año var3: 2004
Mes var3: 6
Dia var3: 13
```

Figura 37: “Salida por pantalla del programa struct3.cpp”

Es claro, que al margen de que los campos poseen el mismo nombre y hasta el mismo offset, no s sobrescriben sus contenidos, ya que las direcciones del espacio de

memoria de sus variables (var, var1, var2 y var3) son distintos. Veamos cómo se representarían en memoria los valores asignados a los distintos campos de la estructuras hasta la línea 25:

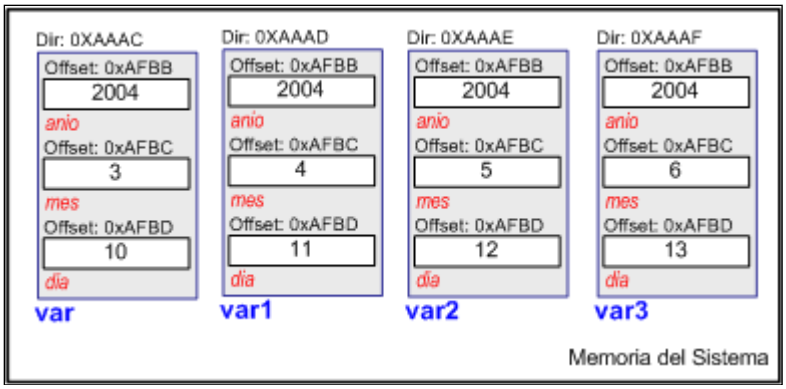


Figura 38: "Conceptualización en Memoria de la estructura definida por el usuario del programa 'struct3.cpp' "

Cuando en la línea 28 se le pide mostrar el contenido de 'var.anio', el C/C++ accede a la dirección de memoria 0xAAAC+0xAFBB, ubicando el espacio de memoria real del campo. Mientras que para acceder al contenido, o bien asignar un valor, del campo anio de var1, se accede al espacio de memoria sumando 0xAAAD+0xAFBB. Esto nos clarifica el hecho de que dos variables de un mismo tipo definido por el usuario, aunque tengan los mismos nombres y offset para acceder a los valores de sus campos, poseen direcciones de memorias asociadas a las mismas variables, distintas tal y como se muestra en la figura 38.

Tipos de Datos Definidos por el Usuario sin Límites

Uno de las incógnitas frecuentes, posterior a la presentación de tipos de datos definidos por el usuario, es ¿Puedo definir los tipos de datos definidos por el usuario que quiera? Y ¿de la forma que quiera?, la respuesta es sí, aquí encontramos parte del poder de C/C++. Veamos a través de un ejemplo, la definición de varios tipos de datos cualesquiera:

Archivo: **struct4.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
6               short int mes;
7               short int dia;
8               };
9
10      struct alumno{
11              char nombre[20];
12              char apellido[20];
13              fecha f_nacimiento;
14              float promedio;
```

```
15          };
16
17     struct libros{
18          char titulo[40];
19          char autor[40];
20          char isbn[20];
21          fecha f_publicacion;
22          float precio;
23          };
24
25     void main(void)
26     {
27      fecha var;
28      libros varlibro;
29      alumno varalumno;
30     }
```

Note que el hecho de definir múltiples estructuras de datos, a la forma y cantidad deseada del programador, no es un problema en C/C++. Veamos un aspecto interesante del programa en las líneas 13 y 21; note que el tipo de datos del campo f_nacimiento, del tipo de datos definido por el usuario "alumno", es fecha (nuestro primer tipo de dato propio). Esto último es completamente válido, ya que se le pide al compilador que interprete las variables f_nacimiento (línea 13) y f_publicacion (línea 21) de sus respectivas estructuras, como el tipo de datos fecha, a esas alturas ya definido. Si en cambio, la definición de la estructura 'libros', tal y como se define desde la línea 17 hasta la 23, estuviese antes de la definición del tipo de datos 'fecha' provocaría un error de compilación del tipo "fecha es un tipo de datos desconocido". Modifiquemos el anterior programa para localizar el punto donde se provocaría dicho error:

Archivo: **struct4.cpp (Modificado)**

```
1      #include <iostream.h>
2      #include <conio.h>
3
4      struct libros{
5              char titulo[40];
6              char autor[40];
7              char isbn[20];
8              fecha f_publicacion;
9              float precio;
10             };
11
12     struct fecha{
13             short int anio;
14             short int mes;
15             short int dia;
16             };
17
18     struct alumno{
19             char nombre[20];
```

```
20              char apellido[20];
21              fecha f_nacimiento;
22              float promedio;
23              };
24
25      void main(void)
26      {
27       fecha var;
28       libros varlibro;
29       alumno varalumno;
30      }
```

En la línea 8 se provocaría el error de compilación anteriormente enunciado, en efecto, el tipo de datos fecha, recién es definido entre las líneas 12 y 16.

Veamos como lucirían las anteriores estructuras del programa "struct4.cpp" original, en memoria:

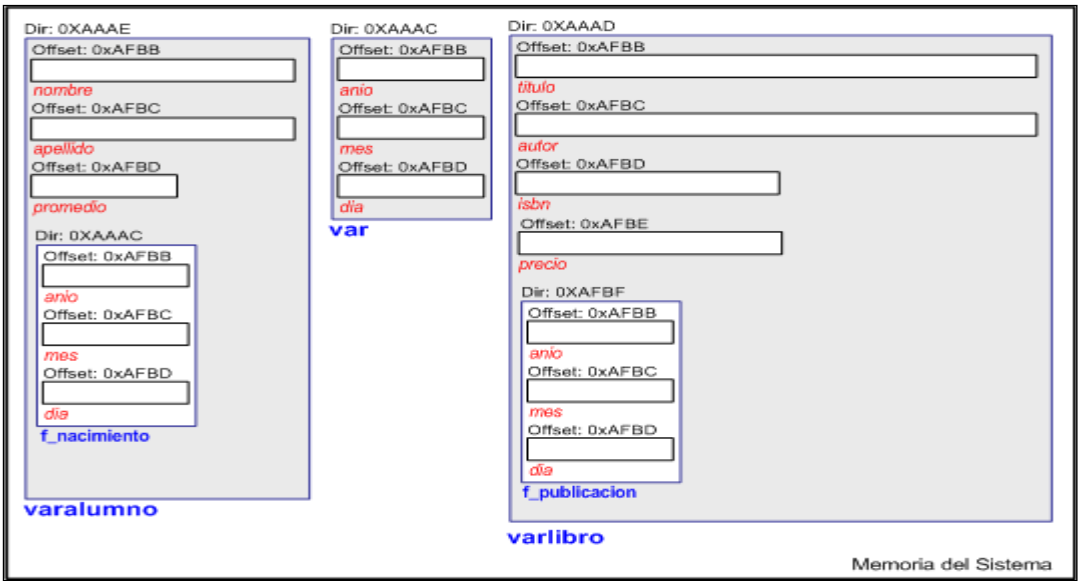


Figura 38: "Conceptualización en Memoria de la estructura definida por el usuario del programa 'struct3.cpp' "

Como se puede ver, el C/C++ permite el anidamiento de tipos de datos definidos por el usuario, dentro de otros, en la cantidad y formas que se deseen. La forma de acceder por ejemplo, al campo anio de la variable varalumno, es como sigue:

```
Varalumno.f_nacimiento.anio=2004;
```

Para acceder al campo f_nacimiento de la variable varalumno, del tipo alumno, se colocaba el nombre de la variable, un punto y el nombre del campo. Como en este último caso, el campo también es un tipo definido por el usuario, para acceder a su contenido, se coloca un punto más y se indica el campo al cual se desea acceder (anio en el anterior ejemplo).

Tipos de Datos Definidos por el Usuario y Punteros

Como se ha visto y analizado hasta esta altura del capítulo, sabemos que sobre cualquier tipo de datos nativo del C/C++ podemos definir variables punteros que direccionen a su posición de memoria. En efecto, esto es posible también para los tipos

de datos definidos por el usuario, y la forma de definición del puntero a la variable del tipo de dato del usuario es igual a la del puntero tradicional. Veamos un ejemplo:

Archivo: **struct5.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
6               short int mes;
7               short int dia;
8               };
9
10      void main(void)
11      {
12       fecha var;
13       fecha *ptrvar=&var;
14      }
```

El puntero del tipo *fecha*, se define en la línea 13, donde también se inicializa con la dirección de memoria de la variable 'var'. Veamos como se representa en memoria:

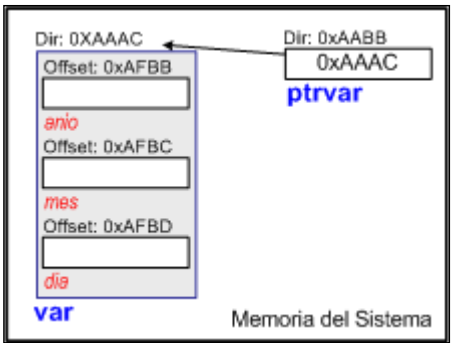


Figura 39: "Conceptualización en Memoria del programa 'struct5.cpp' "

Para acceder al contenido de la variable apuntada por un puntero, utilizábamos el operador '*', por lo que *ptrvar sería sinónimo de 'var', pero ésta última es una estructura, ¿Cómo se accede a los campos de var? Como *ptrvar es sinónimo de var, y como tradicionalmente con var se accedía con 'var.anio' o 'var.mes' a los campos respectivos, entonces debería ser *ptrvar la forma en que a través de '*ptrvar.anio', '*ptrvar.mes' se accedan a sus campos. Bueno, lo dicho es lógico pero no es así en C/C++, por esas cosas del C/C++, la forma de acceder a un campo de la variable apuntada por el puntero (del tipo definido por el usuario) es a través del operador '→' Veamos un ejemplo:

Archivo: **struct6.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
```

```
5               short int anio;
6               short int mes;
7               short int dia;
8               };
9
10      void main(void)
11      {
12       fecha var;
13       fecha *ptrvar=&var;
14
15       var.anio=2004;
16       var.mes=3;
17       var.dia=10;
18
19       clrscr();
20
21       cout<<ptrvar->anio<<"/"<<ptrvar->mes<<"/"<<ptrvar->dia;
22      }
```

El programa “struct6.cpp” muestra la cadena “2004/10/3” en pantalla. Para tranquilidad del lector, cuando se usan punteros asociados a tipos de datos definido por el usuario, se utiliza el operador ‘→’, que en modo de texto se escribe con un signo menos y un mayor: ‘->’.

Veamos ahora una variante del uso de puntero, utilicemos un puntero del tipo de datos que en su momento se está definiendo:

Archivo: **struct7.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
6               short int mes;
7               short int dia;
8               fecha* next;
9               };
10
11      void main(void)
12      {
13       fecha var;
14       fecha var2;
15
16       fecha *ptrvar=&var;
17       var.next=&var2;
18
19       var.anio=2004;
20       var.mes=3;
21       var.dia=10;
```

```
22      var2.anio=2005;
23      var2.mes=4;
24      var2.dia=8;
25
26      clrscr();
27
28      cout<<ptrvar->anio<<"/"<<ptrvar->mes<<"/"<<ptrvar->dia;
29      cout<<"\n";
30      cout<<ptrvar->next->anio<<"/"<<ptrvar->next->mes<<"/"<<ptrvar->next->dia;
31     }
```

Note que al asignar la dirección de la variable var2 (del tipo fecha), se utiliza el operador punto con el puntero next y var (línea 17), ya que next es un campo más de la estructura de 'var' definida en tiempo de compilación. Pero al acceder al contenido de var2, debemos localizar el puntero next con 'ptrvar->next', y de él el campo anio (por ejemplo), entonces queda 'ptrvar->next->anio' como la forma de acceso a dicho campo.

C/C++ permite definir variables punteros del mismo tipo de datos que se está definiendo, aún cuando la definición del tipo de datos no terminó aún (vea la línea 8). Esto último, es una poderosa herramienta para el desarrollo de estructuras de datos complejas, sobre la cual volveremos en capítulos posteriores.

Inicialización de los Campos de un Tipo de Datos Definido por el Usuario

C/C++ permite inicializar las variables de un tipo de datos definido por el usuario, a través de una función especial denominada "constructor", la cual lleva el nombre del tipo de datos seguido por un juego de paréntesis "()", con las instrucciones de inicialización encerrada entre llaves "{}", Veamos un ejemplo:

Archivo: **struct8.cpp**

```
1      #include <iostream.h>
2      #include <conio.h>
3
4      struct fecha{
5              short int anio;
6              short int mes;
7              short int dia;
8              fecha()
9              {//Inicio del Bloque del constructor
10              anio=0;
11              mes=0;
12              dia=0;
13             }//Fin del Bloque del Constructor
14             };
15
16     void main(void)
17     {
18      fecha var;
19     }
```

Es, por definición, obligatorio que la función de inicialización del tipo definido por el usuario, tenga el mismo nombre (tenga en cuenta las mayúsculas y minúsculas también, recuerde que el C/C++ es sensible a las mismas) que el tipo de datos.

Gestión Dinámica de Memoria

Hasta el momento, se ha trabajado sobre memoria estática, o también conocida como "solicitud de memoria en tiempo de compilación", pero ¿que ocurriría si cuando el usuario está ejecutando un programa se requiere mayor cantidad de memoria?, ¿Cómo haríamos para peticionar mayor memoria y de qué forma?, este tema es el punto central de este apartado y sobre el cual iniciaremos su trato en este preciso momento.

Concepto

Cuando definíamos variables en un programa de C/C++, se entendía que al ejecutar el programa, se solicitaría la cantidad necesaria de memoria para abastecer dichas variables, y en caso de no disponer de la cantidad suficiente de memoria en la PC, el programa directamente no se iniciaba. Pero una vez iniciado en condiciones normales, tendríamos una situación como la que muestra el siguiente programa:

Archivo: **gdim1.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        int a,b,c;
7
8        clrscr();
9
10       cout<<"Ingrese a: ";
11       cin>>a;
12
13       cout<<"\nIngrese b: ";
14       cin>>b;
15
16       cout<<"\nIngrese c: ";
17       cin>>c;
18
19       cout<<"\n'a' vale: "<<a;
20       cout<<"\n'b' vale: "<<b;
21       cout<<"\n'c' vale: "<<c;
22      }
```

En memoria, al momento de la ejecución del programa en la línea 7, se tendría la siguiente situación:

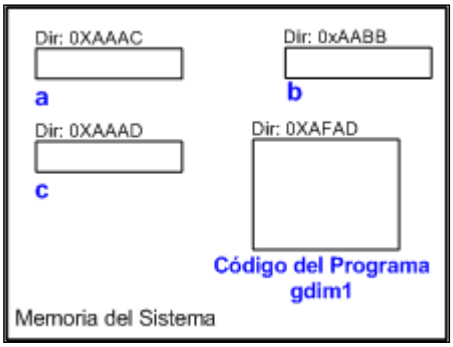


Figura 40: "Conceptualización en Memoria del programa 'gdim1.cpp' "

Este programa solicita la memoria para las tres variables enteras y la memoria necesaria para albergar el código del programa en memoria, no es capaz de solicitar más memoria de acuerdo a su estructura actual. Para mucho de los casos de programación rutinaria, esta situación no es un inconveniente, pero cuando ingresamos en estructuras de datos e incrementamos la complejidad de los algoritmos, se hace necesario solicitar memoria en tiempo de ejecución, a esto último se lo denomina "Gestión Dinámica de Memoria".

Cuando solicitamos memoria en tiempo de ejecución, dicha situación se plantea por una necesidad real de mayor espacio de memoria para el programa, donde el mismo, pasaría a negociar directamente con el sistema operativo para abastecerse de la misma. Esta solicitud se daría como sigue:

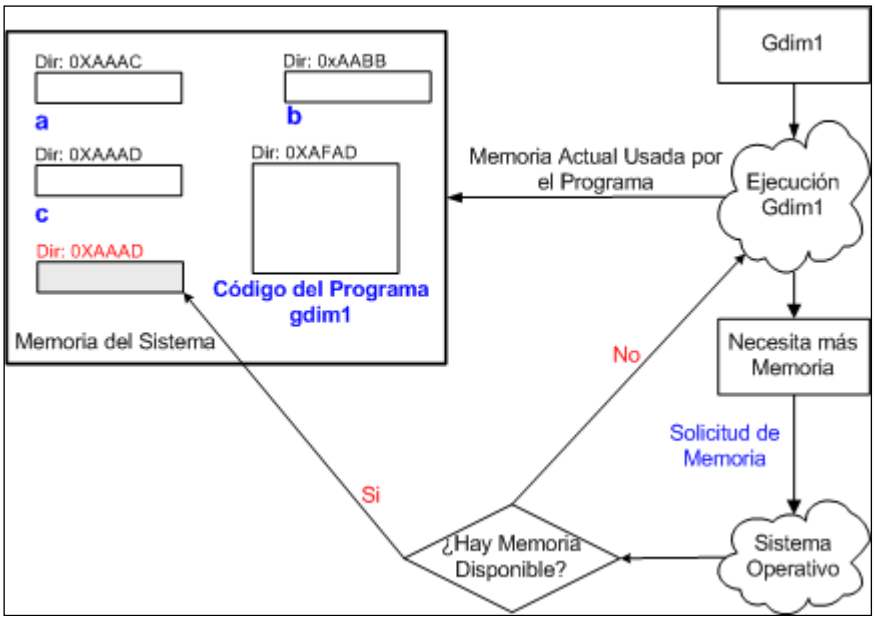


Figura 41: "Conceptualización en Memoria del programa 'gdim1.cpp' ante una Necesidad de Memoria Adicional"

Supongamos que el programa 'gdim1.cpp' es modificado y se convierte en apto de manejar memoria dinámica. En un momento dado de su ejecución necesita más memoria, por lo que efectúa la petición al sistema operativo. El Sistema operativo, si dispone de memoria RAM libre, en la forma y cantidad solicitada por el programa, se la devuelve, de lo contrario, avisa que no es posible otorgar más memoria de la actual,

porque no existe disponibilidad. De este procedimiento se desprenden los siguientes conceptos que:

- En la Gestión dinámica de Memoria existe un mecanismo formal para solicitar la cantidad de memoria adicional al sistema operativo,
- Si el sistema operativo dispone de dicha memoria adicional, la reserva y retorna la dirección de la memoria adicional reservada,
- Debe existir en el programa, una estructura capaz de albergar la dirección de memoria devuelta por el sistema operativo, y hacer uso de la misma,
- Si el sistema operativo no obtiene la cantidad de memoria adicional solicitada, dispone de un mecanismo para indicarle al programa dicha imposibilidad.

Implementación de Gestión Dinámica en C/C++

Dentro de la estructura formal del C/C++, existen los mecanismos necesarios como para solicitar memoria, recibirla y hacer uso de la misma, como así también, informarse en aquellos casos en que no es posible conseguirla.

El C/C++ emplea obligadamente para solicitud y liberación de memoria, punteros, en efecto, son la estructura por excelencia para manipular direcciones de memoria. Para solicitar memoria se utiliza el operador "*new*", mientras que para liberar memoria solicitada que ha sido asignada por el sistema operativo, se utiliza el operador "*delete*".

Es importante mencionar que toda memoria dinámica solicitada por el programa, se convierte en una responsabilidad exclusiva del mismo en cuanto a su liberación, ya que el sistema operativo no lo hace. Por lo tanto, si empleásemos "new" para solicitar memoria, deberemos recordar siempre usar "delete" para liberarla, de lo contrario, quedará la memoria como reservada para el sistema operativo, aún cuando el programa haya finalizado su ejecución, disminuyendo los recursos de memoria de su equipo o servidor.

El formato de solicitud de memoria, suponiendo a 'p' como un puntero a entero, es:

```
p=new int;
```

Al hacer esto, le decimos al sistema operativo: "Sistema Operativo, necesito memoria suficiente para albergar un entero, y si puede obtenerla (o no), devuelve la dirección en el puntero p, que es del tipo 'int'".

En efecto, para apuntar a un espacio de memoria 'int' necesito un puntero 'int', y la pregunta que naturalmente surge es ¿Puedo solicitar memoria de cualquier tipo de datos?, ¿Incluso del tipo de datos definidos por el usuario?, la respuesta es sí. Veamos un ejemplo de cómo solicitar memoria al sistema operativo:

Archivo: **gdim2.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        int *p=NULL;
```

```
7       int a=7;
8
9       p=new int;
10
11      clrscr();
12
13      if(p==NULL)
14       cout<<"No hay memoria";
15      else
16       cout<<"Memoria albergada en "<<p;
17
18       cout<<"\n\nIngrese un Entero: ";
19       cin>>*p;
20
21       cout<<"\nEl Entero ingresado es: "<<*p;
22       cout<<"\nLa Suma del Entero Ingresado y 'a' es: "<<(*p)+a;
23       delete p;
24
25       p=NULL;
26     }
```

El programa ‘gdim2.cpp’ define un puntero entero ‘p’, quien será el encargado de manipular la memoria solicitada al sistema operativo en caso de obtenerla. La definición del puntero ‘p’ se da en la línea 6 junto con su inicialización. La inicialización de cualquier puntero, emplea la palabra reservada “NULL” (todo en mayúscula), quien almacena una dirección nula en el mismo y sirve a los fines de conocer si el puntero está direccionando memoria actualmente o no (si posee el valor NULL se entiende que es un puntero sin memoria asociada).

Mediante la línea 9, se le solicita al sistema operativo la cantidad de memoria solicitada, en este caso, se asume la necesidad de un único entero (Volveremos sobre este tema con posterioridad). Recordemos que en los tipos nativos de C/C++, “*p” era un sinónimo de la variable a la que apuntaba, como en este caso la gestión de memoria es dinámica, se asume a *p como nombre de esa posición de memoria, y como tal se puede manipular como cualquier otra variable (Vea la línea 19, se carga el valor ingresado por teclado en la posición de memoria reservada y apuntada por ‘p’), realizando sobre ella incluso, operaciones matemáticas (vea la línea 22).

En la línea 13, mediante la pregunta natural de si p tiene un valor NULL, sabremos si el sistema operativo pudo o no conseguir la memoria solicitada. Cuando efectuamos la solicitud mediante ‘new’, el mismo nos puede retornar dos valores:

- Una posición de memoria donde albergó la memoria solicitada ó
- Un “NULL” indicando que la memoria no pudo ser obtenida

De allí la importancia de la pregunta de la línea 13.

Cuando no se requiere la memoria reservada, se emplea ‘delete’ para liberarla y devolverla al sistema operativo. Es una buena práctica de programación, inmediatamente después de utilizar ‘delete’, asignarle al puntero el ‘NULL’. En caso de que no lo hiciésemos, no tendríamos forma de saber si el puntero tiene memoria

asociada o no, y podríamos querer hacer uso de una supuesta memoria reservada, que en realidad no es así, y sobrescribir la memoria con el riesgo de inestabilizar el sistema.

Veamos qué ocurre en memoria con el programa 'gdim2.cpp'

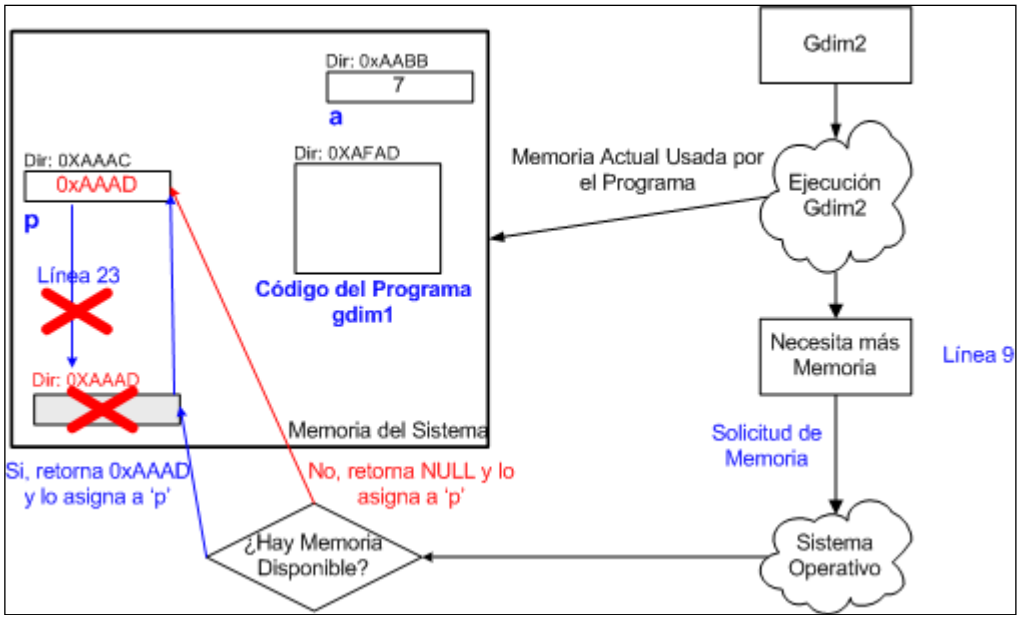


Figura 42: "Conceptualización en Memoria del programa 'gdim2.cpp' ante una Necesidad de Memoria Adicional"

El programa "gdim2.cpp" hasta la línea 8, tiene reservada memoria para su programa, su variable entera inicilizada con 7 y su puntero a entero inicializado con "NULL" (el cual no se ha puesto en el gráfico, a los fines de no entorpecer la explicación de solicitud de memoria).

En la línea 9, a través del operador 'new' se solicita memoria al sistema operativo, la especificación del tipo de datos, le indica la cantidad necesaria de enteros: uno, que implican dos bytes. Si no consigue la memoria solicitada, a través del operador de asignación, el 'new' le asigna 'NULL' al puntero 'p', pero si obtuviese la memoria la reserva y, el operador new, a través del operador de asignación, le asignaría a 'p' la dirección de la nueva memoria, para que la pueda utilizar en el programa.

Cuando no se requiere más de la memoria, se utiliza el operador 'delete' para liberarla, lo que libera la memoria reservada en '0xAAAD' y note que no existe actualización del valor de 'p' en cuanto a la dirección de la memoria a la que apunta. Si no especificamos, posterior al 'delete', 'p=NULL', p mantendría como contenido '0xAAAD' y la consecuencia está a la vista. Si luego del 'delete' no anulamos p, y preguntamos: ¿p apunta a alguna dirección de memoria? La respuesta peligrosamente es sí; la forma en que se pregunta es la misma que en la línea 13 del programa 'gdim2.cpp'.

Gestión Dinámica de Arreglos

Hasta ahora se ha trabajado sobre solicitud de memoria para variables simples, pero ¿qué ocurre si necesitamos gestionar memoria dinámica para una cadena de caracteres o incluso un arreglo de enteros?, veamos como se adapta el código de C/C++ para suplir esta necesidad:

Archivo: **gdim3.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void main(void)
5       {
6        int *p=NULL;
7        int a=7;
8
9        p=new int[10];
10
11       clrscr();
12
13       if(p==NULL)
14        cout<<"No hay memoria";
15       else
16        cout<<"Memoria albergada en "<<p;
17
18        for(int i=0;i<10;i++)
19          {
20           p[i]=i;
21          }
22
23        for(i=0;i<10;i++)
24          {
25           cout<<"\nEl valor de 'i'*'a' en "<<i<<" es: "<<(p[i])*a;
26          }
27
28        delete p;
29
30        p=NULL;
31      }
```

Note que el presente programa solicita memoria para albergar un arreglo de 7 enteros, lo único que cambia en la solicitud de memoria, es que se agrega un '7' entre dos corchetes inmediatamente posterior al tipo de datos. La forma de preguntar si se pudo o no obtener la memoria es igual y, note esta espectacular facilidad del C/C++: "Permite que manipule la misma sintaxis en los arreglos, que aquella empleada en los arreglos definidos estáticamente en su programa", en otras palabra, 'p' se convierte en el nombre del arreglo y para acceder a su posición 'i', lo único que debe hacer para asignar o leer valores es 'p[i]'.

Veamos la salida del anterior programa en pantalla:

```
Memoria albergada en 0x13fe0f4c
El valor de 'i'*'a' en 0 es: 0
El valor de 'i'*'a' en 1 es: 7
El valor de 'i'*'a' en 2 es: 14
El valor de 'i'*'a' en 3 es: 21
El valor de 'i'*'a' en 4 es: 28
El valor de 'i'*'a' en 5 es: 35
El valor de 'i'*'a' en 6 es: 42
El valor de 'i'*'a' en 7 es: 49
El valor de 'i'*'a' en 8 es: 56
El valor de 'i'*'a' en 9 es: 63
```

Figura 43: "Salida por pantalla del programa gdim3.cpp"

Vea como a través del ciclo de la línea 18 se inicializa p[0] en cero, p[1] en uno, p[2] en dos y así sucesivamente. Luego, el siguiente for de la línea 23, multiplica cada uno de los valores por el valor de la variable 'a' y lo muestra en pantalla.

Veamos ahora, un ejemplo de solicitud de memoria para una cadena de caracteres donde el usuario, cargará un mensaje que luego mostraremos por pantalla:

Archivo: **gdim4.cpp**

```
1       #include "iostream.h"
2       #include "conio.h"
3       #include "string.h"
4       #include "stdio.h"
5
6       void main(void)
7       {
8        char* p=NULL;
9
10       //Estimamos que el mensaje no ocupar  m s de 50 bytes
11       p=new char[50];
12
13       clrscr();
14
15       if(p==NULL)
16          {
17           cout<<"\nMemoria Insuficiente";
18           return;
19          }
20
21       cout<<"\nIngrese un Mensaje: ";
22       //Funci¢n empleada para el Ingreso de Cadena de Caracteres:
23       gets(p);
24
25       int a=strlen(p);
26       cout<<"\n\nMensaje: "<<p;
27       cout<<"\nLongitud: "<<a;
28
```

```
29      delete p;
30      p=NULL;
31     }
```

Note la simplicidad de C/C++ para manipular cadenas de caracteres dinámicas, lo único que se requiere para manipularlas, es disponer de un puntero del tipo 'char' (línea 8) y solicitar la cantidad de memoria necesaria para almacenar la cadena entre corchetes (línea 11, capacidad de almacenamiento de 49 caracteres más el carácter especial de fin de cadena '\n').

Hasta el momento no hemos trabajado con programas en los que se requiera que el usuario cargue cadenas de caracteres. En éstos casos, no se emplea el flujo 'cin>>p' (siendo 'p' el puntero a la cadena de caracteres), sino que se utiliza la función de la librería 'string.h': gets(char *), note la simplicidad de su uso en la línea 23. Si no fuese una cadena generada dinámicamente como en este caso y, se tratase de un arreglo definido a la forma 'char cad[20]', su mecanismo de uso es idéntico (para este último ejemplo sería gets(cad)).

Gestión Dinámica de Tipos de Datos Definidos por el Usuario

Es factible en C/C++, solicitar memoria dinámica para los tipos de datos definidos por el usuario. En efecto, luego de que los tipos de datos definidos por el usuario son declarados, pueden emplearse como cualquier otro tipo de datos de C/C++.

Veamos un ejemplo sobre el tipo de datos fecha, de manera de gestionar un arreglo de tipos de datos fecha y una única variable fecha en forma dinámica:

Archivo: **gdim5.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct fecha{
5               short int anio;
6               short int mes;
7               short int dia;
8               };
9
10      void main(void)
11      {
12       fecha *p=NULL;
13       fecha *arreglo=NULL;
14
15       p=new fecha;
16       arreglo=new fecha[5];
17
18       p->anio=2004;
19       p->mes=3;
20       p->dia=10;
21
22       for(int i=8,j=0;i<13 && j<5;i++,j++)
23       {
```

```
24        arreglo[j].anio=2004;
25        arreglo[j].mes=4;
26        arreglo[j].dia=i;
27      }
28
29      clrscr();
30      cout<<"\nFecha Individual con 'p'\n";
31      cout<<p->dia<<"/"<<p->mes<<"/"<<p->anio<<"\n";
32      cout<<"\nFechas del Arreglo 'arreglo':\n";
33      for(i=0;i<5;i++)
34      {
35       cout<<arreglo[i].dia<<"/"<<arreglo[i].mes<<"/"<<arreglo[i].anio<<"\n";
36      }
37
38      delete arreglo;
39      delete p;
40      arreglo=NULL;
41      p=NULL;
42     }
```

Note que no existen prácticamente variantes para solicitar un tipo de datos definido por el usuario de un tipo de datos nativo; sino que las variantes, provienen del manejo de la estructura. Cuando deseábamos acceder a un campo de una estructura cualquiera a través de un puntero, empleábamos el operador '→' (un menos y un mayor '->', en modo texto), hasta el momento, desde la línea 18 a la 20 esto se mantiene. Donde el C/C++ nos facilita este engorroso trabajo, es al definir arreglos, allí nos permite el uso del operador '[]' y el acceso a los campos a través del operador punto. Esto último evita al programador ingresar en la aritmética de punteros, para que tenga una idea:

Forma Simple	Equivalencias
arreglo[i].anio	`(*(arreglo+i)).anio` `(arreglo+i)->anio`

Tabla 11: "Equivalencias del Operador '[]' con la Aitmética de Punteros"

Note que si realmente tuviésemos que emplear aritmética de punteros para este caso simple, la operatoria se tornaría sumamente engorrosa. De todas formas es posible acceder a los contenidos direccionados por el puntero, en cualquiera de las formas enunciadas.

Referencias

- "C Guía de Autoenseñanza"; Herbert Shildt; Editorial McGraw-Hill; España, 1994.
- "El Lenguaje de Programación C"; Kernighan y Ritchie; Editorial Prentice Hall; México, 1991.
- "C++ Guía de Autoenseñanza"; Herbert Schildt; Editorial McGraw-Hill; España, 1995.
- "Progrese con Visual C++"; David Kruglinski; Editorial McGraw-Hill; España 1994.
- "Microsoft C/C++7, Manual de Referencia"; William Murria y Chris Pappas; Editorial McGraw-Hill; España 1994.
- "Borland C++ 3.1"; Ted Faison; Editorial SAMS; México, 1993.
- "Borland C++ Builder 4 Unleashed"; Editorial SAMS; EE.UU, 1999.
- "Programación con C++ Builder 4"; Francisco Charte; Editorial Anaya Multimedia; España, 1999.

2

Recursividad

Concepto de Recursividad

Alguna vez alguien nos enseñó en Literatura "No se define lo definido", claro está que es válido para el ámbito literario, pero contrariamente a lo que uno pensaría, en la matemática es sumamente común y útil, en cierta forma, definir lo definido. Un ejemplo clásico es la función factorial, que se define como:

n! → Si "n" es distinto de 0, (n-1)!
→ Si "n" es igual a cero, la función "n!" es 1.

Analicemos cómo trabajaría ante su invocación, supongamos 5!:

n!	Valor Actual de n	¿n es 0?	Calcular el Valor de n!	Valor Acumulado
5	5	no	5*(5-1)!	5*
	4	no	4*(4-1)!	5*4*
	3	no	3*(3-1)!	5*4*3*
	2	no	2*(2-1)!	5*4*3*2
	1	no	2*(1-1)!	5*4*3*2*1
	0	si	1	5*4*3*2*1*1=**120**

Tabla 12: "Concepto de Recursividad a través de la Función Factorial"

Cuando deseamos obtener el factorial de 5, por definición, es igual a decir 5*(5-1)!, ésta misma acción es una invocación d función recursiva, ya que la misma función se auto invoca aún cuando ella no ha terminado de mostrar un resultado. A simple vista, parecería cosas de locos, algo que nunca nos daría un valor y que implicaría un ciclo infinito, pero notemos que hay una condición de corte recursivo, y ésta es la pregunta ¿n es cero?, en caso de ser falsa prosigue la recursividad, pero si es verdadera retorna automáticamente un uno, permitiendo de esta forma cerrar el ciclo a auto invocación de funciones y brindar un resultado concreto.

Veamos como analizar la misma función factorial, a través de un programa en C/C++:

Archivo: **recurfac.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       float factorial(float);
5
6       void main(void)
7       {
8        float nro=0;
9
10       clrscr();
11       cout<<"Ingrese el N£mero a Calcular su factorial: ";
12       cin>>nro;
13       cout<<"El factorial de "<<nro<<" es "<<factorial(nro);
14      }
15
16      float factorial(float num)
17      {
```

```
18        return (num==0)?1:(num*factorial(num-1));
19      }
```

El programa "recurfac.cpp", implementa una función factorial simple en C/C++, solicita que se cargue un número al usuario y a partir de allí, se calcula y muestra su factorial.

Note que una invocación recursiva es relativamente simple, tan solo se invoca la función dentro de la misma función, teniendo en cuenta la posición de la condición de corte (vea la línea 18).

En Memoria, al momento de las invocaciones de las funciones, ocurre lo siguiente:

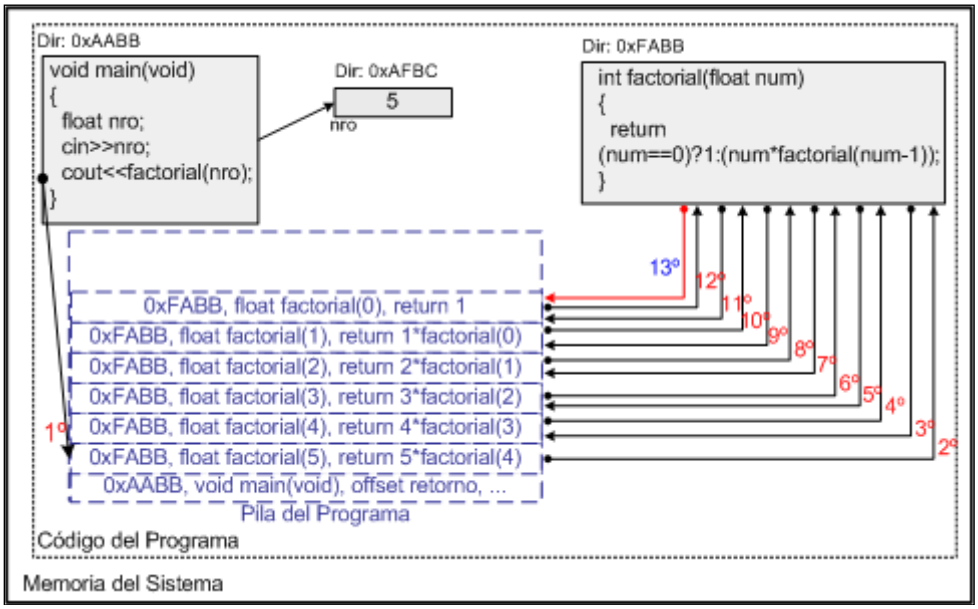


Figura 44: "Análisis en Memoria de la Recursividad del programa recurfac.cpp"

Note que al momento de la invocación inicial de la función "factorial" (1º), antes de ingresar en el código de la función, se registra el llamado de la función en la pila del programa. En efecto, el registrar el llamado a función en la pila, tiene por finalidad saber en que línea deberá seguir ejecutándose el programa, una vez devuelto el valor de la función, qué tipo de valor debe retornar, qué parámetros se pasan, entre otras cuestiones internas al C/C++. Registrada el llamado a la función, se ejecuta el código con los valores de los parámetros pasados por el programa principal (2º). En alguna línea de la función factorial, se produce una nueva invocación de sí misma, lo que provoca que se registre la nueva invocación de la función (3º) con los nuevos valores, y se ejecute la función factorial nuevamente (4º). De esta forma se continuará registrando funciones en la pila y ejecutando su código hasta que se encuentre la condición de corte en la función (13º, Recuerde que la condición de corte de la función factorial era "si n=0 retorne 1"), allí se produce el desenlace de la función, que analizaremos en el gráfico a continuación:

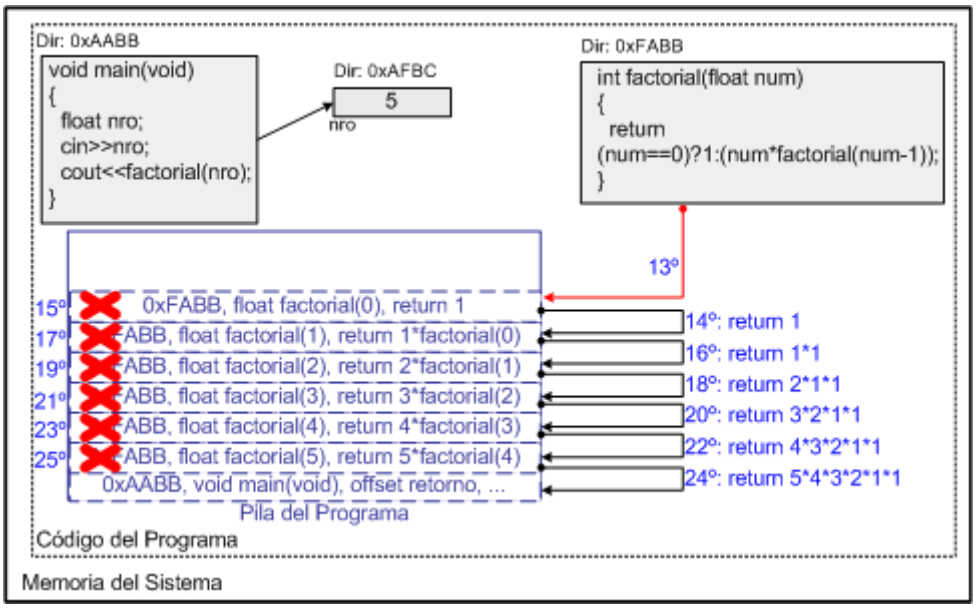


Figura 45: "Análisis en Memoria de la Recursividad del programa recurfac.cpp"

Vea que cuando la función factorial encuentra su condición de corte (13º), inicia su "retorno" al programa, ¿Cómo lo hace?, vea que primero retorna el valor 1, por la misma condición de corte (14º), luego se encarga el programa se encarga de eliminar el registro de la pila (15º). El valor retornado en 14º, era esperado por el registro de la penúltima línea de la pila, quien al obtener dicho valor, produce su retorno (16º) y se procede a eliminar su registro de la pila (17º), de igual forma el siguiente registro esperaba el valor de la función que invocó (penúltima línea de la pila), para proseguir su ejecución y retornar su valor (18º), luego elimina su registro en la pila (19º), y así sucesivamente hasta la primera invocación de la función factorial (24º), quien retornará el valor acumulado y eliminará su registro (25º) para continuar la normal ejecución del "main".

Problemas de la Recursividad

La Recursividad, como ve, es una forma elegante de slucionar muchos problemas del mundo real, pero notará que al momento de su implementación hay muchos factores a tener en cuenta, como ser:

- **Presencia de la Condición de Corte de la Recursividad:** Debe estar en primera instancia definida la condición de corte, su ausencia implicará una invocación infinita de funciones, las cuales producirán un inminente "Snack Overflow" (Sobrecarga de Pila), y por consiguiente el "Tilde" del programa.
- **Condición de Corte Bien Definida:** No solo basta que la condición de corte esté definida, sino que es necesario que la condición de corte en laguna situación dé "Verdadero", de lo contrario caeremos en el mismo problema de invocación infinita de funciones. Incluso si la condición de corte da verdadera en alguna situación, corremos el riesgo de que esté mal definida, por ejemplo: suponga el caso que en la función factorial, en lugar de preguntar si n=0, se pregunte si n<0, ¿Qué nos devolvería la función?, lo analizaremos en el próximo apartado.
- **Ubicación Correcta de la Condición de Corte:** No solo es necesario que la condición de corte esté presente y se encuentre bien definida, debe estar bien

ubicada, de lo contrario es posible que nunca se llegue a la misma y se caiga nuevamente en una invocación infinita de funciones.

- **Precisión de las Variables Involucradas:** Tenga en cuenta los valores que deberán almacenar las variables que son involucradas en las llamadas recursivas. Por ejemplo, la función factorial implementada anteriormente, claramente no puede soportar cualquier cálculo de factorial; supongamos el factorial de 25.000, sería 25000*24999*24998*24997*24996*24995…, lo que evidentemente sobrepasaría la capacidad de almacenamiento de un "float". En consecuencia, un análisis superfluo de este punto, puede provocar desbordamiento de la capacidad de almacenamiento, y un consiguiente manejo de basura en el programa, lo que repercute directamente en el mal funcionamiento del mismo.
- **Almacenamiento en Pila:** No abuse del uso de la recursividad, si puede evítela, y si la debe usar, asegúrese de que su problema recursivo está bien definido y delimitado. Recuerde de que por cada invocación recursiva, se almacena información en la pila, por lo que un uso indebido de este punto, provocaría un desbordamiento inminente.

Análisis de la Condición de Corte Recursivo Mal Definida

Como se mencionó, la condición de corte mal definida, produce una invocación infinita de funciones o bien un resultado erróneo de la función, lo que provoca un desbordamiento de pila. Veamos un ejemplo de esta situación:

Archivo: **recubad1.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       float factorial(float);
5
6       void main(void)
7       {
8        float nro=0;
9
10       clrscr();
11       cout<<"Ingrese el N£mero a Calcular su factorial: ";
12       cin>>nro;
13       cout<<"El factorial de "<<nro<<" es "<<factorial(nro);
14      }
15
16      float factorial(float num)
17      {
18       return (num<0)?1:(num*factorial(num-1));
19      }
```

El programa "recubad1.cpp" no ingresa en un ciclo infinito de invocación a funciones, pero retorna un valor incorrecto de la función factorial.

n!	Valor Actual de n	¿n < 0?	Calcular el Valor de n!	Valor Acumulado
	5	no	5*(5-1)!	5*
	4	no	4*(4-1)!	5*4*
	3	no	3*(3-1)!	5*4*3*
5	2	no	2*(2-1)!	5*4*3*2
	1	no	2*(1-1)!	5*4*3*2*1
	0	no	0*(0-1)!	5*4*3*2*1*1*0
	-1	si	1	5*4*3*2*1*1*0*1=**0**

Tabla 13: "Análisis de la Condición de Corte Recursivo Mal Definida"

Al establecer la condición de corte en "n<0", 0 se convierte en parte de la productoria, por lo que eminenmente se producirá para cualquier número, capaz de almacenarse en un flotante, un retorno de función igual a cero. Note que la condición de corte da verdadera en una situación, está bien ubicada y sintácticamente no existen inconvenientes. Es un claro problema de mala definición semántica de la recursividad.

Veamos ahora una situación, en que la condición de corte se encuentre mal definida y nunca diese "verdadero":

Archivo: **recubad2.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       float factorial(float);
5
6       void main(void)
7       {
8        float nro=0;
9
10       clrscr();
11       cout<<"Ingrese el N£mero a Calcular su factorial: ";
12       cin>>nro;
13       cout<<"El factorial de "<<nro<<" es "<<factorial(nro);
14      }
15
16      float factorial(float num)
17      {
18       return (num<num)?1:(num*factorial(num-1));
19      }
```

Es claro que "num" jamás será menor a sí mismo, por lo que esta condición provoca un desbordamiento de pila inminente. Parecen errores simples, pero al momento de programar la recursión se comenten generalmente errores simples que son complicados de localizar. Veamos cómo se comportaría el programa anterior:

n!	Valor Actual de n	¿n < n?	Calcular el Valor de n!	Valor Acumulado
	5	no	5*(5-1)!	5*
	4	no	4*(4-1)!	5*4*
	3	no	3*(3-1)!	5*4*3*
	2	no	2*(2-1)!	5*4*3*2
5	1	no	2*(1-1)!	5*4*3*2*1
	0	no	0*(0-1)!	5*4*3*2*1*1*0
	-1	no	-1*(-1-1)!	5*4*3*2*1*1*0*-1
	-2	no	-2*(-2-1)!	5*4*3*2*1*1*0*-1*-2
	-3	no	-3*(-3-1)!	...

Tabla 14: "Análisis de la Condición de Corte Recursivo Mal Definida"

De acuerdo a lo mencionado en el párrafo, si se observa la tabla 14, jamás se verificará la condición de corte, provocándose una invocación infinita de funciones (Representada con "…" en la columna "Valor Acumulado").

Análisis de la Condición de Corte Bien Definida pero Mal Ubicada

Es frecuente, que analicemos la condición de corte y pensemos: "Sí, en algún momento nos debería dar 'Verdadero', por ejemplo el factorial, si la condición de corte es n=0, en algún momento debería cumplirse", pero ¿qué ocurre si la mal ubicamos en el código?:

Archivo: **recubad3.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       float factorial(float);
5
6       void main(void)
7       {
8        float nro=0;
9
10       clrscr();
11       cout<<"Ingrese el N£mero a Calcular su factorial: ";
12       cin>>nro;
13       cout<<"El factorial de "<<nro<<" es "<<factorial(nro);
14      }
15
16      float factorial(float num)
17      {
18       return num*factorial(num-1);
19
20       if(num==0) return 1;
21      }
```

Note que la condición de corte está bien definida "Si num es igual a cero, retorne 1", pero se encuentra mal ubicada. Como la línea 18 tiene una sentencia cancelativa, nunca se evalúa la condición de la línea 20, y se produce una invocación infinita de funciones. ¿Si quitamos el "return", y mantenemos el resto de la línea 18?, también se

produce una invocación infinita de funciones, ya que la invocación de la función factorial, ocurre antes de la evaluación de la condición de corte, por lo que en realidad nunca se evalúa la condición de corte. Veamos como trabaja el programa antes esta situación:

n!	Valor Actual de n	¿n < 0?	Calcular el Valor de n!	Valor Acumulado
	5		5*(5-1)!	5*
	4		4*(4-1)!	5*4*
	3		3*(3-1)!	5*4*3*
	2		2*(2-1)!	5*4*3*2
5	1		2*(1-1)!	5*4*3*2*1
	0		0*(0-1)!	5*4*3*2*1*1*0
	-1		-1*(-1-1)!	5*4*3*2*1*1*0*-1
	-2		-2*(-2-1)!	5*4*3*2*1*1*0*-1*-2
	-3		-3*(-3-1)!	...

Tabla 15: "Análisis de la Condición de Corte Recursivo Mal Ubicada"

Como puede verse, nunca se evalúa la condición de corte, por lo que se llamará en infinitas oportunidades a la función "factorial", provocando el desborde de pila. Esta situación es similar al caso de que no exista la condición de corte, ya que si no existiese no hay qué evaluar, pero si existe y nunca se evalúa, tampoco hay qué evaluar de acuerdo a la secuencia del programa. La pregunta que naturalmente puede surgirle ¿Cómo soluciono esta situación?, la respuesta es simple pero no siempre fácil de localizar, invierta la línea 18 con la 20 y asunto terminado.

Eliminación de la Recursión

Como se mencionó anteriormente, siempre que pueda eliminar la recursión hágalo, pero en muchas situaciones es "Un Mal Necesario" debido a la naturaleza de los problemas. Veamos como se puede eliminar la recursividad del programa "recurfac.cpp":

Archivo: **recusin.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       float factorial(float);
5
6       void main(void)
7       {
8        float nro=0;
9
10       clrscr();
11       cout<<"Ingrese el N£mero a Calcular su factorial: ";
12       cin>>nro;
13       cout<<"El factorial de "<<nro<<" es "<<factorial(nro);
14      }
15
16      float factorial(float num)
17      {
18       float acu=1;
19
```

```
20      for(;((int)num)>0;num--)
21           acu*=num;
22
23      return acu;
24     }
```

En el programa "recusin.cpp", note que la función factorial, rescrita en modo no recursivo, implica mayor espacio en código pero elimina el inconveniente de lidiar con la pila del programa. Para eliminar la recursividad de la función factorial, se utilizó un acumulador flotante inicializado en 1 (Debido a que es una productoria, si lo hubiésemos iniciado en 0, el producto y por ende el factorial, hubiese dado siempre cero), luego se empleo un ciclo controlado por la misma variable "num", cuya condición de corte, era que la parte entera de dicha variable sea siempre superior a cero, si no lo era, indicaba el fin de ciclo y por ende el fin del cálculo del factorial. Finalmente, se retorna el valor de factorial almacenado en "acu".

Veamos como trabaja esta rutina "factorial" del programa "recusin.cpp", ante la solicitud del cálculo de factorial(5):

n!	Valor Actual de num	¿n > 0?	Valor acu	Acu=Acu*num	num--
5	5	si	1	Acu=1*5=5	(5--)→4
	4	si	5	Acu=5*4=20	(4--)→3
	3	si	20	Acu=20*3=60	(3--)→2
	2	si	60	Acu=60*2=120	(2--)→1
	1	si	120	Acu=120*1=120	(1--)→0
	0	no	**120**		

Tabla 16: "Análisis del Mecanismo del Algoritmo Factorial No Recursivo"

Este ejemplo tiene por fin, demostrar que en determinadas situaciones, con un análisis minucioso de la problemática, es posible evitar la recursividad.

Búsqueda Binaria Recursiva

Un ejemplo de utilización de la recursividad, es en la Búsqueda Binaria. La Búsqueda Binaria es un Algoritmo de Búsqueda que parte de la premisa de que el lugar donde se va a buscar un determinado valor, cumple con la condición de que se encuentra ordenado.

Veamos un ejemplo simple de la misma:

Archivo: **recubbin.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       short int bbin(int[], int buscado, int low, int high);
5
6       void main(void)
7       {
8        int arreglo[]={1,3,4,6,8,9,10,13,16,18,19,20,24,26,27};
9
10       clrscr();
11       int buscado=8;
```

```
12      cout<<"El Número "<<buscado<<""<<((bbin(arreglo,buscado,0,14)==1)?"Existe":"No
        Existe");
13     }
14
15     short int bbin(int a[],int buscado, int low, int high)
16     {
17      int mitad=0;
18
19      if(low>high)
20       {
21        return -1;
22       }
23
24      mitad=(low+high)/2;
25
26      return (buscado==a[mitad]?1:(buscado<a[mitad])?bbin(a,buscado,low,mitad-1):
        bbin(a,buscado,mitad+1,high));
27     }
```

Analicemos el comportamiento del Algoritmo de Búsqueda Binaria Recursiva, para comprender el trabajo y la importancia de la recursividad en el mismo:

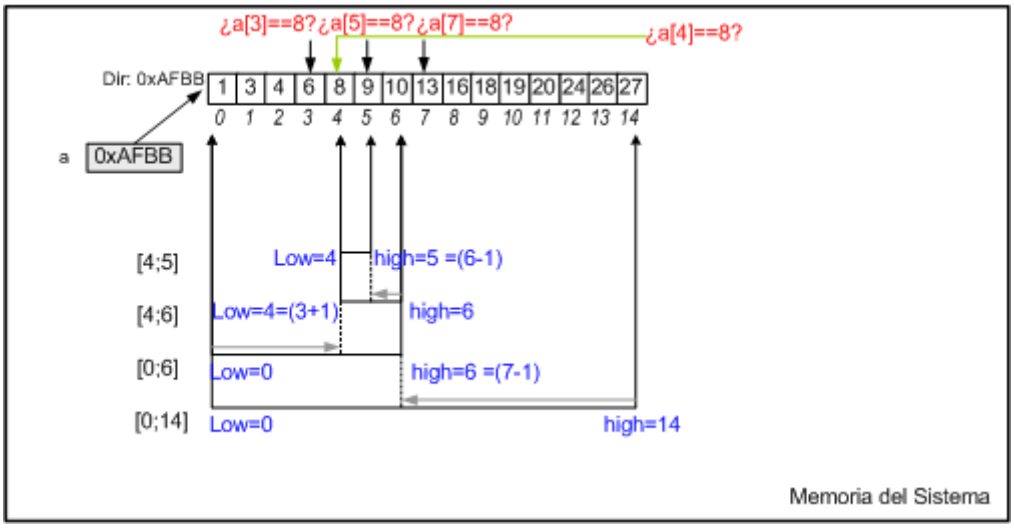


Figura 46: "Análisis del Comportamiento Recursivo de la Búsqueda Binaria"

Supongamos, al igual que lo hicimos en el programa "recubbin.cpp", que el valor que deseamos buscar en nuestro arreglo "a" es el 8. Al invocar a la búsqueda binaria se indica bbin(a,8,0,14), donde:

- "a" es el arreglo donde se buscará el valor,
- 8 es el valor buscado,
- 0 es el límite inferior del arreglo,
- 14 es el límite superior del arreglo.

La Búsqueda Binaria pregunta, ¿En la mitad del intervalo entero está el valor buscado?, esta pregunta la formula como sigue:

- medio=(0+14)/2=7,
- ¿a[7]==8? No

Al ser distinto, la búsqueda binaria se reinvoca definiendo un nuevo intervalo dado por la siguiente pregunta ¿8<a[7]? (donde a[7] posee un 13), al ser positiva la respuesta el nuevo intervalo queda definido por [0;7-1], o sea [0;6]. ¿Porqué define de esta forma el intervalo? Partiendo de la condición de que el arreglo está ordenado, si la pregunta dio falsa, el algoritmo sabe que de la posición actual (7) hasta el final del arreglo, son números mayores al buscado, por lo que no nos interesa y descarto la mitad del arreglo. Esto es interesante desde el punto de vista que se acota a la mitad el espacio de búsqueda.

Definido el nuevo intervalo [0;6], se reitera la pregunta ¿En la mitad del intervalo entero está el valor buscado?:

- medio=(0+6)/2=3,
- ¿a[3]==8? No

Al ser distinto nuevamente, ya que a[3] posee un 6, se pregunta: ¿8<a[3]? No. Como en este caso, el número buscado es superior a la mitad del intervalo, se sabe que desde dicha mitad hasta el inicio del arreglo serán números inferiores al buscado, por lo que no se encuentra allí el 8. De esta forma, se invoca nuevamente a la función bbin, definiendo el nuevo intervalo como [3+1,6], en este caso el truncamiento del intervalo se produjo desde el lado izquierdo del mismo.

Con el intervalo [4;6] definido, se procede a preguntar nuevamente ¿En la mitad del intervalo entero está el valor buscado?:

- medio=(4+6)/2=5,
- ¿a[5]==8? No

Nuevamente la respuesta a la pregunta es no, ya que a[5] es 9. ¿8<a[5]? Sí, entonces el acotamiento del intervalo será por el lado derecho del mismo, ya que los valores que contiene son superiores al buscado. Así la invocación de la función bbin, define el nuevo intercalo como [4;6-1], o sea [4;5].

Con el intervalo [4;5] definido, se procede a preguntar ¿En la mitad del intervalo entero está el valor buscado?:

- medio=(4+5)/2=4,5=4 (Como la variable 'medio' es entera, solo almacena el valor entero),
- ¿a[4]==8? Sí, valor encontrado, retorna.

Al encontrar el valor en la posición 4 del arreglo 'a', inicia la secuencia de retornos, indicando que el valor fue encontrado.

Aclaración: Queda como tarea de análisis para el lector, describir el comportamiento recursivo del algoritmo de búsqueda binaria, para el caso en que se desee buscar el valor "7".

Exponenciación Recursiva

Veamos un ejemplo de implementación de la función Exponenciación, con base y exponente positivo, implementada recursivamente:

Archivo: **recuexpo.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <math.h>
4
5       float expo(float,float);
6
7       void main(void)
8       {
9        int base,exponente;
10       base=exponente=0;
11       clrscr();
12
13       cout<<"Ingrese la base: ";
14       cin>>base;
15       cout<<"Ingrese el Exponente: ";
16       cin>>exponente;
17
18       base=abs(base);
19       exponente=abs(exponente);
20
21       cout<<"\n\n"<<base<<" a la "<<exponente<<" es: "<<expo(base,exponente);
22      }
23
24      float expo(float base,float exponente)
25      {
26       return (exponente==0)?1:base*expo(base,--exponente);
27      }
```

Exponenciación No Recursiva

Veamos ahora, la forma de eliminar la recursividad de la función “expo”:

Archivo: **recuexp2.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <math.h>
4
5       float expo(float,float);
6
7       void main(void)
8       {
9        int base,exponente;
```

```
10      base=exponente=0;
11      clrscr();
12
13      cout<<"Ingrese la base: ";
14      cin>>base;
15      cout<<"Ingrese el Exponente: ";
16      cin>>exponente;
17
18      base=abs(base);
19      exponente=abs(exponente);
20
21      cout<<"\n\n"<<base<<" a la "<<exponente<<" es: "<<expo(base,exponente);
22     }
23
24     float expo(float base,float exponente)
25     {
26      float acu=1;
27
28      for(;exponente!=0;exponente--)
29      {
30       acu*=base;
31      }
32      return acu;
33     }
```

Mostrar una Cadena de Caracteres Mediante Aritmética de Punteros y Algoritmos Recursivos

Veamos como implementar un algoritmo, que nos permita mostrar una cadena de caracteres cualesquiera, mediante aritmética de punteros y algoritmo recursivo:

Archivo: **recucad.cpp**

```
1      #include <iostream.h>
2      #include <conio.h>
3      #include <string.h>
4
5      void veo(char*);
6
7      void main(void)
8      {
9       char cad[]="Mi cadena recursiva\n";
10
11      veo(cad);
12     }
13
14     void veo(char* cad)
15     {
16      if(cad==NULL) return;
```

```
17
18      if(*cad=='\n')
19       {
20        cout<<"\n\nFin de la Cadena";
21        return;
22       }
23
24      cout<<(char)*cad;
25
26      veo(++cad);
27     }
```

Referencias

- "El Lenguaje de Programación C"; Kernighan y Ritchie; Editorial Prentice Hall; México, 1991. Sección 4.10, Capítulo 4: "Recursividad".

- "Estructuras de Datos con C y C++"; Langsam, Aungenstein y Tenenbaum; Editorial Prentice Hall; México, 1997. Capítulo 3: "Recursión".

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995. Capítulo 5: "Recursión".

- "Some Techniques for Recursión renoval from Recursive Functions"; Arsac y Kodratoff; ACM Transactions on Programming Languages and Systems (TOPLAS); Volumen 4, Edición 2, Abril de 1982. Páginas 295-322.

- "Recursion As an Effective Step in Program Development"; L. Colussi; ACM Transactions on Programming Languages and Systems (TOPLAS);Volumen 6, Edición 1, Enero de 1984. Páginas 55-67.

- "Recursion in gradual steps (is a Recursion really that difficult)"; J. Ángel Velásquez-Iturbide; ACM SIGCSE Bulletin, Procedente del trigésimo primer simposio técnico sobre Educación de Ciencias de la Computación. Marzo de 2000.

- "A Framework for Teaching Recursion"; Gary Ford; ACM SIGCSE Bulletin, Volumen 14, Edición 2, Junio de 1982.

3 Análisis de Algoritmos

Marco de Referencia del Análisis de Algoritmos

En la mayoría de las situaciones, al momento de dar una alternativa de solución a un requerimiento de sistema informático, existirán distintos algoritmos que serán de utilidad. Con el avance de la tecnología, en muchas situaciones se abusa de la potencia de los microprocesadores, no efectuando una correcta selección de algoritmos.

Llegado el caso en que el programador decida ordenar una lista, las alternativas, desde el punto de vista algorítmico, son bastas y sobradas; pero no todas ellas se adecúan de la misma forma a la situación planteada. ¿Cómo saber cuál Algoritmo emplear para cada situación?, más aún, ¿Cómo podemos efectuar una comparación de Algoritmos? Allí entra en juego el Análisis de Algoritmos.

El Análisis de Algotimos trata de estudiar el comportamiento de un algoritmo, independientemente de las instrucciones que lo codifiquen o sobre qué ordenador se ejecute, y de esta forma, clasificarlo dentro de determinadas categorías que permitan homologar las características de los mismos.

El **"Primer Paso"** en el Análisis de un Algoritmo, consiste en identificar la naturaleza de su entrada y decidir qué tipo de análisis se efectuará sobre el mismo. Idealmente, desearíamos identificar la distribución de tiempos que demora un algoritmo en cumplir su objetivo, teniendo en cuenta las distintas entradas posibles al mismo. Esto último, en la práctica, se torna sumamente complejo para aquellos algoritmos que no sean triviales.

Por lo regular se tiende a encontrar un *límite superior* para el algoritmo, independientemente de cual fuese la entrada, e intentando obtener el tiempo de ejecución *medio* para su entrada *aleatoria*.

El **"Segundo Paso"** del análisis, consiste en identificar las *operaciones abstractas* en las que se basa un Algoritmo. Esto último, permite aislar el estudio del algoritmo de la plataforma donde se ejecute. En otras palabras, permite interpretar la siguiente sentencia:

$$if\ (a[j] < c)\ \ldots$$

Como una operación abstracta en términos del análisis de algoritmos, en lugar de mencionar que tarda 3 nano segundos en ejecutarse en un micrprocesador Pentium III. De esta forma, podremos analizar los algoritmos en términos de cantidad de comparaciones o intercambio de información, en lugar de atar el análisis a la arquitectura del ordenador.

El **"Tercer Paso"** del análisis, consiste en analizar el algoritmo desde un punto de vista matemático, con el fin de encontrar los valores del caso medio y del peor caso para cada una de las cantidades fundamentales.

La idea será desarrollar, para un algoritmo dado, los pasos enunciados anteriormente y, de esta forma, clasificarlo en una de las categorías enunciadas en el apartado siguiente.

Clasificación de los Algoritmos

La mayoría de los algoritmos tienen un parámetro "N", que representa la cantidad de datos a procesar, la magnitud de éste, afectará en forma directa al tiempo de ejecución. Dentro de las clasificaciones comúnmente conocidas encontramos:

- **"1 o K":** Significa que el Algoritmo, independientemente de la entrada, se ejecuta en una cantidad de pasos constantes. Ej: Si todas las instrucciones de un programa tuviesen esta propiedad, entonces el tiempo de ejecución del programa sería constante.
- **"log(N)":** Implica que el tiempo de ejecución de un programa, irá aumentando en forma logarítmica, a medida que se incremente N. Recordemos que N, puede ser interpretado com el tamaño de la entrada, por lo que a medida que un algoritmo "X" reciba cada vez mayor magnitud en la entrada, mayor serásu tiempo de ejecución, y éste último variará en forma logarítmica.
- **"N":** Implica que el tiempo de ejecución del algoritmo variará linealmente con la magnitud de la entrada. Ej: Si un algoritmo "X" tarda 5 segundos para ordenar un arreglo de 1000 elementos, entonces para ordenar un arreglo con 2000 elementos tardará 10 segundos.
- **"N*log(N)":** Esta categoría corresponde a aquellos algoritmos cuyo tiempo de ejecución aumenta al incrementar N, pero el incremento del tiempo de ejecución, estará dado por la función "N*log(N)".
- **"N^2":** Cuando el tiempo de un algoritmo es cuadrático, solo es útil en pequeños problemas. ¿La Razón? Imagine que su entrada es un arreglo con 1000 elementos, el tiempo insumido por un algoritmo lineal para ordenar el arreglo es de 5 segundos, entonce el cuadrático tardará 25 segundos.
- **"N^3":** Un algoritmo que procesa tríos de elementos de datos (ejemplo: un buvle anidado triple) tiene un tiempo de ejecución cúbico y no es útil más que en problemas pequeños.
- **"2^n":** Estos algoritmos son rara vez utilizados, pero corresponden a esta categoría los algoritmos de fuerza bruta, aquellos que forzosamente efectúan todas las comparaciones u operaciones posibles para solucionar un problema.

K	$\log_{10}(N)$	N	$N*\log_{10}(N)$	N^2	N^3	2^n
k	1	10	10	100	1.000	1.024
k	2	100	200	10.000	1.000.000	*más grande aún*
k	3	1.000	3.000	1.000.000	1.000.000.000	
k	4	10.000	40.000	100.000.000	1.000.000.000.000	
k	5	100.000	500.000	10.000.000.000	1.000.000.000.000.000	

Tabla 17: "Comparación de las Categorías de Algoritmos"

Complejidad del Cálculo

Una técnica de aproximación al estudio del rendimiento de los algoritmos consiste en examinar el peor caso, con el fin de determinar la dependencia funcional del tiempo de ejecución del número de datos de entrada.

En otras palabras, dada:

- **"N":** Cantidad de elementos de datos que ingresan al algoritmo,
- **"t":** Tiempo que demora el algoritmo en cumplir su objetivo para la cantidad "N" ingresada.

Encontrar la función "f", tal que:

$$t = f(N)$$

Una vez encontrada la relación "f", podremos determinar si es proporcional a alguna de las categorías enunciadas en el apartado anterior "Clasificaciónde los Algoritmos".

El hecho de poder definir, la proporcionalidad entre dos funciones que representan el comportamiento temporal de dos algoritmos respectivamente, se da a través de un artificio matemático denominado ***notación O***, y se define como sigue:

Notación. Se dice que una función ***g(N)*** pertenece a ***O(f(N))***, si existen las constantes c_o y N_o tales que ***g(N)*** es menor que $c_o x f(N)$ *para todo* $N > N_o$.

Matemáticamente se escribe:

$$g(N) \in O(f(N)) \leftrightarrow \exists c_0 \wedge N_0 \;/\; g(N) < c_0 * f(N) \quad \forall N > N_0$$

La función "g" que está en función de "N" (la cantidad de datos procesados por el algoritmo), tiene como imagen los tiempos asociados a cada uno de los N procesados. De esta forma, mediante la variación del tamaño de la entrada ante un algoritmo dado, se pueden obtener los distintos tiempos de ejecución asociado, y de esta forma saber si el algoritmo pertenece a una O(f(N)), como ser N^2, o $\log_{10}(N)$, entre otras.

Ejemplo, suponga que prueba un algoritmo "Y", con las siguientes entradas: 10, 100, 1000 y los tiempos devueltos son los siguietes:

N	Tiempo (t)
10	5 segundos
100	50 segundos
1.000	500 segundos

Tabla 18: "Complejidad del Cálculo "

Evidentemente, la función "g" que vincula **N** con **t**, es lineal, por ende el algoritmo se aproxima esta en **O(N)**.

Note que se menciona "aproxima", ya que en este caso estamos tomando tiempos de ordenador, ¿Es posible efectuar el mismo análisis, pero en lugar de tomar el tiempo se tome el número de comparaciones para aislarnos de la arquitectura? Claro que sí, de hecho en lugar de figurar la columna "Tiempo", estaría "Número de Comparaciones", por lo que podríamos clasificar al algoritmo basándonos en la relación entre **N** y el número de comparaciones. Así, la tabla 17 quedaría:

N	Nº Comparaciones
10	20
100	200
1.000	2000

Tabla 19: "Complejidad del Cálculo con Nº de Comparaciones"

De esta forma, la relación "g" entre la entrada N y el número de comparaciones, para un algoritmo "Y", es nuevamente lineal. Este aspecto es al que se refería en apartados anteriores con emplear operaciones abstractas en lugar de tiempo de ordenador.

Estos casos que se presentan de relación entre N y su tiempo (ó Nº de comparaciones) asociado, son relativamente sencillos. En la realidad suelen, presentarse valores asociados a cada entrada N, que hace dificultoso aproximar una función que relacione a N con t (ó el número de comparaciones). En estos casos, se habla de aproximación de la relación "g", por ejemplo a través de la técnica de mínimos cuadrados, lo que implica una consecuente aproximación al orden del algoritmo. Esto suele ocurrir en algoritmos complejos, en los que se hace sumamente engorroso el análisis por operaciones abstractas.

Análisis del Caso Medio

Otra forma de estudiar el rendimiento de un algoritmo consiste en examinar el caso medio. Partimos de caracterizar con precisión la entrada al algoritmo, esto implica que definimos por ejemplo, como entrada de un algoritmo de ordenamiento, un arreglo de tamaño N.

Entonces, se calcula el número de veces que se ejecuta cada instrucción (frecuencia). Con ello, multiplicando la frecuencia de cada instrucción por el tiempo que se necesita para ejecutarla, y sumando así el tiempo medio de todas las instrucciones, se obtiene el tiempo medio total del programa. Si consideramos el tiempo medio del algoritmo, teniendo presente su estimación, a partir de elementos de datos diferentes por cada ejecución, podremos aproximar el *O(N)* del mismo.

Matemáticamente sería, para una instrucción "i":

$$ACM = \frac{\left(\sum_{j=1}^{q_p} \sum_{i=1}^{q_i} (f_i * t_i)\right)}{q_p}$$

donde:

f_i: Frecuencia promedio de ejecución de la instrucción "i",
t_i: Tiempo que insume ejecutar la instrucción "i",
q_i: Cantidad de instrucciones en el algoritmo

q_p: Cantidad de Veces que se corre el Algoritmo (con distintas entradas)
ACM: Análisis del Caso Medio

Este tipo de análisis presenta tres **dificultades** claras, a saber:

1. En algunos ordenadores puede resultar difícil determinar con precisión el tiempo que se necesita para cada instrucción. Además, se está atando al Análisis del Algoritmo a la Arquitectura.
2. A menudo, su este tipo de análisis suele complicarse desde el punto de vista matemático, lo que en circunstancias lo torna en una alternativa no muy feliz.
3. El modelo de datos empleado para analizar el algoritmo, puede no ser representativo de casos en la vida real.

Resultados Aproximados y Asintóticos

Al tratar de estimar el orden de un algoritmo (*O(f(N))*), se tiende a aproximar al mismo. Fuese por el método que se escoja, obtendremos resultados aproximados, y en muchos casos ignoraremos el valor de c_o. Ignorar el valor de c_o puede traer aparejado la asignación de un orden incorrecto al algoritmo (ya que c_o podría representar una constante de tiempo elevada que al multiplicar a *f(N) lo afecte en sobremanera*).

Técnicamente no se tiene seguridad real de que se pueden ignorar los números pequeños, como c_o, en la aproximación del orden de un algoritmo. *Estas constantes podrían afectar los tiempos del algoritmo*, pero no alteraría su orden, sino que se convertirían en una asíntota de su valor anterior.

Por ejemplo:

c_o	N	*f(N)*	*f(N)+O(g(N))*
300	10	5	305
300	100	50	350
300	1.000	500	800
300	10.000	5.000	5.300

Suponga que O(g(N)) representa el valor de una constante c_o, así:

$$f(N) + O(g(N)) = f(N) + c_0$$

De esta forma, los tiempos de g(N) son asintóticamente grandes con respecto a f(N):

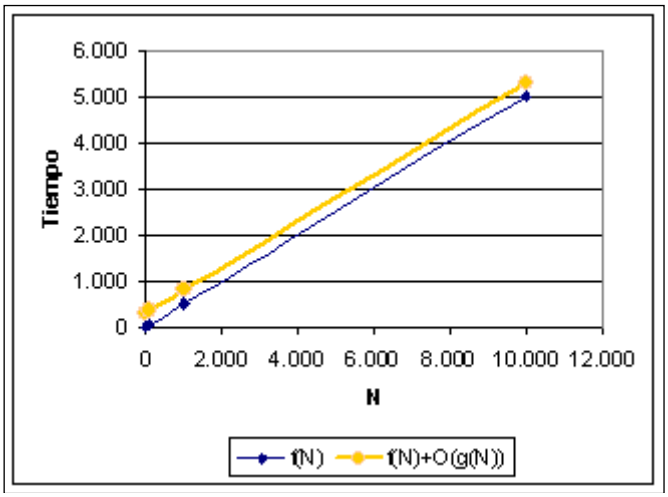


Figura 47: "Relaciones Asíntotas entre Algoritmos, a causa de c_o"

Análisis de Algoritmos Recursivos

Dentro de los algoritmos recursivos, por lo general cada invocación, trata de trabajar sobre un rango o espectro mejor definido que el anterior, por lo que supone que si inicialmente se invocó el algoritmo con datos de longitud N, la segunda invocación estará dada por una longitud N_2 (siendo $N_2 < N$), la tercera invocación estará dada por una longitud N_3 (siendo $N_3 < N_2$), y de esta forma sucesivamente hasta encontrar su punto de retorno.

En este tipo de algoritmos, no solo hay que considerar las comparaciones suscitadas en cada invocación recursiva del algoritmo, sino el tiempo transcurrido en la invocación y el retorno. De esta forma, el orden de un algoritmo recursito (OAR), estará dado por:

$$OAR = O(N) + O(N_2) + O(N_3) + \ldots + Ti + Tr$$

Donde:

Ti: Tiempo Total insumido en la invocación
Tr: Tiempo Total insumido durante el retorno

Casos Prácticos con Análisis

Ordenamiento por Burbuja

El ordenamiento por burbuja, es uno de los algoritmos tradicionales, tanto para iniciarse en la temática de ordenamiento como en la de Análisis de Algoritmos. Veamos a continuación una implementación de dicho algoritmo:

Archivo: **orden1.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4
5       void ordenar(char*);
6       void intercambio(char*,int,int);
7
8       void main(void)
9       {
10       char arreglo[15];
11       strcpy(arreglo,"hola mundo");
12
13       clrscr();
14       cout<<"Cadena Inicial: "<<arreglo;
15       ordenar(arreglo);
16       cout<<"\nCadena Ordenada: "<<arreglo;
17
18      }
19
20      void ordenar(char *ar)
21      {
22       short int tam=strlen(ar)-1;
23       cout<<tam;
24       int i,j;
25
26       for(i=tam;i>=1;i--)
27        for(j=1;j<=i;j++)
28          if(ar[j-1]>ar[j]) intercambio(ar,j-1,j);
29      }
30
31      void intercambio(char* ar, int desde, int hasta)
32      {
33       char tmp=ar[desde];
34       ar[desde]=ar[hasta];
35       ar[hasta]=tmp;
36      }
```

Veamos como trabaja el siguiente algoritmo, a partir de la cadena de entrada "hola mundo":

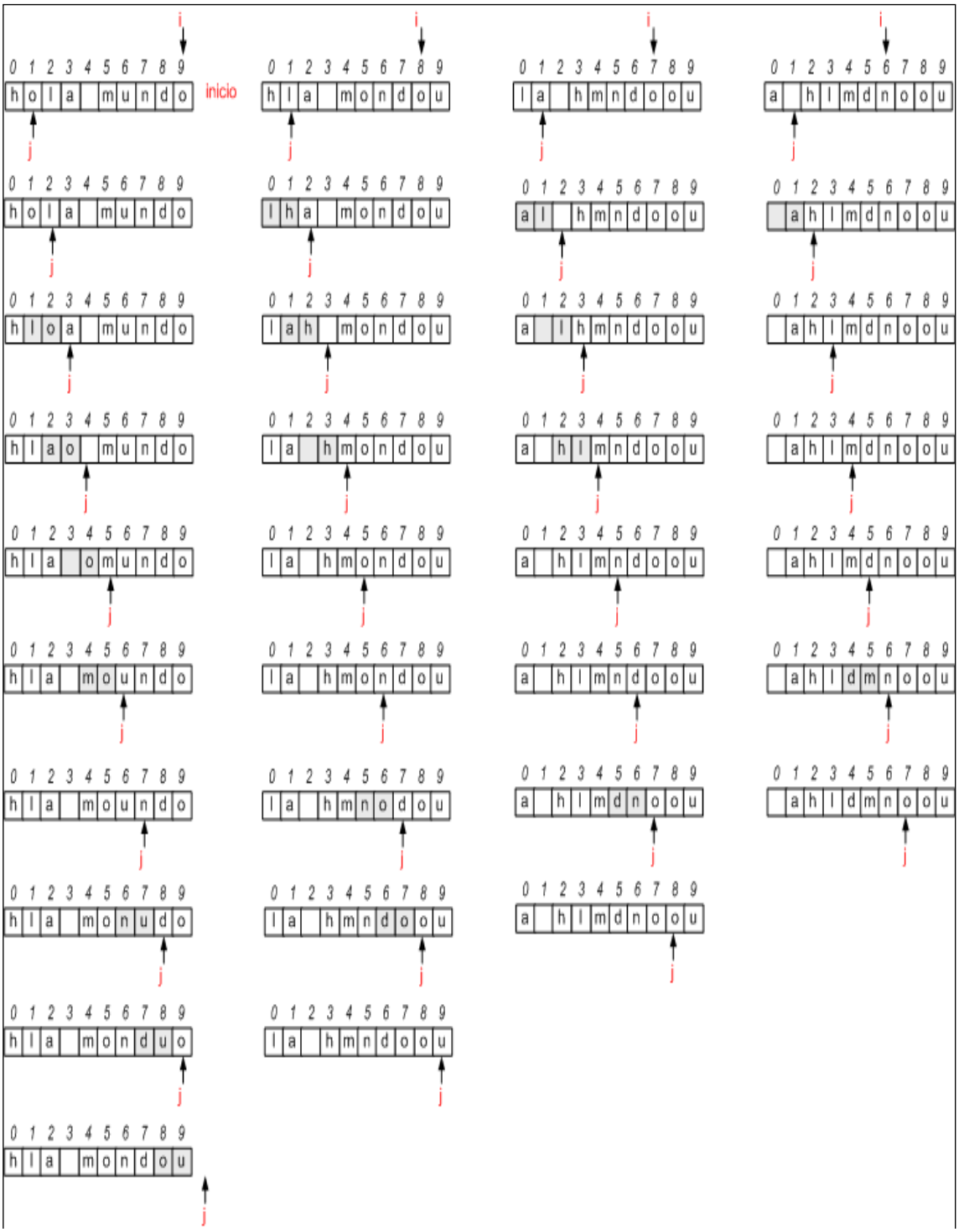


Figura 48: "Análisis del Algoritmo de Ordenamiento basado en Burbujas"

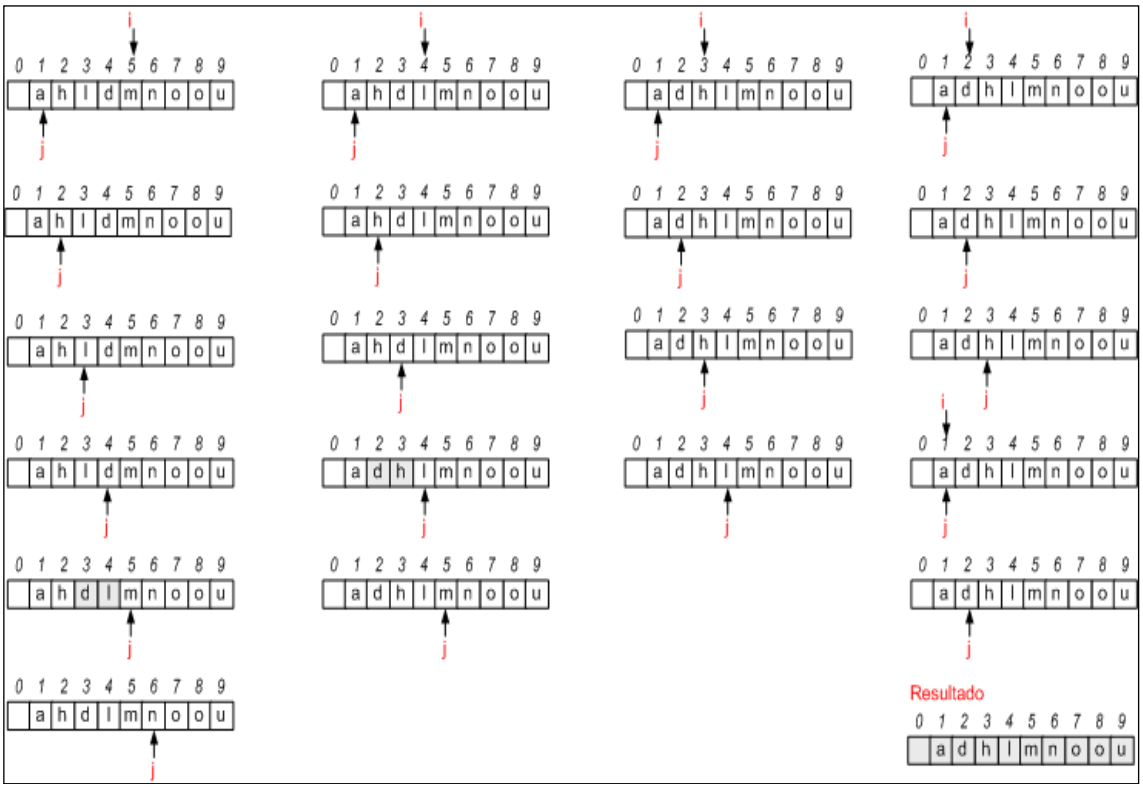


Figura 49: "Análisis del Algoritmo de Ordenamiento basado en Burbujas (Continuación)"

Vea la cantidad de comparaciones que efectua el Algoritmo de Burbujas, tan solo para una cadena como "hola mundo". Imagine el número de comparaciones que llegaría a efectuar el algoritmo para una cadena como "Mi Amnigo Juan es de River". El Algoritmo de Burbujas es considerado del orden $O(N^2)$.

Analicemos la cantidad de comparaciones que se realizan para cada entrada de tamaño N, de cara a fundamentar el orden del algoritmo:

N (Tamaño del Arreglo)	Nº Comparaciones
10	45
20	190
30	435
40	780
50	1225
60	1770
70	2415
80	3160

Tabla 20: "Evolución del Nº de Comparaciones del Algoritmo de Burbujas"

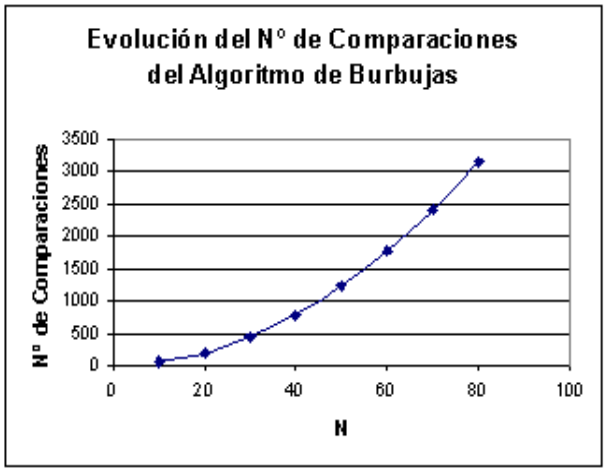


Figura 50: "Análisis Gráfico del Comportamiento del Algoritmo de Burbujas"

De más están las palabras, ante un claro ejemplo de algoritmo de orden N^2 como el que brinda el "Algoritmo de Burbujas".

Ordenamiento por Inserción

El ordenamiento por inserción es otro de los algoritmos tradicionales, empleados para efectuar ordenamientos básicos. Veamos un programa que lo implementa para luego efectuar el Análisis del Algoritmo.

Archivo: orden2.cpp

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4
5       void insercion(char*,int N);
6
7       void main(void)
8       {
9        char arreglo[100];
10       strcpy(arreglo,"hola mundo");
11
12       clrscr();
13       cout<<"Cadena Inicial: "<<arreglo;
14       insercion(arreglo,strlen(arreglo)-1);
15       cout<<"\nCadena Ordenada: "<<arreglo;
16
17      }
18
19      void insercion(char *a,int N)
20      {
21       char v;
22       int cmp=0;//contador de comparaciones
23
24       for(int i=1;i<=N;i++)
25       {
26        v=a[i];
27
28        cmp++;
29        while(a[i-1]>v && i>0)
30         {
31          a[i]=a[i-1];
32          i--;
33          a[i]=v;
34         }
35       }
36
37       cout<<"\nTotal de Comparaciones: "<<cmp;
38      }
```

Veamos como trabaja el siguiente algoritmo, a partir de la cadena de entrada "hola mundo":

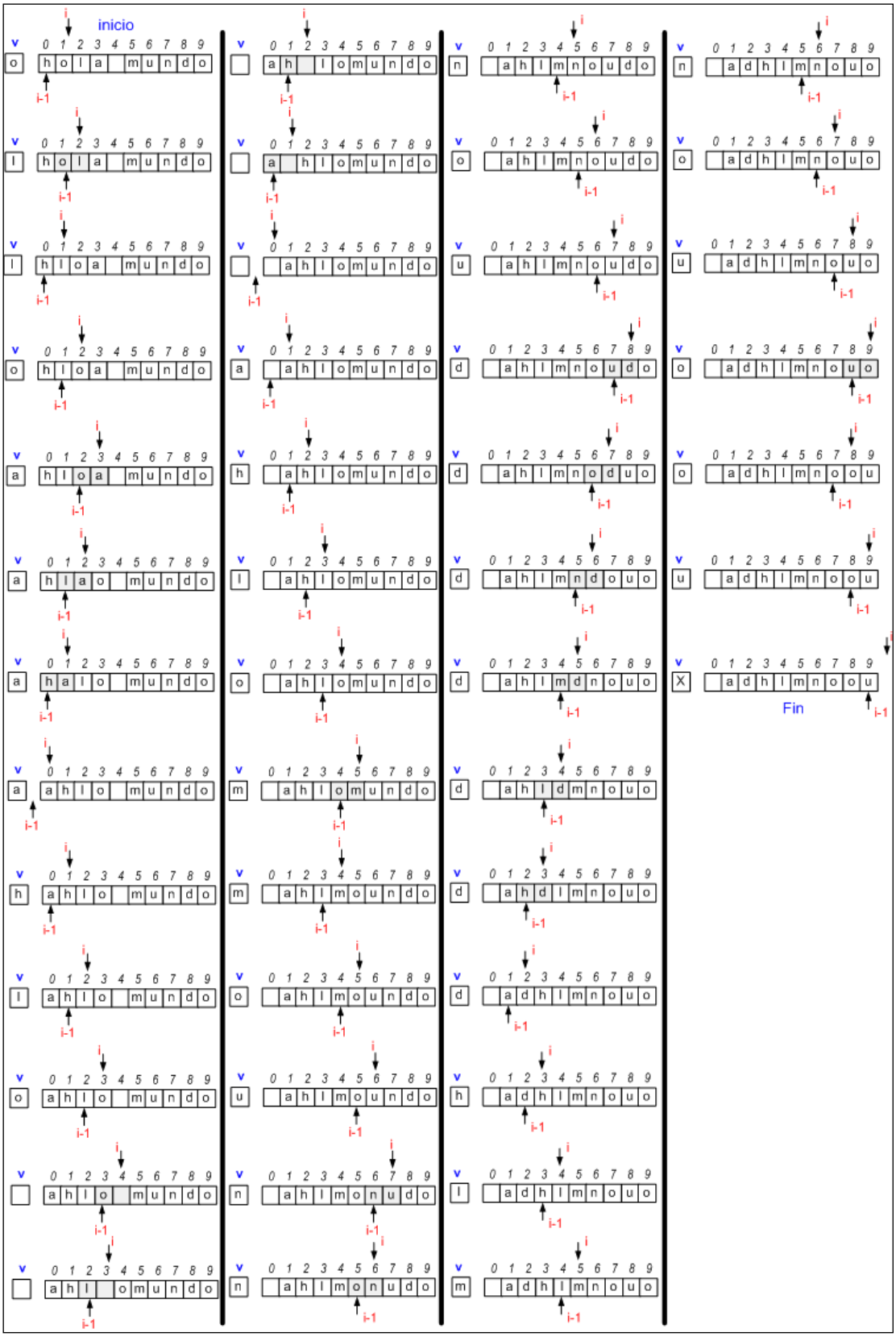


Figura 51: “Análisis Gráfico del Comportamiento del Algoritmo de Ordenamiento por Inserción”

Vea que la cantidad de comparaciones que efectua el Algoritmo de Inserción, es sensiblemente inferior (para la entrada N dada) al Burbujas, teniendo en cuenta la cadena "hola mundo". Veamos que ocurre con su comportamiento cuando N se incrementa:

N (Tamaño del Arreglo)	Nº Comparaciones
10	27
20	99
30	215
40	375
50	579
60	827
70	119
80	1455
90	1835
100	2259

Tabla 21: "Evolución del Nº de Comparaciones del Algoritmo de Ordenamiento por Inserción"

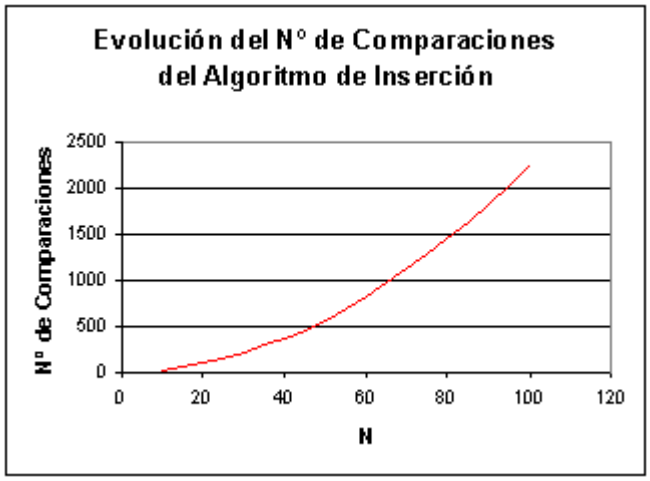


Figura 52: "Análisis Gráfico del Comportamiento del Algoritmo de Ordenamiento por Inserción"

El Algoritmo de Inserción posee un orden N^2, pero ¿será superior ó inferior al Algoritmo de burbujas? Es claramente inferior. Si bien ambos son del $O(N^2)$, recuerde que la frase "*del orden de*" implica un acercamiento y no un valor exacto. Por lo que en este último caso, en que ambos algoritmos son de $O(N^2)$, no implica que tengan el mismo comportamiento sino que desde el punto de vista de tipología de algoritmos, se comportan "**similarmente**".

Veamos la comparación gráfica del comortamiento del Algoritmo de Inserción con el de Burbujas:

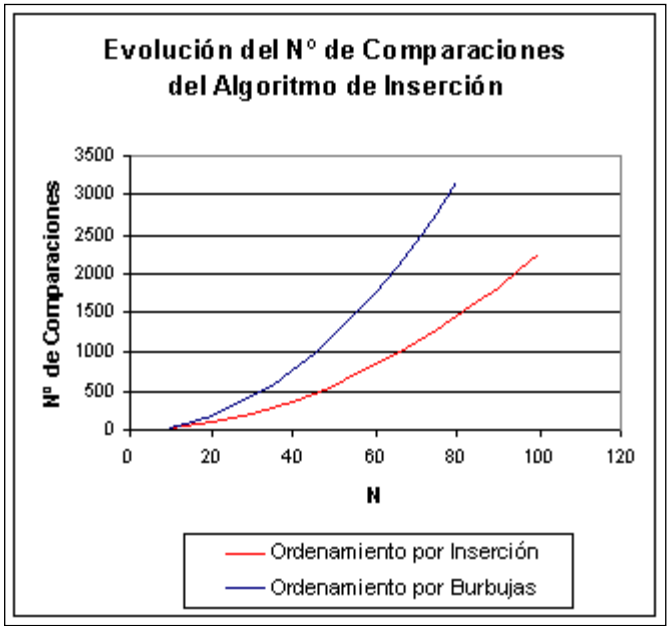


Figura 53: "Análisis Comparativo del Comportamiento del Algoritmo de Inserción, para con el de Burbujas"

Referencias

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995. Capítulo 6: "Análisis de Algoritmos".

- "Análisis de Algoritmos: Complejidad"; José Mañas; Departamento de Ingenieía de Sistemas Telemáticos, Universidad Politécnica de Madrid. España, Noviembre de 1997.

- "Algoritmos y su Complejidad"; Prof. Dr. Eric Jeltsch F; Material Asociado a la Currícula 'Diseño y Análisis de Algoritmos con Java', Ingeniería en Computación, Universidad de la Serena; Chile, Mayo de 2004.

- "The Design and Analysis of Computer Algorithms"; A.V. Aho, J.E. Hopcroft y J.D. Ullman; Addison-Wesley, 1974.

- "Introduction to Algorithms"; T. Cormen, C.Leiserson y R. Rivest; Editorial McGraw-Hill & MIT Press; 1990.

4

Estructuras de Datos Simples

Introducción a las Estructuras de Datos

En cualquier lenguaje de desarrollo normalmente se soportan los denominados "tipos de datos naturales". Estos mismos representan un conjunto de formas de representar distintos valores de un subconjunto de la realidad bien acotado. Para el resto de los datos en los que un lenguaje determinado no brinda soporte, se facilita algún mecanismo de extensibilidad, que en el caso de C/C++ se brinda a través de las estructuras y clases. No conforme con ello, un programador requiere en numerosas oportunidades gestionar sus tipos de datos en forma dinámica, gestionando él mismo la creación y eliminación, como así también la gestión de los mismos mediante una estructura o jerarquía que los relacione entre sí, permitiendo un crecimiento y dinamismo controlado, que permite lograr programas de mayor eficiencia. En este último rubro, por así llamarlo, es donde ingresan los **T**ipos **A**bstractos de **D**atos (TAD).

Dentro de los TADs existen numerosas representaciones de la realidad, como ser listas y sus variantes, colas, pilas, entre otras. El objetivo de este capítulo, es demostrar al lector los mecanismos de implementación de estos tipos abstractos, de cara a que sea capaz de interpretarlos y aplicarlos en distintos aplicativos.

Listas

Concepto

Existen numerosas situaciones de programación que requieren del manejo de una secuencia de tipos de datos, ordenados o no, que crezcan y disminuyan en forma dinámica y controlada, éste tipo de problemas son los característicos al momento de analizar la aplicación de "Listas".

Típicamente una lista se define como:

> "Secuencia de Datos Vinculados y Relacionados, cuyo crecimiento y/o decrecimiento dinámico es plausible y controlado".

Interpretando gráficamente la anterior definición tenemos:

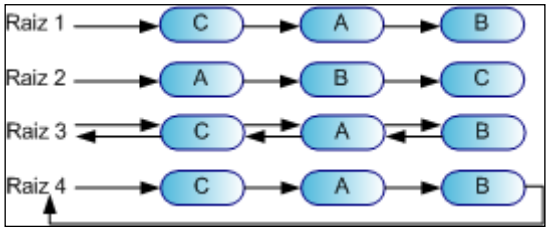


Figura 54: "Análisis Conceptual de las Listas"

Por Raíz se entiende el lugar desde donde parte y nace una lista, independientemente de que se encuentre o no ordenada. En el caso de la lista asociada a la Raíz 1, tenemos una secuencia de nodos (Ej: C, A y B) vinculados entre sí mediante algún mecanismo dinámico que permita, su crecimiento y/ó decrecimiento, lo que se denomina "Lista Simplemente Enlazada".

Si a esta Lista Simplemente Enlazada, le agregamos la restricción de que el contenido de sus nodos (tomando uno de los campos asociados al nodo, por Ejemplo

"nombre"), se encuentre ordenado en cualquier momento, entonces concluimos en la lista asociada a la "Raíz 2", lo que se denomina "Lista Ordenada Simplemente Enlazada".

La Característica particular de las listas asociadas a la Raíz 1 y la Raíz 2, es que son "Unidireccionales". Las Listas Unidireccionales, cuyo último elemento dispone un enlace con el primer elemento de la misma, provocan un vínculo circular, a lo que se denomina "Listas Circulares", y es el ejemplo asociado a la Raíz 4.

Las Listas que se pueden navegar en ambos sentidos, independientemente de si son circulares y/u ordenadas, como la asociada a la "Raíz 3", se denominan "Listas Bidireccionales".

Implementación de Listas Simplemente Enlazadas

El programa a continuación, permite al usuario cargar una cantidad indefinida de nombres (cuya longitud por nombre no supere los veinte caracteres), y los va almacenando en una lista simplemente enlazada. Una vez cargada la cantidad de nombres deseada, se procede a mostrarlos tal y como se almacenaron en la lista para finalmente, liberar el espacio de memoria empleado por la lista. El nodo que compone la lista es una estructura simple que alberga un arreglo de veinte caracteres y un puntero del tipo nodo para vincularse con sus "compañeros" de lista.

Archivo: **lista1.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4       #include <stdio.h>
5
6       //Definici¢n del Nodo de la Lista
7       struct nodo{
8               char nombre[20];
9               nodo* next;
10              };
11
12      void insert(nodo*&,char*);
13      void borrarlista(nodo*&);
14      void mostrarlista(nodo*);
15
16      void main(void)
17      {
18       nodo* raiz=NULL;//Raiz de la Lista
19
20       char nombres[20];
21       short int flag=0;
22
23       clrscr();
24       do{
25          cout<<"\nIngrese un nombre (Fin con Punto \".\"): ";
26          gets(nombres);
27
```

```
28          if(strcmp(nombres,".")==0)
29             {
30              flag=1;
31             }
32            else
33             {
34              insert(raiz,nombres);
35             }
36         }while(flag!=1);
37
38       cout<<"\n\nINICIO DE MUESTRA DE LOS NOMBRES INGRESADOS";
39       mostrarlista(raiz);
40       borrarlista(raiz);
41
42       if(raiz==NULL) cout<<"\nMemoria Liberada";
43      }
44
45      void insert(nodo*& raiz, char* string)
46      {
47       nodo *ptr=new nodo;
48
49       strcpy(ptr->nombre,string);
50       ptr->next=raiz;
51       raiz=ptr;
52      }
53
54      void borrarlista(nodo*& raiz)
55      {
56       nodo* ptr=raiz;
57       nodo* elim=raiz;
58
59       while(ptr!=NULL)
60        {
61         elim=ptr;
62         ptr=ptr->next;
63         delete elim;
64         elim=NULL;
65        }
66
67        raiz=NULL;
68      }
69
70      void mostrarlista(nodo* raiz)
71      {
72       nodo *ptr=raiz;
73
74       while(ptr!=NULL)
75        {
76         cout<<"\nNombre: "<<ptr->nombre;
```

```
77          ptr=ptr->next;
78         }
79      }
```

Supongamos que al presente programa, le cargamos los siguientes nombres en dicho orden:

1. Mario
2. José
3. Teodoro
4. Pedro

La Salida por pantalla del mismo será:

```
INICIO DE MUESTRA DE LOS NOMBRES INGRESADOS
Nombre: Pedro
Nombre: Teodoro
Nombre: josé
Nombre: Mario
Memoria Liberada
```

Figura 55: "Salida por pantalla del programa lista1.cpp"

Note que, de acuerdo al algoritmo de inserción en la lista del programa "lista1.cpp", el ingreso de los nodos en la lista se da como:

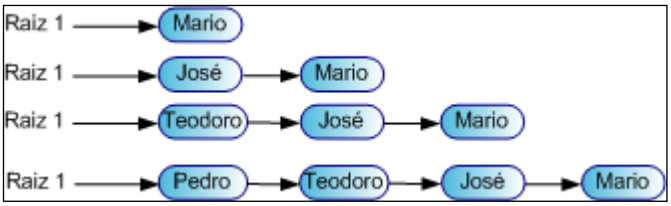


Figura 56: "Análisis de la Modalidad de Inserción en la Lista según lista1.cpp"

Implementación de una Lista Ordenadas Simplemente Enlazada

El programa a continuación permite que un usuario cargue un número de nombres, cuya longitud individual no supere los veinte caracteres, almacenándolos en una lista. La salvedad del programa "lista2.cpp", es que al momento de ingresar el nombre a la lista, lo hace en forma ordenada.

Archivo: **lista2.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4       #include <stdio.h>
5
6       //Definici¢n del Nodo de la Lista
7       struct nodo{
8               char nombre[20];
9               nodo* next;
10              };
```

```
11
12      void insertordenado(nodo*&,char*);
13      void borrarlista(nodo*&);
14      void mostrarlista(nodo*);
15
16      void main(void)
17      {
18       nodo* raiz=NULL;//Raiz de la Lista
19
20       char nombres[20];
21       short int flag=0;
22
23       clrscr();
24       do{
25          cout<<"\nIngrese un nombre (Fin con Punto \".\"): ";
26          gets(nombres);
27
28          if(strcmp(nombres,".")==0)
29             {
30              flag=1;
31             }
32            else
33             {
34              insertordenado(raiz,nombres);
35             }
36         }while(flag!=1);
37
38       cout<<"\n\nINICIO DE MUESTRA DE LOS NOMBRES INGRESADOS";
39       mostrarlista(raiz);
40       borrarlista(raiz);
41
42       if(raiz==NULL)
43        {
44         cout<<"\nMemoria Liberada";
45        }
46      }
47
48      void insertordenado(nodo*& raiz, char* string)
49      {
50       nodo *ptr=new nodo;
51
52       nodo *ptrnow=raiz;//Nodo Actual
53       nodo *ptrant=NULL;//Nodo Anterior
54
55       strcpy(ptr->nombre,string);
56       ptr->next=NULL;
57
58       while(ptrnow!=NULL && strcmp(ptr->nombre,ptrnow->nombre)<0)
59       {//
```

```
60        ptrant=ptrnow;
61        ptrnow=ptrnow->next;
62       }
63
64       if(ptrnow==NULL)
65        {
66         if(ptrant==NULL)
67           {//es el inicio de la lista
68            raiz=ptr;
69           }
70          else
71           {//es el final de la lista
72            ptrant->next=ptr;
73           }
74        }
75       else
76        {
77         if(ptrant==NULL)//es el primer elemento
78          {
79           ptr->next=raiz;
80           raiz=ptr;
81          }
82         else
83          {//es un elemento intermedio
84           ptr->next=ptrant->next;
85           ptrant->next=ptr;
86          }
87        }
88      }
89
90      void borrarlista(nodo*& raiz)
91      {
92       nodo* ptr=raiz;
93       nodo* elim=raiz;
94
95       while(ptr!=NULL)
96        {
97         elim=ptr;
98         ptr=ptr->next;
99         delete elim;
100        elim=NULL;
101       }
102
103       raiz=NULL;
104     }
105
106     void mostrarlista(nodo* raiz)
107     {
108      nodo *ptr=raiz;
109
```

```
110      while(ptr!=NULL)
111       {
112        cout<<"\nNombre: "<<ptr->nombre;
113        ptr=ptr->next;
114       }
115     }
```

Supongamos que un usuario ingresa los siguientes nombres en dicho orden:

1. mario
2. laura
3. sandra
4. jose

El Algoritmo de inserción, según el programa "lista2.cpp", produciría la siguiente salida por pantalla:

```
INICIO DE MUESTRA DE LOS NOMBRES INGRESADOS
Nombre: sandra
Nombre: mario
Nombre: laura
Nombre: jose
Memoria Liberada
```

Figura 57: "Salida por pantalla del programa lista2.cpp"

A través de la salida por pantalla de la figura 57, podemos notar que el presente algoritmo de inserción, ordena en orden decreciente de palabras. Veamos como los ordena paso a paso:

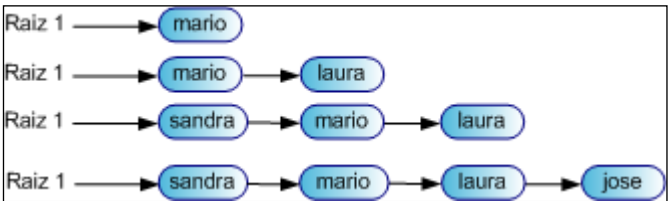


Figura 58: "Análisis de la Modalidad de Inserción en la Lista según lista2.cpp"

Implementación de una Lista Circular

Una alternativa al trabajar con listas, es implementar una lista "Cerrada", o también conocida como lista "Auto Vinculada al Inicio" o "Circular". El siguiente programa permite al usuario cargar un cantidad indefinida de nombres, cuya longitud individual no supere los veinte caracteres, y almacenando los mismos en forma dinámica, en un lista circular.

Archivo: **lista3.cpp**

```
1      #include <iostream.h>
2      #include <conio.h>
```

```
3       #include <string.h>
4       #include <stdio.h>
5
6       //Definici¢n del Nodo de la Lista
7       struct nodo{
8               char nombre[20];
9               nodo* next;
10              };
11
12      void insert(nodo*&,char*);
13      void borrarlista(nodo*&);
14      void mostrarlista(nodo*);
15
16      void main(void)
17      {
18       nodo* raiz=NULL;//Raiz de la Lista
19
20       char nombres[20];
21       short int flag=0;
22
23       clrscr();
24       do{
25          cout<<"\nIngrese un nombre (Fin con Punto \".\"): ";
26          gets(nombres);
27
28          if(strcmp(nombres,".")==0)
29             {
30              flag=1;
31             }
32            else
33             {
34              insert(raiz,nombres);
35             }
36         }while(flag!=1);
37
38       cout<<"\n\nINICIO DE MUESTRA DE LOS NOMBRES INGRESADOS";
39       mostrarlista(raiz);
40       borrarlista(raiz);
41
42       if(raiz==NULL)cout<<"\nMemoria Liberada";
43      }
44
45      void insert(nodo*& raiz, char* string)
46      {
47       nodo *ptr=new nodo;
48
49       strcpy(ptr->nombre,string);
50
51       if(raiz==NULL)//si es el primero
52        {
```

```
53        ptr->next=raiz;
54        raiz=ptr;
55       }
56      else
57       {
58        nodo* ptrnow=raiz;
59
60        if(ptrnow->next==NULL)
61         {
62          ptrnow->next=ptr;
63          ptr->next=ptrnow;
64         }
65        else
66         {//hay mas de un nodo en el anillo
67          while(ptrnow->next!=raiz)
68          {
69           ptrnow=ptrnow->next;
70          }
71
72          //ptrnow->next es raiz
73          ptrnow->next=ptr;
74          ptr->next=raiz;
75         }
76       }
77     }
78
79     void borrarlista(nodo*& raiz)
80     {
81      nodo* ptr=raiz;
82      nodo* elim=raiz;
83      if(raiz==NULL) return;
84      do
85       {
86        elim=ptr;
87        ptr=ptr->next;
88        delete elim;
89        elim=NULL;
90       }while(ptr!=raiz);
91
92      raiz=NULL;
93     }
94
95     void mostrarlista(nodo* raiz)
96     {
97      nodo *ptr=raiz;
98      short int contador=0;
99
100     while(ptr!=NULL && contador<5)
101      {
```

```
102        cout<<"\nNombre: "<<ptr->nombre;
103        ptr=ptr->next;
104
105        if(ptr==raiz) contador++;
106       }
107     }
```

Supongamos que un usuario ingresa los siguientes nombres en dicho orden:

1. mario
2. laura
3. sandra
4. jose

El Algoritmo de inserción, según el programa "lista3.cpp", produciría la siguiente salida por pantalla:

```
INICIO DE MUESTRA DE LOS NOMBRES INGRESADOS
Nombre: mario
Nombre: laura
Nombre: sandra
Nombre: jose
Nombre: mario
Nombre: laura
Nombre: sandra
Nombre: jose
Nombre: mario
Nombre: laura
Nombre: sandra
Nombre: jose
Nombre: mario
Nombre: laura
Nombre: sandra
Nombre: jose
Nombre: mario
Nombre: laura
Nombre: sandra
Nombre: jose
Memoria Liberada
```

Figura 59: "Salida por pantalla del programa lista3.cpp"

Veamos como los inserta paso a paso:

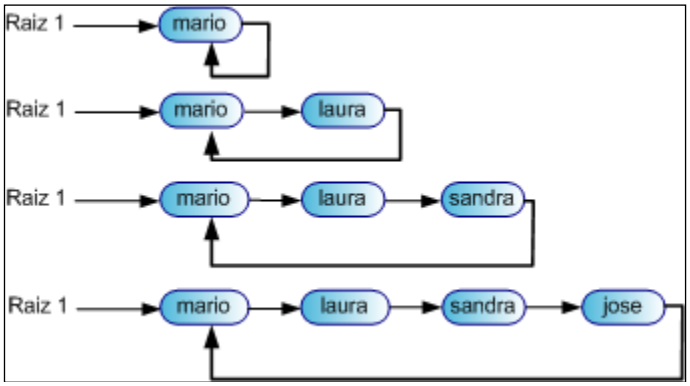


Figura 60: "Análisis de la Modalidad de Inserción en la Lista según lista3.cpp"

Implementación de una Lista Doblemente Enlazada

Dentro de las variedades de las listas dinámicas, tenemos también aquellas con doble vínculo entre sus nodos, que posibilitan una ida y vuelta en forma natural. A estas listas se las conoce como "Listas Doblemente Enlazada". El programa a continuación, permite al usuario ingresar una cantidad indefinida de nombres, cuya longitud individual no supere los veinte caracteres, y almacenando los mismos en una lista doblemente vinculada.

Archivo: **lista4.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4       #include <stdio.h>
5
6       //Definici¢n del Nodo de la Lista
7       struct nodo{
8               char nombre[20];
9               nodo* next;
10              nodo* prev;
11              };
12
13      void insert(nodo*&,char*);
14      void borrarlista(nodo*&);
15      void mostrarlista(nodo*);
16
17      void main(void)
18      {
19       nodo* raiz=NULL;//Raiz de la Lista
20
21       char nombres[20];
22       short int flag=0;
23
24       clrscr();
25       do{
26          cout<<"\nIngrese un nombre (Fin con Punto \".\"): ";
27          gets(nombres);
28
29          if(strcmp(nombres,".")==0)
30             {
31              flag=1;
32             }
33            else
34             {
35              insert(raiz,nombres);
36             }
37         }while(flag!=1);
```

```
38
39      cout<<"\n\nINICIO DE MUESTRA DE LOS NOMBRES INGRESADOS";
40      mostrarlista(raiz);
41      borrarlista(raiz);
42
43      if(raiz==NULL) cout<<"\nMemoria Liberada";
44     }
45
46     void insert(nodo*& raiz, char* string)
47     {
48      nodo *ptr=new nodo;
49
50      strcpy(ptr->nombre,string);
51
52      if(raiz==NULL)
53       {
54        ptr->next=raiz;
55        ptr->prev=NULL;
56        raiz=ptr;
57       }
58      else
59      {
60       if(raiz->next==NULL)
61        {
62         ptr->next=raiz;//vinculo de ida
63         ptr->prev=NULL;
64         raiz->prev=ptr;//vinculo de vuelta
65         raiz=ptr;
66        }
67       else
68        {
69         ptr->next=raiz;
70         raiz->prev=ptr;
71         ptr->prev=NULL;
72         raiz=ptr;
73        }
74      }
75     }
76
77     void borrarlista(nodo*& raiz)
78     {
79      nodo* ptr=raiz;
80      nodo* elim=raiz;
81
82      while(ptr!=NULL)
83       {
84        elim=ptr;
85        ptr=ptr->next;
86        delete elim;
87        elim=NULL;
```

```
88        }
89
90        raiz=NULL;
91      }
92
93      void mostrarlista(nodo* raiz)
94      {
95       nodo *ptr=raiz;
96       nodo *ptrant=NULL;
97
98       while(ptr!=NULL)
99        {
100        cout<<"\nNombre: "<<ptr->nombre;
101        ptrant=ptr;
102        ptr=ptr->next;
103       }
104
105      //Muestro en Orden Inverso
106      cout<<"\n\nMuestra en Orden Inverso";
107
108      while(ptrant!=NULL)
109       {
110        cout<<"\nNombre: "<<ptrant->nombre;
111        ptrant=ptrant->prev;
112       }
113     }
```

Supongamos que un usuario ingresa los siguientes nombres en dicho orden:

1. mario
2. laura
3. sandra
4. jose

El Algoritmo de inserción, según el programa "lista3.cpp", produciría la siguiente salida por pantalla:

```
INICIO DE MUESTRA DE LOS NOMBRES INGRESADOS
Nombre: jose
Nombre: sandra
Nombre: laura
Nombre: mario

Muestra en Orden Inverso
Nombre: mario
Nombre: laura
Nombre: sandra
Nombre: jose
Memoria Liberada
```

Figura 61: "Salida por pantalla del programa lista4.cpp"

Veamos como los inserta paso a paso:

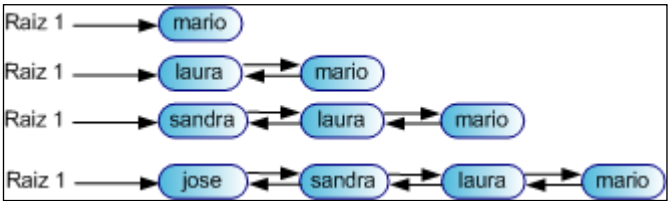


Figura 62: "Análisis de la Modalidad de Inserción en la Lista según lista4.cpp"

Colas

Hay un dicho en Argentina que revela: "Argentina es el país de las colas", en efecto, cualquier lugar donde uno pueda asistir sufre de este "Mal". En realidad no es que sea malo, en términos informáticos, por el contrario, es de suma utilidad. Ocurre que en la realidad, quienes están encargados de la atención de la cola no es un microprocesador y la memoria del ordenador, sino personas, y aquí empiezan las demoras…

Técnicamente podemos definir una cola como:

> "Secuencia de Datos Vinculados, donde el elemento a extraer siempre será el primero de la secuencia y cualquier nuevo elemento a ingresar, siempre lo hará al final de la misma.
>
> Lo dicho anteriormente, en términos de ciencias de la computación, se entiende como un mecanismo FIFO (First Input, First Output –Primero en Entrar, Primero en Salir-), en una secuencia de datos dada.

Veamos una interpretación gráfica de dicha definición:

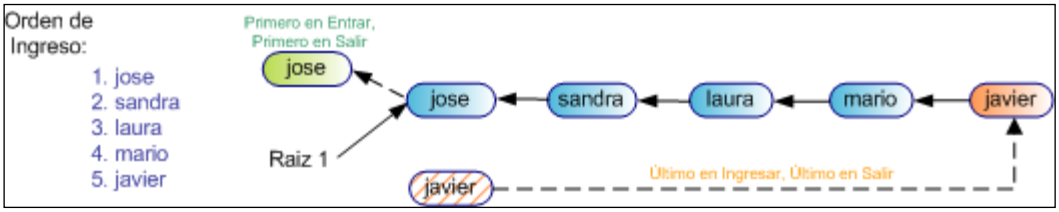


Figura 63: "Análisis Gráfico de una Cola"

Note que el primero en salir de la cola, es quien ingreso en primer lugar, en nuestro ejemplo se trata del nodo que contiene la palabra "jose". Mientras que aquel nodo que quisiese ingresar a la cola, lo hará al final de la misma, este es el caso del nodo que contiene la palabra "javier".

Implementación de una Cola a través de Listas Simplemente Enlazadas

Como se vio en el apartado "Listas" del presente capítulo, una de las principales características de estas últimas, era su aptitud de crecimiento/decrecimiento dinámico. Las Colas se caracterizan por su dinamismo en cuanto al ingreso y/o egreso de nodos de la misma, por lo que la aplicación de una lista, al momento de codificar una cola, con la restricción FIFO, es una alternativa viable.

Archivo: **cola.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4       #include <stdio.h>
5
6       //Definici¢n del Nodo de la Cola
7       struct nodo{
8               char nombre[20];
9               nodo* next;
10              };
11
12      void insert(nodo*&,char*);
13      void borrar(nodo*&);
14      void borrarlista(nodo*&);
15      void mostrarlista(nodo*);
16
17      void main(void)
18      {
19       nodo* raiz=NULL;//Raiz de la Lista
20
21       char nombres[5][20];
22
23       strcpy(nombres[0],"Mario");
24       strcpy(nombres[1],"jose");
25       strcpy(nombres[2],"laura");
26       strcpy(nombres[3],"sandra");
27
28       clrscr();
29       for(int i=0;i<4;i++)
30       {
31        insert(raiz,nombres[i]);
32       }
33
34       cout<<"\n\nINICIO DE MUESTRA: COLA COMPLETA";
35       mostrarlista(raiz);
36       cout<<"\n";
37
38       for(i=0;i<4;i++)
39       {
40        borrar(raiz);
41        cout<<"\nCola con "<<(3-i)<<" elementos";
42        mostrarlista(raiz);
43        cout<<"\n";
44       }
45
46       borrarlista(raiz);
47
```

```
48      if(raiz==NULL) cout<<"\nMemoria Liberada";
49     }
50
51     void insert(nodo*& raiz, char* string)
52     {
53      nodo *ptr=new nodo;
54      strcpy(ptr->nombre,string);
55
56      if(raiz==NULL)
57       {//Primer nodo de la Cola
58        ptr->next=raiz;
59        raiz=ptr;
60       }
61      else
62      {//Inserta al final de la Cola
63       nodo *ptrant=NULL;
64       nodo *go=raiz;
65       while(go!=NULL)
66       {
67        ptrant=go;
68        go=go->next;
69       }
70
71       ptrant->next=ptr;
72       ptr->next=NULL;
73      }
74     }
75
76
77     void borrar(nodo*& raiz)
78     {//Quita el primer nodo de la Cola
79       if(raiz==NULL) return;//No hay mas nodos en la cola
80
81       nodo* elim=raiz;
82       if(elim->next==NULL)
83        {//queda un nodo en la cola;
84         delete elim;
85         elim=NULL;
86         raiz=NULL;
87        }
88       else
89       {//Hay al menos dos nodos en la cola
90        raiz=raiz->next;//la raiz pasa a puntar al siguient nodo de la cola
91        delete elim;//borra al primero en la cola
92        elim=NULL;
93       }
94     }
95
96     void borrarlista(nodo*& raiz)
97     {
```

```
98      nodo* ptr=raiz;
99      nodo* elim=raiz;
100
101     while(ptr!=NULL)
102      {
103       elim=ptr;
104       ptr=ptr->next;
105       delete elim;
106       elim=NULL;
107      }
108
109      raiz=NULL;
110    }
111
112    void mostrarlista(nodo* raiz)
113    {
114     nodo *ptr=raiz;
115
116     while(ptr!=NULL)
117      {
118       cout<<"\nNombre: "<<ptr->nombre;
119       ptr=ptr->next;
120      }
121    }
```

Veamos como se comporta la salida por pantalla del programa “cola.cpp”:

```
INICIO DE MUESTRA: COLA COMPLETA
Nombre: Mario
Nombre: jose
Nombre: laura
Nombre: sandra

Cola con 3 elementos
Nombre: jose
Nombre: laura
Nombre: sandra

Cola con 2 elementos
Nombre: laura
Nombre: sandra

Cola con 1 elementos
Nombre: sandra

Cola con 0 elementos

Memoria Liberada
```

Figura 64: “Salida por pantalla del programa cola.cpp”

El programa “cola.cpp” carga un arreglo que contiene los nombres: “Mario”, “jose”, “laura” y “sandra” (en dicho orden), en la cola. Posteriormente muestra la cola cargada, para luego empezar a quitar de a un nodo de la misma, mostrando como va

quedando la cola a medida que se quitan los nodos. El resultado de la ejecución del programa está representado en la Figura 64.

Pilas

Cuando uno visita una biblioteca, es factible que vea libros apilados en numerosos lugares, por lo que podemos ver libros situados uno sobre otros donde, al momento de necesitar uno de ellos, se quitará primero los posicionados en la parte superior y así, en forma descendente. Esto mismo es el concepto de pila, pero viéndolo desde el punto de vista técnico, podemos definir una pila como:

> ”Secuencia de Datos Vinculados, donde el elemento a extraer siempre será el primero de la secuencia y cualquier nuevo elemento a ingresar, siempre lo al principio (o en la parte superior) de la misma.
>
> Lo dicho anteriormente, en términos de ciencias de la computación, se entiende como un mecanismo LIFO (Last Input, First Output –Primero en Entrar, Primero en Salir-), en una secuencia de datos dada.

Veamos una interpretación gráfica de dicha definición:

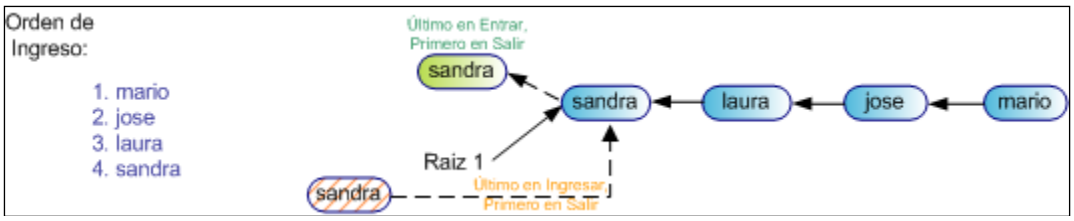


Figura 65: “Análisis Gráfico de una Pila”

Note que el primero en salir de la cola, es quien ingreso en último lugar, en nuestro ejemplo se trata del nodo que contiene la palabra “sandra”. Mientras que aquel nodo que quisiese ingresar a la cola, lo hará al principio de la misma, este es el caso del nodo que contiene la palabra “sandra” nuevamente.

Implementación de una Pila a través de Listas Simplemente Enlazadas

Como se vio en el apartado “Listas” y en el de “Colas” del presente capítulo, es factible implementar una secuencia de datos vinculados que se comportan en forma dinámica a través de Listas. En el caso de una Pila, es viable implementarse a través de una lista con una restricción LIFO.

Archivo: **pila.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4       #include <stdio.h>
5
6       //Definici¢n del Nodo de la Pila
7       struct nodo{
```

```
8               char nombre[20];
9               nodo* next;
10              };
11
12      void push(nodo*&,char*);
13      void pop(nodo*&);
14      void borrarlista(nodo*&);
15      void mostrarlista(nodo*);
16
17      void main(void)
18      {
19       nodo* raiz=NULL;//Raiz de la Lista
20
21       char nombres[5][20];
22
23       strcpy(nombres[0],"Mario");
24       strcpy(nombres[1],"jose");
25       strcpy(nombres[2],"laura");
26       strcpy(nombres[3],"sandra");
27
28       clrscr();
29       for(int i=0;i<4;i++)
30       {
31        push(raiz,nombres[i]);
32       }
33
34       cout<<"\n\nINICIO DE MUESTRA: PILA COMPLETA";
35       mostrarlista(raiz);
36       cout<<"\n";
37
38       for(i=0;i<4;i++)
39       {
40        pop(raiz);
41        cout<<"\nPila con "<<(3-i)<<" elementos";
42        mostrarlista(raiz);
43        cout<<"\n";
44       }
45
46       borrarlista(raiz);
47
48       if(raiz==NULL) cout<<"\nMemoria Liberada";
49      }
50
51      void push(nodo*& raiz, char* string)
52      {
53       nodo *ptr=new nodo;
54       strcpy(ptr->nombre,string);
55
56       ptr->next=raiz;
```

```
57      raiz=ptr;
58     }
59
60
61     void pop(nodo*& raiz)
62     {//Quita el primer nodo de la Cola
63       if(raiz==NULL) return;//No hay mas nodos en la cola
64
65       nodo* elim=raiz;
66       if(elim->next==NULL)
67        {//queda un nodo en la cola;
68         delete elim;
69         elim=NULL;
70         raiz=NULL;
71        }
72       else
73       {//Hay al menos dos nodos en la cola
74        raiz=raiz->next;//la raiz pasa a puntar al siguient nodo de la cola
75        delete elim;//borra al primero en la cola
76        elim=NULL;
77       }
78     }
79
80     void borrarlista(nodo*& raiz)
81     {
82      nodo* ptr=raiz;
83      nodo* elim=raiz;
84
85      while(ptr!=NULL)
86       {
87        elim=ptr;
88        ptr=ptr->next;
89        delete elim;
90        elim=NULL;
91       }
92
93       raiz=NULL;
94     }
95
96     void mostrarlista(nodo* raiz)
97     {
98      nodo *ptr=raiz;
99
100     while(ptr!=NULL)
101      {
102       cout<<"\nNombre: "<<ptr->nombre;
103       ptr=ptr->next;
104      }
105    }
```

Veamos como se comporta la salida por pantalla del programa “pila.cpp”:

```
INICIO DE MUESTRA: PILA COMPLETA
Nombre: sandra
Nombre: laura
Nombre: jose
Nombre: Mario

Pila con 3 elementos
Nombre: laura
Nombre: jose
Nombre: Mario

Pila con 2 elementos
Nombre: jose
Nombre: Mario

Pila con 1 elementos
Nombre: Mario

Pila con 0 elementos

Memoria Liberada
```

Figura 65: “Salida por pantalla del programa pila.cpp”

El programa “pila.cpp” carga (**push**) un arreglo que contiene los nombres: “Mario”, “jose”, “laura” y “sandra” (en dicho orden), en la pila. Posteriormente muestra la pila cargada, para luego empezar a quitar de a un nodo de la misma (**pop**), mostrando como va quedando la pila a medida que se quitan los nodos. El resultado de la ejecución del programa está representado en la Figura 65.

Árboles Binarios

Introducción

Un árbol binario es una estructura no lineal, representado como un conjunto finito de elementos que se encuentra vacío o dividido en tres subconjuntos:

1. El primer subconjunto contiene un único elemento denominado raíz. A través de este elemento lograremos el acceso a los restantes elementos que conforman el árbol.
2. Los dos subconjuntos restantes pueden interpretarse como el subconjunto izquierdo y derecho del árbol. No son más que las ramificaciones posibles a partir del nodo raíz.

Note que un Árbol Binario, tiene tan solo dos caminos posibles: El izquierdo ó el derecho, pero no ambos, lo cual es la premisa fundamental que determina su nombre, “Binarios”.

El Árbol Binario, puede ser implementado mediante arreglos o bien, mediante listas. En el caso de la implementación mediante arreglos, se restringiría la flexibilidad del árbol a un entorno definido en una magnitud determinada de memoria, no permitiendo su crecimiento dinámico. En cuanto a la implementación mediante listas, permite el libre crecimiento y decrecimiento del árbol, permitiendo un mejor manejo de la memoria del ordenador, pero en requiere manejo dinámico de memoria bien

controlado, ya que cualquier nodo del árbol no liberado, se mantiene luego como "Memoria Reservada", provocando una merma en el rendimiento del ordenador.

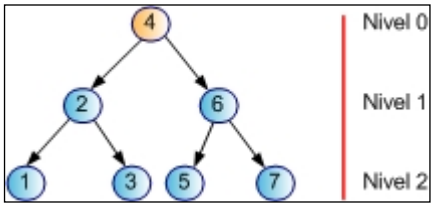


Figura 66: "Representación Conceptual de un Árbol Estrictamente Binario"

Se denomina nodo de un árbol, a aquella estructura que almacena la información de sus estructuras hijas, como del valor clave del nodo, con su información asociada. Por valor clave del nodo, se entiende aquel que lo identifica en forma única (Ejemplo: el 6 en la Figura 66, o el 5, o el 3, etc.). La información asociada a la clave del nodo, puede ser cualquier dato que el programador desee almacenar en dicho nodo, como por ejemplo una instancia de la estructura "persona" con los datos de "Juan Pérez".

Por nodo inferior izquierdo del nodo actual, se entiende a aquél nodo que puede ser localizado mediante el puntero izquierdo del nodo actual. Mientras que por nodo inferior derecho del nodo actual, se entiende aquél nodo que pude ser localizado mediante el puntero derecho del nodo actual. Por ejemplo, si nuestro nodo actual es el que contiene el valor "4", su nodo inferior izquierdo será el "2", mientras que su nodo inferior derecho será el "6".

Por nodo "Raíz" de un árbol se entiende aquél nodo que no posee padre, es huérfano, en nuestro ejemplo de la figura 66, es el "4".

Se entiende por nivel "0", al ocupado por el nodo "Raíz", mientras que los subsiguientes nodos, tendrán un nivel más que sus padres. Ejemplo: El nodo "2" tiene el nivel 1, debido a que su padre es "Raíz" y posee el nivel 0, entonces 0+1= 1.

Se entiende por nodo "hoja" a aquel nodo que no posee hijos. Se entiende por nodo Hermano, a dos nodos que corresponden al mismo padre. Se entiende por profundidad de un árbol al máximo nivel de cualquier hoja en un árbol. En el ejemplo de la figura 66, tanto 1, como 3, 5 y 7 son hojas y se encuentran en el máximo nivel "2", entonces la profundidad del árbol es "2".

Se entiende por"Nodo Estrictamente Binario", a aquél árbol cuyos nodos padres poseen todos dos nodos hijos, de lo contrario no poseen ninguno, lo que implicaría que son nodos "hojas". El ejemplo de la figura 66, es un nodo estrictamente binario, note que los nodos padres: 4,2 y 6, poseen todos dos hijos y, aquellos que no poseen dos hijos son nodos hojas, es el caso de los nodos 1, 3, 5 y 7.

La cantidad de nodos que un árbol estrictamente binario puede almacenar, está dado por:

$$\sum_{j=0}^{n} 2^{j}$$

Continuando con el ejemplo, la cantidad de nodos en un árbol estrictamente binario es de $2^0+2^1+2^2=7$ nodos.

Cómo podrá observar, un árbol binario es una estructura más eficiente para búsqueda que una lista. En el caso de una lista de tamaño "N", debe efectuar en promedio N/2 comparaciones para localizar un valor, mientras que un árbol binario, como el de la figura 66, efectúa 3 comparaciones. Es claro que un árbol binario nunca demorará más que una lista para localizar un valor, debido a que el peor caso de un

árbol binario, es una lista ordenada, entonces a lo sumo demorará lo mismo que una lista, nunca de más.

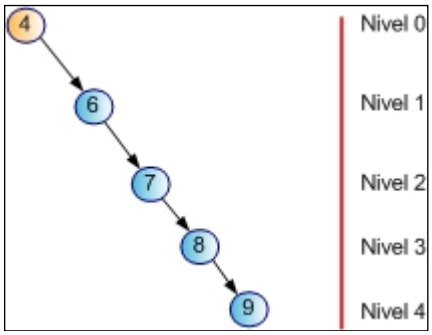


Figura 67: "Representación Conceptual del Peor Caso de un Árbol Binario"

Como primera apreciación, note que el árbol binario de la figura 67, no es un árbol estrictamente binario, ya que de los nodos padres, existe al menos uno que no posee dos hijos (En este caso, ninguno de los padres posee dos hijos, ni el nodo 4, ni el 6, ni el 7, ni tampoco el 5), suficiente para no ser considerado Árbol Estrictamente Binario. Por consiguiente la fórmula de la página anterior no coincidirá en cantidad de nodos para el árbol, pero sí nos permitiría dar una orientación de cuál es la cantidad de nodos que el árbol, con la profundidad actual de 4, puede almacenar.

Como segunda apreciación, note que en el peor de los casos, el Árbol Binario se comporta como una lista ordenada.

Como tercera apreciación fundamental, note que el árbol binario es sumamente dependiente del orden de inserción de sus valores. El árbol binario de la figura 67, se obtuvo insertando los valores: 4,6,7,8 y 9 en dicho orden, pero note como hubiese cambiado el árbol si el orden de inserción de dichos valores hubiese sido: 7, 4, 6, 8 y 9:

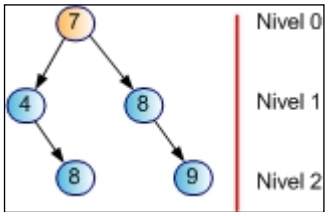


Figura 68: "Representación Conceptual del Peor Caso Mejorado de un Árbol Binario"

Note como modificando el orden de inserción de los valores, disminuimos dos niveles en el árbol binario, ¿Consecuencia? Note que para buscar el valor "9" en el árbol binario de la figura 67 se requieren 5 comparaciones, mientras que en el árbol de la figura 68, se requieren 3 comparaciones. En este último caso, se emplean 2 comparaciones menos, lo que implica un 40% en promedio menos de comparaciones. Los ejemplos planteados sencillos a los efectos de poder internalizarlos rápidamente, pero intuya ¿cuánto representaría este 40% menos de comparaciones en un árbol de cien mil elementos?

Implementación de un Árbol Binario

Archivo: **abinario.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       struct nodo{
5               int valor;
6               nodo *izq;
7               nodo *der;
8               nodo()
9               {
10              izq=NULL;
11              der=NULL;
12              }
13             };//fin de la definici¢n de la estructura
14
15      void insert(nodo*&,int);
16      void borrar(nodo*&,int);
17      void mostrar(nodo*);
18
19      void main(void)
20      {
21       nodo* raiz=NULL;
22
23       insert(raiz,4);
24       insert(raiz,2);
25       insert(raiz,6);
26       insert(raiz,1);
27       insert(raiz,3);
28       insert(raiz,5);
29       insert(raiz,7);
30
31       clrscr();
32       cout<<"Inicial: ";
33       mostrar(raiz);
34      /* borrar(raiz,4);
35       mostrar(raiz);*/
36       borrar(raiz,4);
37       cout<<"\nLuego de Borrar 4: ";
38       mostrar(raiz);
39       borrar(raiz,2);
40       cout<<"\nLuego de Borrar 2: ";
41       mostrar(raiz);
42       borrar(raiz,6);
43       cout<<"\nLuego de Borrar 6: ";
44       mostrar(raiz);
45       borrar(raiz,1);
```

```
46      cout<<"\nLuego de Borrar 1: ";
47      mostrar(raiz);
48      borrar(raiz,3);
49      cout<<"\nLuego de Borrar 3: ";
50      mostrar(raiz);
51      borrar(raiz,5);
52      cout<<"\nLuego de Borrar 5: ";
53      mostrar(raiz);
54      borrar(raiz,7);
55      cout<<"\nLuego de Borrar 7: ";
56      mostrar(raiz);
57     }
58
59     void insert(nodo*& raiz,int val)
60     {
61      nodo* ins=new nodo();
62      ins->valor=val;
63
64      if(ins==NULL) return;//no pudo crear el nodo
65
66      if(raiz==NULL)
67       {//primer nodo del arbol
68         raiz=ins;
69       }
70      else
71       {
72        nodo* p=raiz;
73        nodo* ant=NULL;
74
75        while(p!=NULL)
76        {
77          ant=p;
78
79          if(ins->valor==p->valor)
80           {
81            cout<<"El nodo que quiere insertar ya existe";
82            return;
83           }
84
85          if(ins->valor<p->valor)
86             p=p->izq;
87            else
88             p=p->der;
89        }//fin del while
90
91        if(ins->valor<ant->valor)
92             ant->izq=ins;
93            else
94             ant->der=ins;
```

```
95        }//fin del else
96      }//fin de la funcion
97
98      void borrar(nodo*& raiz,int valor)
99      {
100       nodo* ant=NULL;
101       nodo* p=raiz;
102
103       if(p==NULL)return;//arbol vacio
104
105       while(p!=NULL && valor!=p->valor)
106       {
107        ant=p;
108
109        if(valor<p->valor)
110           p=p->izq;
111          else
112           p=p->der;
113       }
114
115       if(p==NULL)
116        {
117         cout<<"\nEl valor que desea borrar no existe";
118         return;
119        }
120
121       if(p->der==NULL && p->izq==NULL)
122       {
123         if(ant==NULL)
124          {
125           delete raiz;
126           raiz=NULL;
127           return;
128          }
129         else
130          {
131           if(p->valor < ant->valor)
132             ant->izq=NULL;
133            else
134             ant->der=NULL;
135
136           delete p;
137           p=NULL;
138           return;
139          }
140       }
141
142       if(p->der==NULL && p->izq!=NULL)
143       {
144         if(ant==NULL)
```

```
145         {
146           nodo* elim=raiz;
147           raiz=raiz->izq;
148           delete elim;
149           elim=NULL;
150           return;
151         }
152        else
153         {
154           nodo* elim=p;
155           p=p->izq;
156
157           if(p->valor < ant->valor)
158              ant->izq=p;
159            else
160              ant->der=p;
161
162           delete elim;
163           elim=NULL;
164           return;
165         }
166      }
167
168      if(p->der!=NULL)
169      {
170       nodo* ptr=p->der;
171
172       if(ptr->izq!=NULL)
173        {
174         nodo* a=NULL;
175         nodo* b=NULL;
176
177         while(ptr!=NULL)
178          {
179           a=b;
180           b=ptr;
181           ptr=ptr->izq;
182          }
183
184         a->izq=NULL;
185         b->izq=p->izq;
186         b->der=p->der;
187         nodo* elim=p;
188
189         if(ant==NULL)
190           {
191            raiz=b;
192           }
193          else
```

```
194           {
195           if(b->valor < ant->valor)
196                   ant->izq=b;
197                  else
198                   ant->der=b;
199           }
200
201          delete elim;
202          elim=NULL;
203          return;
204        }
205       else
206        {
207          ptr->izq=p->izq;
208
209          if(ant==NULL)
210            {
211              raiz->der=NULL;
212              raiz=ptr;
213            }
214           else
215            {
216             if(ptr->valor < ant->valor)
217                   ant->izq=ptr;
218              else
219                   ant->der=ptr;
220
221            }
222
223          delete p;
224          p=NULL;
225          return;
226        }
227      }
228    }
229
230    void mostrar(nodo* p)
231    {
232     if(p==NULL) return;//arbol vacio
233
234     cout<<p->valor;
235     mostrar(p->izq);
236     mostrar(p->der);
237    }
```

La Salida del programa “abinario.cpp”, muestra en un primer momento el árbol completo luego de todas las inserciones, y luego muestra el estado del árbol a medida que se efectúan los sucesivos borrados de los nodos.

```
Inicial: 4213657
Luego de Borrar 4: 521367
Luego de Borrar 2: 53167
Luego de Borrar 6: 5317
Luego de Borrar 1: 537
Luego de Borrar 3: 57
Luego de Borrar 5: 7
Luego de Borrar 7:
```

Figura 69: “Salida por pantalla del programa abinaria.cpp”

Análisis del Algoritmo de Inserción

Como se mencionó en apartados anteriores, un árbol binario es sumamente susceptible al orden de inserción de los valores. En este caso, se analizará cómo se va formando el árbol binario con el siguiente orden: 4261357.

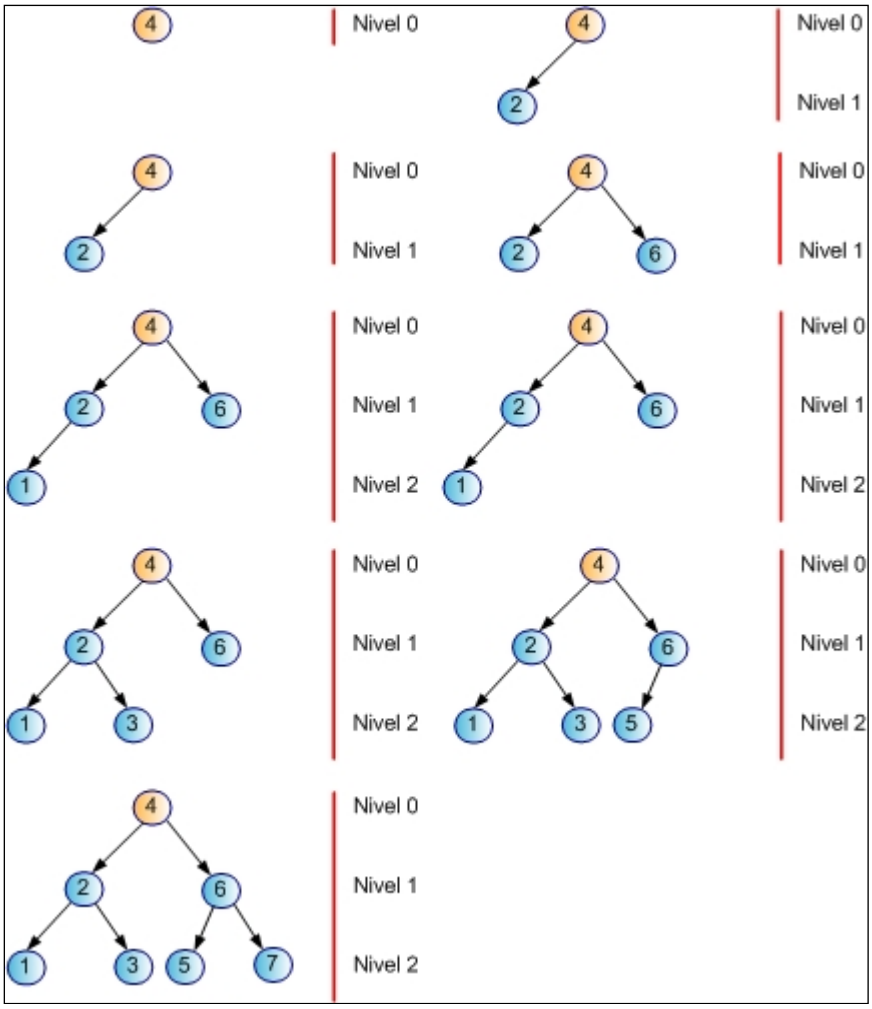


Figura 70: “Formación del Orden Binario de acuerdo al Orden de Inserción 4261357”

La figura 70, va mostrando los distintos estados por los que atraviesa el árbol binario a medida que se van insertando los distintos valores. El gráfico se lee de izquierda a derecha y de arriba hacia abajo, así en el ángulo superior izquierdo se

encuentra el árbol binario luego de habérsele insertado el “4”, a su derecha se encuentra el árbol binario luego de habérsele insertado el “2”, y así en forma sucesiva.

Note que al insertar, el algoritmo consulta nodo a nodo, si el valor a insertar es menor o mayor al comparado contra el nodo, si es menor desciende por el hijo izquierdo, si es mayor desciende por el hijo derecho. En caso de que el valor exista previamente en el árbol, puede optar por:

1. Llevar un contador de las repeticiones,
2. Responder con un mensaje como “El Nodo ya Existe en el árbol”

En el programa “abinario.cpp”, se optó por la segunda alternativa, si el valor existe se responde con un mensaje al usuario.

Una vez encontrada la posición donde insertar el nodo, se lo vincula con el padre para incorporarlo dentro del árbol

Análisis del Algoritmo de Borrado

A continuación, previo análisis del algoritmo de inserción, veremos como trabaja el algoritmo de borrado de un árbol binario paso a paso, analizando los distintos estados por los que transita el árbol:

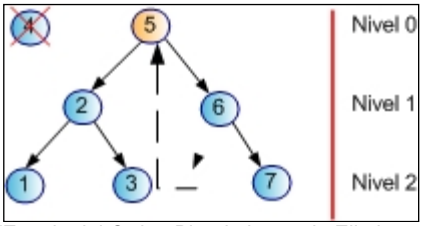


Figura 71: “Estado del Orden Binario luego de Eliminar el Nodo ‘4’”

Note que al eliminar la raíz del árbol, se busca el valor inmediatamente superior, situado en la extremidad izquierda inferior del nodo a la derecha de la raíz. El nodo a la derecha de la raíz es el “6”, cuya extremidad izquierda inferior está representada por el nodo “5”, en consecuencia el nodo “5” pasa a situarse como la nueva “raíz” manteniendo el orden del Árbol Binario. Obviamente al reubicarse el nodo “5” como la nueva raíz, el enlace con su antiguo nodo padre, el “6”, es roto o quebrado, anulándose el mismo.

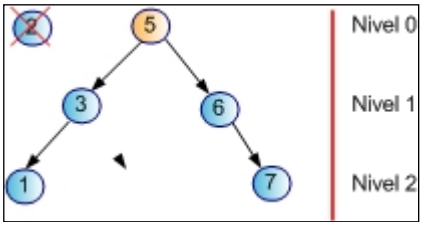


Figura 72: “Estado del Orden Binario luego de Eliminar el Nodo ‘2’”

En la figura 72, se elimina el nodo con el valor 2, lo que provoca que se busque el valor inmediatamente superior al actual eliminado. Dicho valor en nuestro árbol, es el 3, éste valor es ubicado en la posición del nodo eliminado, de cara a mantener el orden del árbol binario. Ahora el nodo 3, no es más hijo del nodo 2, ya que es el eliminado, sino que pasa a ser hijo del nodo “5”.

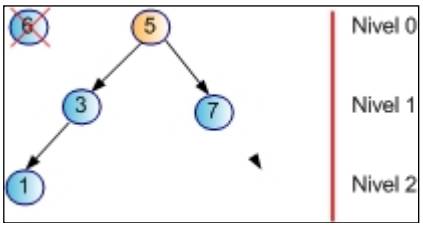


Figura 73: “Estado del Orden Binario luego de Eliminar el Nodo ‘6’”

En la figura 73 se elimina el nodo “6”, provocando que en su posición se ubique el nodo inmediatamente superior, el 7, y cambiando el padre del 7 desde el antiguo padre, el “6”, al nuevo padre, el “5”.

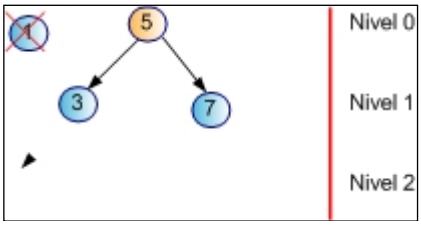


Figura 74: “Estado del Orden Binario luego de Eliminar el Nodo ‘1’”

En la figura 74 se elimina el nodo “1”, provocando que se elimine directamente el nodo ya que es una hoja, anulando el enlace con su antiguo padre, el nodo “3”.

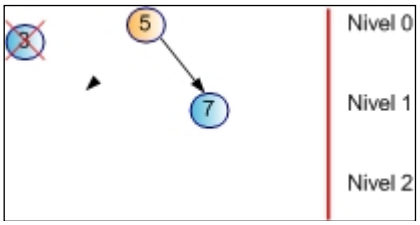


Figura 75: “Estado del Orden Binario luego de Eliminar el Nodo ‘3’”

En la figura 75 se elimina el nodo “3”, provocando que se elimine directamente el nodo ya que es una hoja, anulando el enlace con su antiguo padre, el nodo “5”.

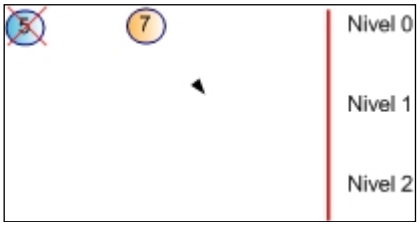


Figura 76: “Estado del Orden Binario luego de Eliminar el Nodo ‘5’”

En la figura 76 se elimina el nodo “3”, provocando que se elimine directamente el nodo y, que se ubique como raíz el nodo inmediatamente superior, en este caso el nodo 7, anulando el enlace con su antiguo padre, el nodo “5” y transformándose en el nuevo nodo raíz.

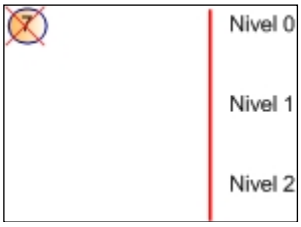


Figura 77: "Estado del Orden Binario luego de Eliminar el Nodo '7'"

En la figura 77 se elimina el nodo "7", provocando que se elimine directamente el nodo quedando sin nodos el árbol binario.

Análisis del Algoritmo de Muestra

A partir del árbol cargado inicialmente, veremos como trabaja el algoritmo de muestra del árbol binario, que tiene la particularidad de ser recursivo.

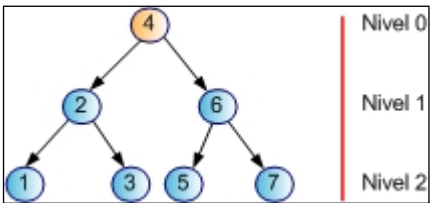


Figura 78: "Representación Conceptual de un Árbol Estrictamente Binario"

Para analizar el funcionamiento, se transcribirá la sección del programa "abinario.cpp" que implementa la función de muestra:

```
230    void mostrar(nodo* p)
231    {
232     if(p==NULL) return;//arbol vacio
233
234     cout<<p->valor;
235     mostrar(p->izq);
236     mostrar(p->der);
237    }
```

La condición de corte del algoritmo de muestra recursivo es:

```
if(p==NULL)
```

Lo que nos da la pauta de que solo retornará al encontrar una hoja en el árbol binario. Inicialmente, se le pasa la raíz como parámetro, lo que provoca que se muestre el valor "4", y se invoque con la dirección del nodo inferior izquierdo (`p->izq`, que es el nodo "2"). Al ingresar con la dirección del nodo 2, como p es distinto de null, se muestra el "2", y se reinvoca el algoritmo con la dirección del nodo inferior izquierdo (`p->izq`, el nodo "1"). Nuevamente al ingresar con la dirección del nodo "1", se muestra el "1" y se reinvoca con la dirección del nodo inferior izquierdo de "1", como es NULL al ingresar nuevamente a la función, se produce un retorno y ¿Dónde continúa?.

La última llamada en la pila es la proveniente desde el nodo "1", lo que provoca que siga ejecutándose el cuerpo de la función, ahora se reinvoca la función mostrar, con

la dirección del nodo inferior derecho, como nuevamente es NULL, la condición de corte provocará que al llamarse la función se produzca el retorno anticipado, de esta forma concluye la ejecución de mostrar con la dirección del nodo 1, ¿Dónde continúa?.

La función que llamó a mostrar con la dirección del nodo 1, es aquella que tenía la dirección del nodo 2, con lo que continúa mostrando la parte derecha del nodo 2, allí reinvoca la función mostrar con la dirección del nodo inferior derecho del nodo 2, que es el nodo 3. Al ingresar la a la función mostrar con la dirección del nodo "3", muestra el 3, y sigue mostrando ahora el nodo inferior izquierdo del nodo 3, como es NULL, retorna en forma anticipada y sigue mostrando el nodo inferior derecho del nodo "3", nuevamente al ser NULL, retorna anticipado.

Terminado de ejecutarse la función mostrar para la dirección del nodo "3", sigue ejecutando lo que restaba de la función mostrar con la dirección del nodo "2", como de la función con la dirección del nodo "2" no restaba nada más, continúa ejecutando la función mostrar con la dirección del nodo inferior derecho de "4". A partir de allí, se repite el mismo funcionamiento del algoritmo para el lado derecho del árbol.

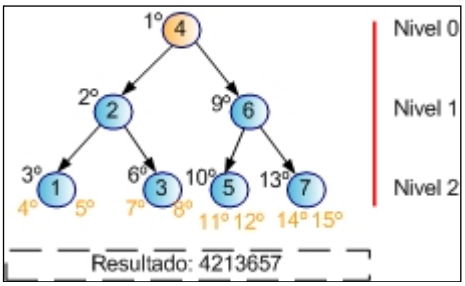


Figura 79: "Representación Conceptual del Funcionamiento del Algoritmo de Muestra"

Note que la figura 79, indica las direcciones del puntero "nodo" que es pasado como parámetro cada vez que se invoca la función mostrar en forma recursiva. Inicialmente como se mencionó, se da la dirección del nodo raíz y se muestra su valor (figura como 1º), luego se sigue con el nodo inmediatamente inferior izquierdo, y se muestra el 2 (figura como 2º), luego con el nodo inmediatamente inferior izquierdo al nodo 2, y se muestra el 1, cuando quiere invocarse el nodo inmediatamente inferior izquierdo y el derecho del nodo 1, en ese orden, provoca un inicio de retorno del algoritmo recursivo (figuran como 4º y 5º en naranja, porque son "NULLs" y no agregan nada al resultado), el cual continuará con el nodo inferior derecho de 2 (invocador de muestra con la dirección del nodo 1) para concluir su muestra. De esta forma sucesivamente. Note que la figura le indica cómo se recorre y muestra el árbol, mencionando con el número de orden al margen del nodo, cuál es la dirección del nodo con la que trabajará el algoritmo de muestra en cada instante.

Referencias

- "Estructuras de Datos con C y C++"; Langsam, Aungenstein y Tenenbaum; Editorial Prentice Hall; México, 1997.
- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995.
- "Algoritmos y Estructuras de Datos"; N. Wirth; Editorial Prentice Hall, 1987.
- "Estructuras de Datos"; L. Joyanes y I. Zahonero; Editorial McGraw-Hill, 1998.
- "Classic Data Structures in C++"; T. Budd; Editorial Addison-Wesley, 1994.
- "Queuing Systems"; L. Kleinrock; Editorial Wiley; 1975.
- "Data Structures and Algorithm Analysis in C++"; A.W. Weiss; Editorial Benjamin/Cummings; 1994.

5

Algoritmos de Ordenamiento

Introducción a los Algoritmos de Ordenamiento

En la operatoria normal de cualquier empresa, las tareas de ordenamiento son algo propio, se ordenan facturas para pagos, órdenes de compras, se priorizan tareas, entre otras cosas. Así como en la operatoria normal de una empresa, los aplicativos, o bien el software de base, necesitan frecuentemente ejecutar tareas de ordenamiento, como ser el listado de los archivos de un directorio por nombre, por fecha, por propietario. En fin, la aplicabilidad del ordenamiento a los distintos estratos de trabajo, son variados, fluidos y sumamente necesarios.

Dentro de los algoritmos tradicionales de ordenamiento, se encuentra el algoritmo de burbujas, selección e inserción. Éstos, de Orden N^2, posibilitan el ordenamiento de una secuencia de estructuras, que podrán ser números, cadenas, letras, etc con un número de comparaciones eficaz, pero que dista mucho de ser eficiente.

Este capítulo, se focaliza sobre algoritmos más refinados de ordenamiento, como ser el Shellsort, Heapsort, Combsort, entre otros. Los mismos, son una aproximación más realista, a los algoritmos empleados en un software de base open source, comercial y/o cualquier aplicativo de porte empresarial.

Shellsort

Uno de las principales deficiencias del Algoritmo de Inserción, es que solo permite el intercambio de elementos si y solo si, son adyacentes, en caso de no serlo, debe recorrer la totalidad del arreglo, para finalmente, concretar la permutación. Esta desventaja es salvada en el algoritmo de Shell, permitiendo éste, el intercambio de elementos extremos, como se verá en el ejemplo dentro del presente apartado.

La Ordenación por Shell, nunca efectuará más de $N^{3/2}$ comparaciones, lo que pone de manifiesto en hechos, la mejora con respecto al algoritmo de inserción de orden N^2.

El programa “shellsor.cpp” es una de las tantas implementaciones disponibles del algoritmo de ordenamiento por shell. El mismo posee un arreglo de diez enteros definido en el código, el que será ordenado por shell, mostrando el arreglo antes y después del ordenamiento en la salida por pantalla.

Archivo: **“shellsor.cpp”**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void shellsort(int[],int tam);
5
6       void main(void)
7       {
8        int aordenar[10]={96,58,79,2,9,36,24,13,17,49};
9
10       clrscr();
11       cout<<"Arreglo Inicial\n";
12       for(int i=0;i<10;i++)
13         cout<<aordenar[i]<<"-";
14
```

```
15      shellsort(aordenar,10);
16      cout<<"\n\nArreglo Ordenado\n";
17      for(i=0;i<10;i++)
18        cout<<aordenar[i]<<"-";
19     }
20
21     void shellsort(int aordenar[],int tam)
22     {
23      int numero=5;
24      int incrementos[]={11,7,5,3,1};
25      int incr,j,k,span,y;
26
27      for(incr=0;incr<numero;incr++)
28      {
29       span=incrementos[incr];
30
31        for(j=span;j<tam;j++)
32         {
33          y=aordenar[j];
34          for(k=j-span;k>=0 && y<aordenar[k];k-=span)
35                    aordenar[k+span]=aordenar[k];
36
37          aordenar[k+span]=y;
38         }
39      }
40     }
```

La salida por pantalla del programa “shellsor.cpp”, es la siguiente:

```
Arreglo Inicial
96-58-79-2-9-36-24-13-17-49-

Arreglo Ordenado
2-9-13-17-24-36-49-58-79-96-
```

Figura 80: “Salida por pantalla del programa shellsor.cpp”

Note que el arreglo inicial dado al algoritmo de ordenamiento en el programa “shellsor.cpp”, en absoluto es un caso favorable para el mismo, por el contrario, posee muchos elementos “pesados o tortugas” (*claves de mayor magnitud con respecto a las restantes, al inicio del arreglo*) al inicio.

Veamos ahora, cómo trabaja paso a paso el algoritmo de ordenación por Shell, para el arreglo con los valores 96, 58, 79, 2, 9, 36, 24, 13, 17, y 49 en este orden.

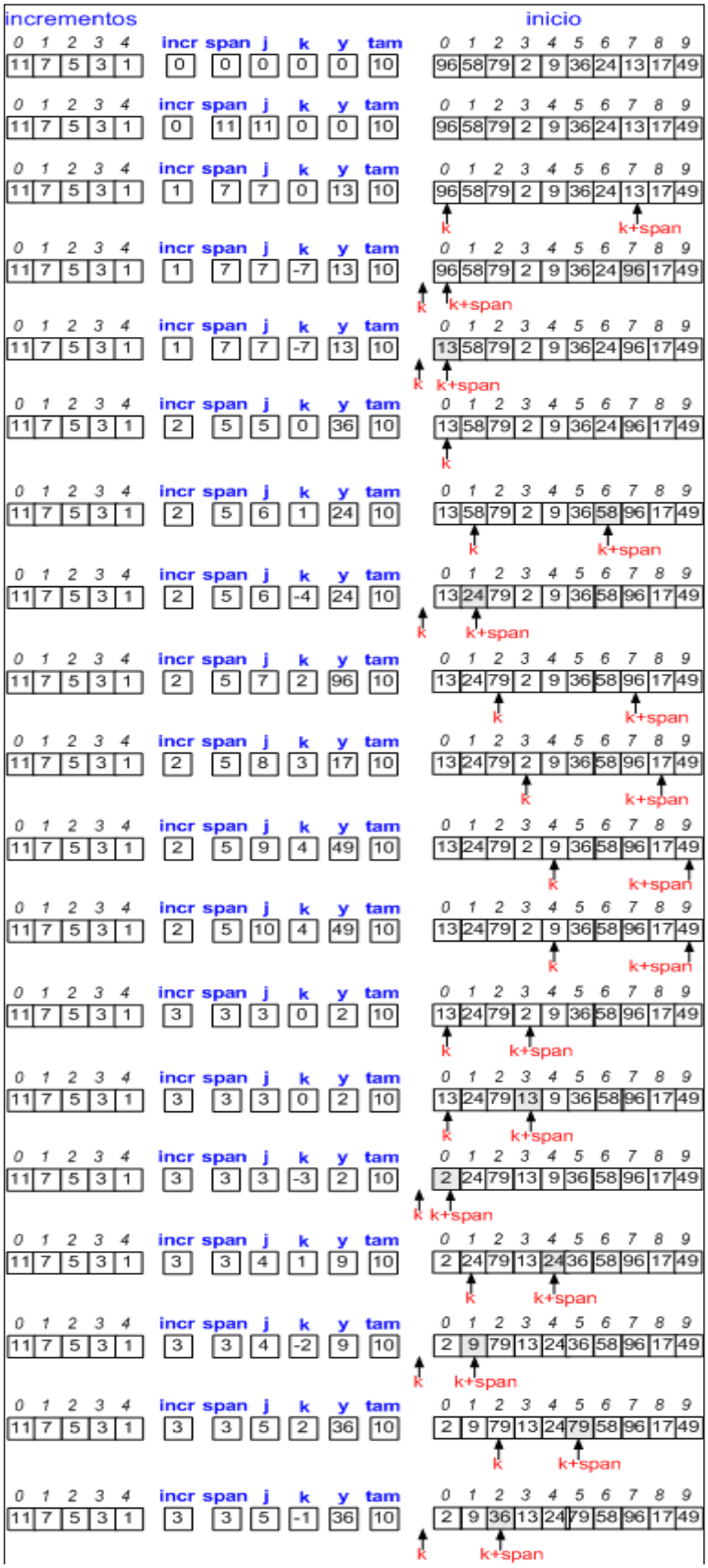


Figura 81: “Análisis del Funcionamiento del Algoritmo ShellSort (Parte 1 de 3)”

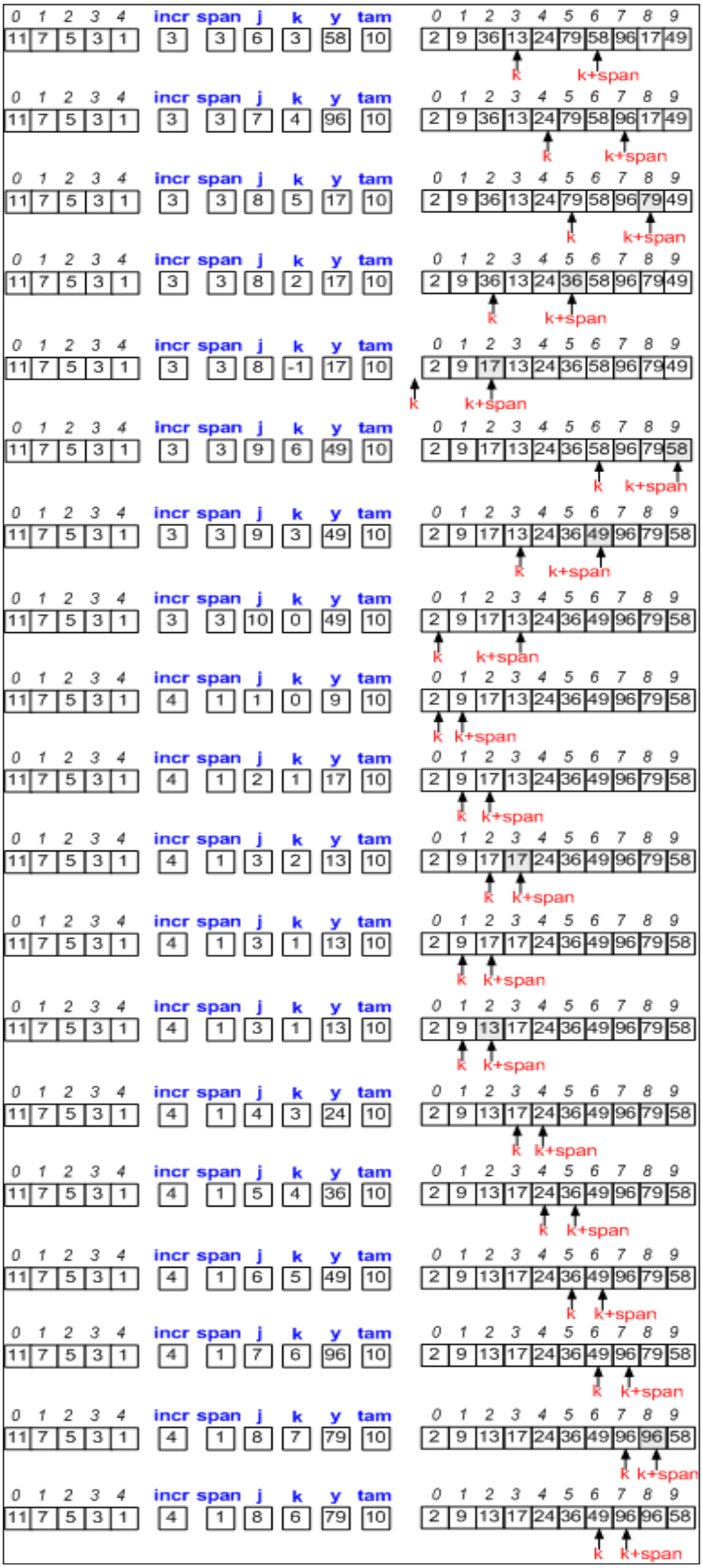


Figura 82: “Análisis del Funcionamiento del Algoritmo ShellSort (Parte 2 de 3)”

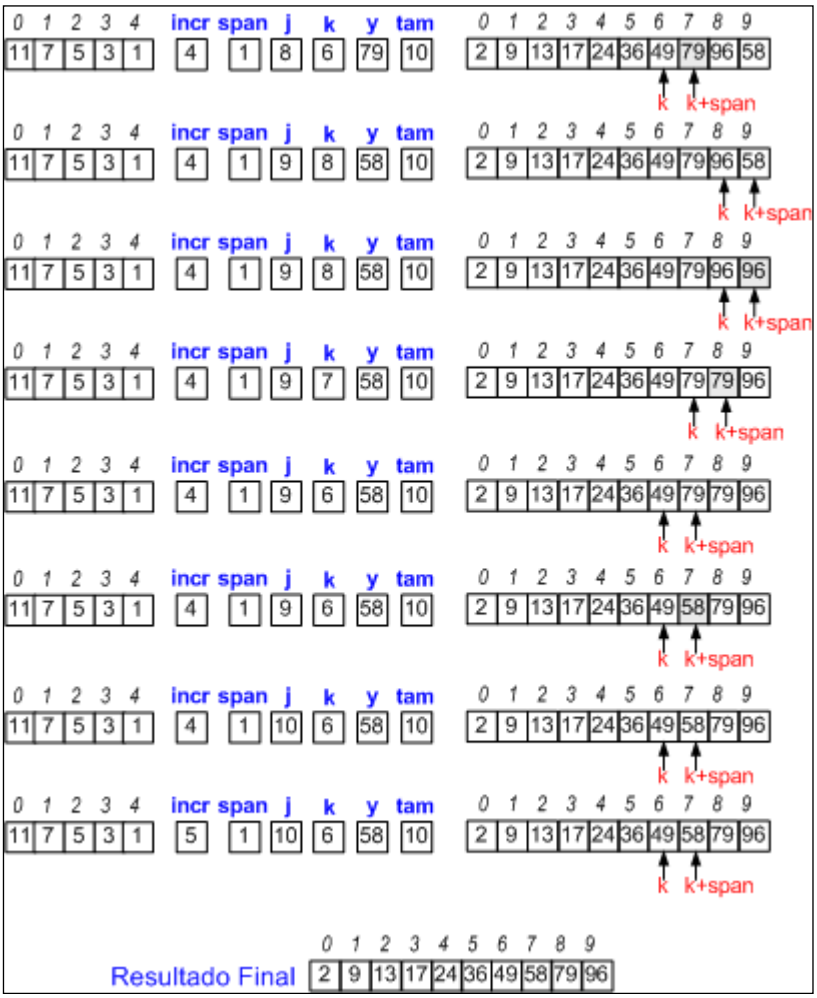


Figura 83: "Análisis del Funcionamiento del Algoritmo ShellSort (Parte 3 de 3)"

Veamos algunas consideraciones a partir del Funcionamiento del Shellsort:

- Depende fuertemente de la secuencia de números **primos** incorporada en el arreglo "incrementos".
- El valor máximo de la secuencia de números primos *debe* ser superior al tamaño del arreglo a ordenar.
- Los números primos del arreglo "incrementos" deben ser ingresados por código fuente, generados al momento del ordenamiento o bien, mediante un proceso Batch, pero de todas formas, se agrega un tiempo adicional al propio del ordenamiento.
- Inicia Comparando los extremos del Arreglo, provocando permutaciones de extremos en caso de ser necesario. Note que el valor de "span" es el de los números primos, por lo que inicia con 11, extremo derecho del arreglo.
- Si el número primo es superior al tamaño del arreglo, no se produce comaración alguna entre los elementos del arreglo. Esto último nos da una pauta de cuál será el máximo número primo que se deberá generar al momento de ordenar un arreglo de tamaño aleatorio o desconocido.

- “k+span” representa el extremo derecho del intervalo de comparación, que decrementa de a “span” magnitud, si se torna negativo el “k”, se incrementa el “j”, acotando a la izquierda el intervalo. Si “k” no es negativo, se compara el valor que se trae desde “k+span” con el nuevo “k”, si es menor, se mueve la posición k a “k+span” y vuelve a decrementarse k, así hasta que no encuentra ninguno menor al que trae desde la posición inicial de “k+span”, a éste último lo almacena en la última posición rotada hacia k+span, previo al decremento. As´, vemos una limpia rotación en Anillo, con el valor original de “k+span” almacenado temporalmente en la variable “y”, hasta que se produzca un nuevo acotamiento del intervalo a través del incremento de “j”.
- Note que el “k+span”, representa el límite derecho inicial, y si encuentra un valor “bajo” al final del arreglo, trata de desplazarlo hacia la izquierda tanto como le sea posible.
- La variable “k” (diferencia entre j y span), indica inicialmente el límite izquierdo de comparación, el que al igual que “k+span”, va corriéndose hacia la izquierda de acuerdo al valor “span”, en el caso de que exista un número mayor en el arreglo, en la posición “k” que el de la posición “k+span”.

Quicksort

El Quicksort es un Algoritmo, desarrollado inicialmente en forma recursiva , en 1960 por C.A.R. Hoare, es uno de los algoritmos más prácticos a la hora de implementar, con muy buena eficiencia. Es uno de los algoritmos más rápidos, en materia de ordenamiento, que se conocen públicamente a la fecha en la materia. Han surgidos numerosas modificaciones al Quicksort original, tratando de hacerlo más eficiente, pero siempre partiendo del quicksort original.

El Quicksort utiliza

$$2 * N * Ln(N)$$

Comparaciones por término medio.

El Quicksort surge bajo la filosofía de “Cercamiento de Frentes y división para la conquista”, ¿Qué es esto?, el algoritmo inicialmente, realiza una pasada cercando simultáneamente el arreglo a ordenar desde izquierda y derecha, efectuando un primer ordenamiento entre extremos. Concluída esta primera etapa, fracciona el lote a ordenar en dos (parte izquierda y derecha), y aplica la misma filosofía a cada una de las partes (Irá cercando la parte izquierda –por ejemplo, luego seguirá con la parte o fracción derecha-simultáneamente desde la izquierda y la derecha, y una vez hecho vuelve a fraccionar en dos la parte); Seguirá Fraccionando hasta que no haya más que fraccionar, allí reconoce el final del ordenamiento de la parte y, puede ocurrir que continue con la parte derecha (si aún no la analizó, repitiendo la secuelas de fraccionamiento y análisis), o bien que finalice el algoritmo, lo cual nos provee el arreglo ordenado. Veamos cómo trabaja lo dicho sobre el cercamiento y fraccionamiento, del arreglo de datos, en el siguiente gráfico:

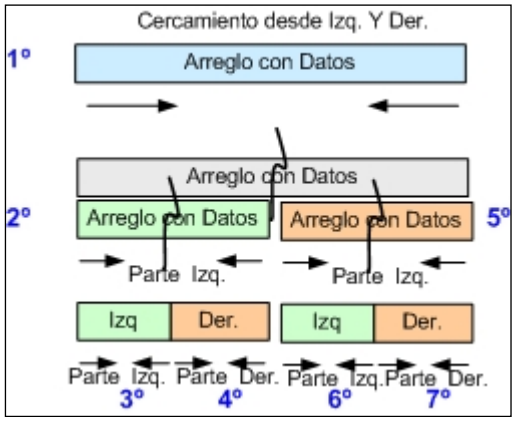


Figura 84: "Conceptualización del Comportamiento del Quicksort"

Suponga un arreglo de datos inicial de un tamaño cualquiera, el quicksort inicialemente cerca el arreglo desde izquierda y derecha, simultáneamente, efectuando en el camino los cambios que pudiese efectuar entre extremos, ¿Qué logra? Balancear el arreglo, llevando los elementos liviandos a la primera sección del arreglo y los elementos tortugas al final, no garantizando aún un orden lineal.

Culminada la primera etapa (señalada en la figura por el "1º"), fracciona el arreglo, generándose la parte izquierda (señalada con verde en la figura 84) y la parte derecha (señalada con naranja en la misma figura). Una vez que efectúa la división o particionamiento del arreglo, como prefiera llamarle, toma la parte izquierda del mismo, y efectúa el cercamiento, al igual que lo hizo en 1º, pero sobre la sección partida izquierda, ¿Objeto de ello? Nuevamente balancear los elementos de la parte izquierda, llevando los elementos más pesados (de los inicialmente livianos), hacia el final y, los livianos al inicio.

Culminado la etapa de cercamiento 2º, fracciona nuevamente generando de la subparte izquierda inicial (de la etapa 2º), una subparte izquierda y derecha, a las cuales les aplicará el mismo proceder. Inicialmente toma la parte izquierda (3º), si no puede seguir fraccionando al terminar con el cercamiento de dicha parte, continua con la parte derecha de su parte padre (El padre es parte izquierda de Etapa 2º , y la parte derecha de dicho padre, es el indicado por 4º). Allí efectuará el cercamiento y al finalizar el mismo, tratará de fraccionar, si no pudiese, continuará con la misma operatoria en la parte derecha de su abuelo (El abuelo es el arreglo de la Etapa 1º, y la parte derecha es 5º).

Al finalizar de Cercar y de fraccionar en subpartes en el lado derecho (5º), análogamente a lo que se efectuó en la parte izquierda iniciada en 2º, culminará el algoritmo, dando como resultado un arreglo ordenado.

Veamos ahora, una implementación básica del algoritmo quicksort:

Archivo: **quicksor.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void quicksort(int[],int,int,int);
5
6       void main(void)
7       {
```

```
8       int aordenar[10]={96,58,79,2,9,36,24,13,17,49};
9
10      clrscr();
11      cout<<"Arreglo Inicial\n";
12      for(int i=0;i<10;i++)
13        cout<<aordenar[i]<<"-";
14
15      quicksort(aordenar,10-1,0,10-1);
16      cout<<"\n\nArreglo Ordenado\n";
17      for(i=0;i<10;i++)
18        cout<<aordenar[i]<<"-";
19     }
20
21     void quicksort(int a[],int tam,int izq,int der)
22     {//Quicksort Recursivo
23            int v;
24            int i,j;
25
26            if(der>izq)
27            {
28                v=a[der];
29                i=izq-1;
30                j=der;
31
32                for(;;)
33                {
34                    ++i;
35                    while(i<=tam && a[i]<v)++i;
36                    --j;
37                    while(j>=0 && a[j]>v)--j;
38                    if(i>=j)break;
39                    int aux=a[i];
40                    a[i]=a[j];
41                    a[j]=aux;
42                }
43
44                int aux2=a[i];
45                a[i]=a[der];
46                a[der]=aux2;
47
48                quicksort(a,tam,izq,i-1);
49                quicksort(a,tam,i+1,der);
50            }
51     }
52
```

Note que para poseer un Algoritmo potente no hace falta gran cantidad de líneas, sino una intrincada red de lógica algorítmica.

A continuación, y de la misma forma que lo hemos realizado con el Shellsort, enviaremos a nuestro algoritmo al campo de batalla y, veremos en acción su lógica comportamental.

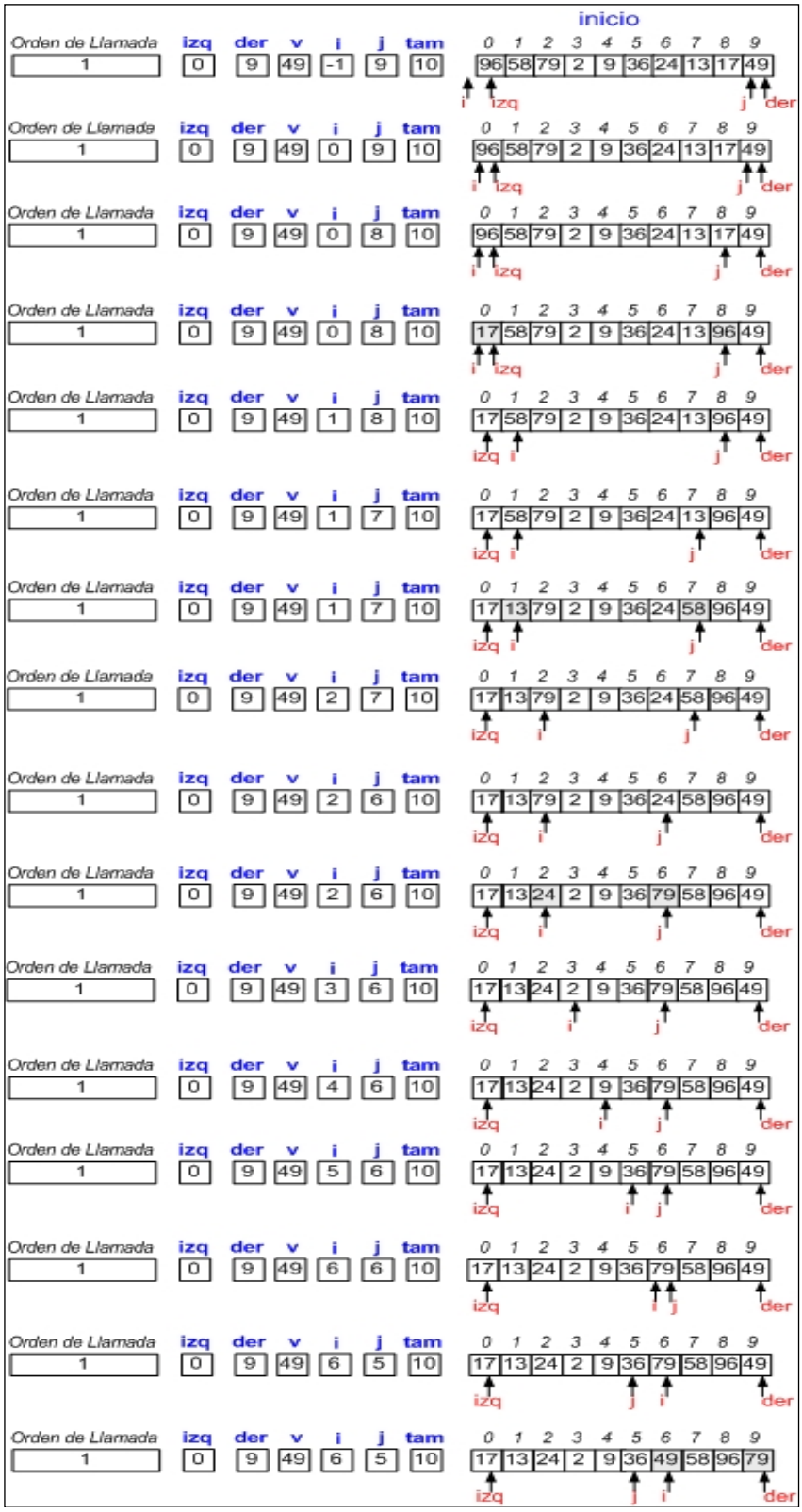


Figura 85: “Análisis del Funcionamiento del Algoritmo QuickSort (Parte 1 de 4)”

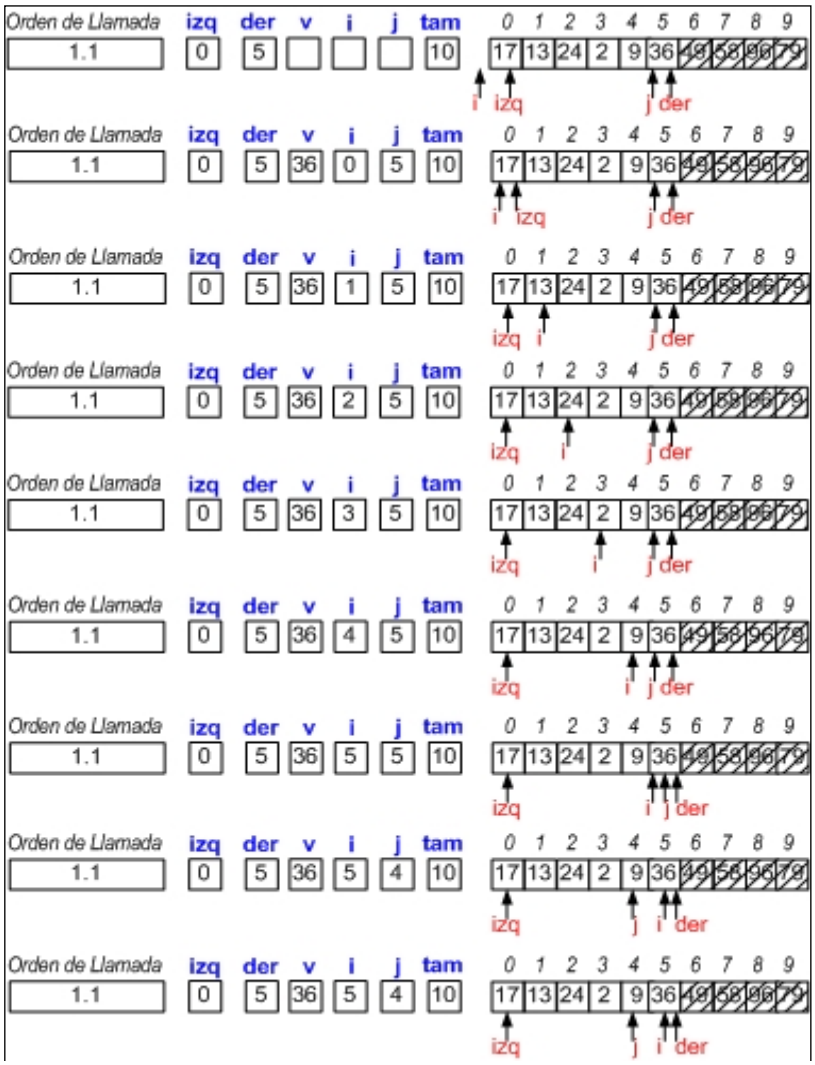


Figura 86: “Análisis del Funcionamiento del Algoritmo QuickSort (Parte 2 de 4)”

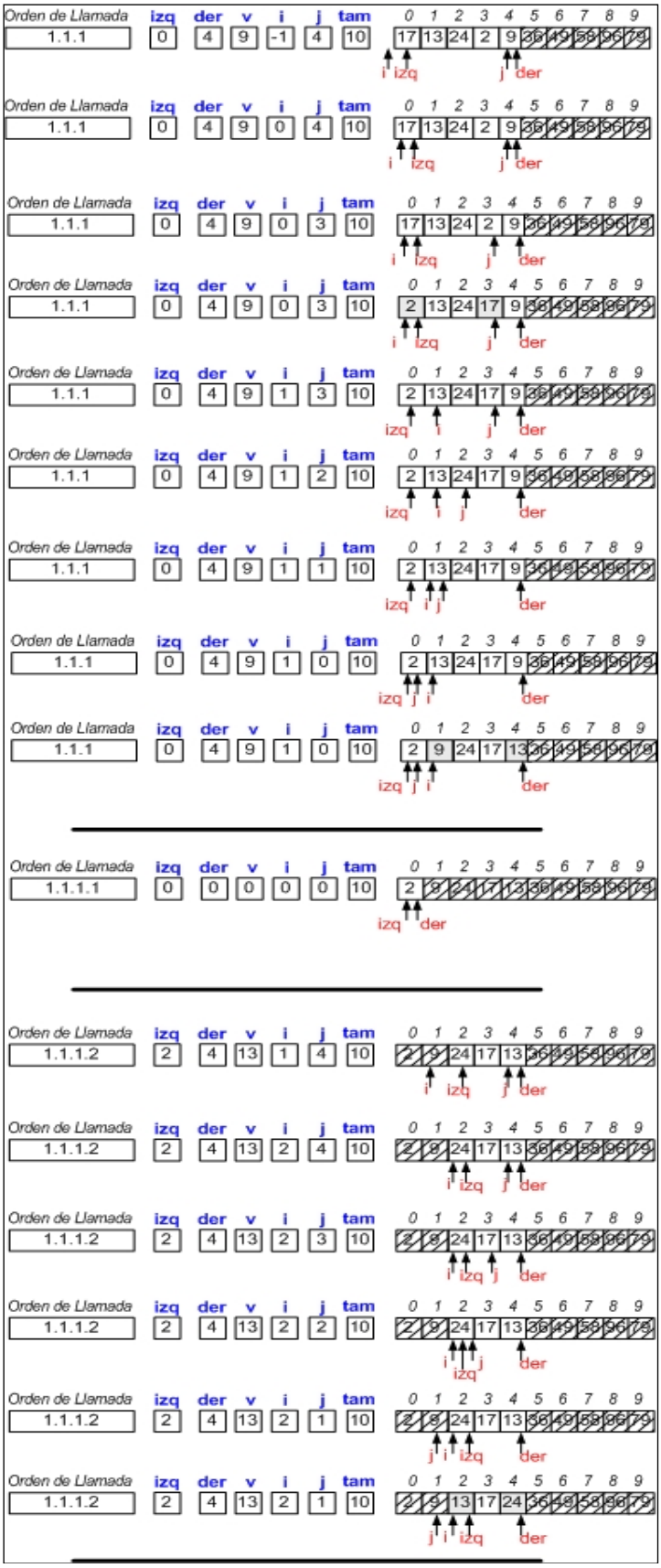


Figura 87: “Análisis del Funcionamiento del Algoritmo QuickSort (Parte 3 de 4)”

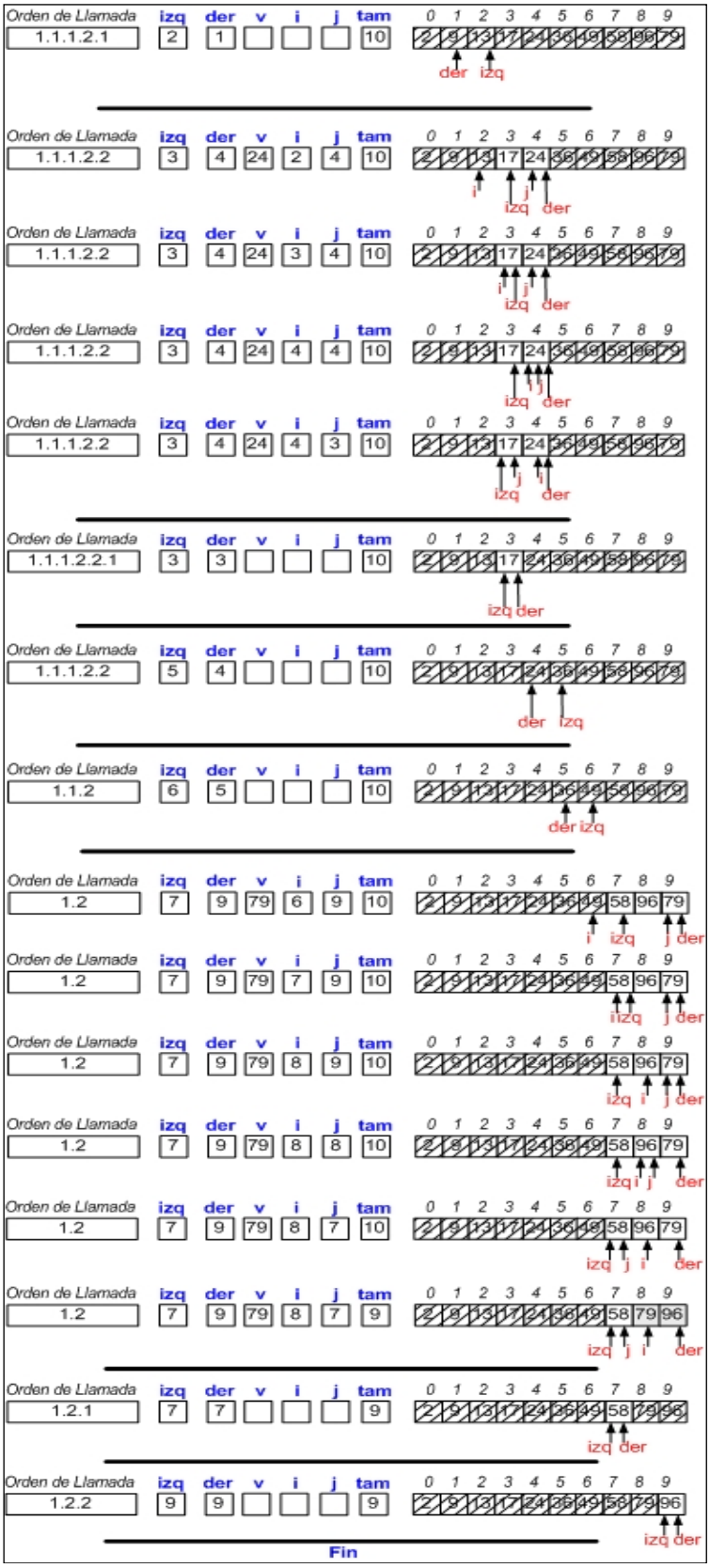


Figura 88: “Análisis del Funcionamiento del Algoritmo QuickSort (Parte 4 de 4)”

El Análisis del funcionamiento, para facilidad del lector, está hecho siguiendo el orden de ejecución recursivo del algritmo. Para mayor claridad y ubicación del lector, a continuación, la figura 90, muestra el orden las llamadas recursivas y cómo se va cercando y particionando el arreglo.

Previamente, analizaremos la simbología que se empleará en el gráfico de llamadas recursivas, para un mejor entendimiento:

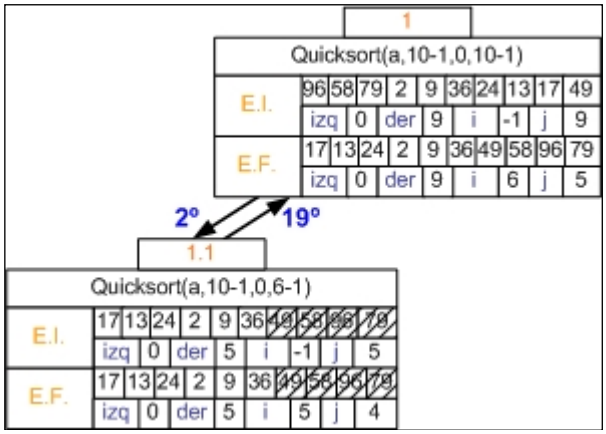


Figura 89: “Análisis de la Simbología Empleada en el Gráfico de Llamadas Recursivas”

El número superior de cada recuadro, contiene un número separado por puntos (salvo la raíz), que indica desde donde se invocó el algoritmo recursivamente y qué sección se encuentre ejecutando del arreglo de datos. En nuestro ejemplo es “1.1”, el mismo indica que fue invocado desde la primera ocurrencia del algoritmo quicksort, y está trabajando en la sección izquierda del arreglo (dado por el 1 luego del punto, si fuese 1.2, estaría trabajando en la sección derecha del arreglo).

El segundo recuadro, contiene la llamada a función con la cual se invocó a quicksort, y por ende define la parte del arreglo sobre la que trabajará, en nuestro caso es “quicksort(a,10-1,0,6-1)”, lo que indica que se trabajará sobre el arreglo “a”, desde “izq“ (donde izq es 0), hasta “der” (donde der es 6-1=5); en otras palabras la parte del arreglo sobre la que se trabajará con quicksort queda definida como [0;5].

El tercer recuadro, contiene el estado inicial al invocar el algoritmo, donde se muestra el estado de ordenamiento del arreglo “a” (17, 13, 24, 2, 9, 36, 49, 58, 96 y 79), los valores iniciales, de acuerdo a los parámetros de invocación de la función, de “izq” y “der”, que definirán el intervalo del arreglo sobre el cual trabajará el algoritmo; y los valores “i” y “j”, los cuales surgen a partir de “izq” y “der”, pero ed de gran importancia conocerlos, ya que en la reinvocación del algoritmo recursivo definen el nuevo intercalo (son los que fraccionan). Esto último lo veremos así:

1. Vea que en el estado final i=6 y j=5,
2. Al finalizar la ejecución de una de las ramas recursivas del quicksort, y previo a invocarse, la sección “E.F.” o “Estado Final”, muestra la situación de las variables trascendentales y del arreglo,
3. la primera invocación, según la línea 48 del quicksort es: **quicksort(a,tam,izq,i-1)**,

Note como ese sútil “i-1” está acotando al quicksort, a trabajar en la sección izquierda de la parte actual del arreglo. Análogamente con “der”, en la línea 49, **quicksort(a,tam,i+1,der)**, quien acota al quicksort a trabajar en la parte derecha de la parte actual. Una pregunta natural que le puede sugir es ¿y “j”? Justamente “i” y “j”, son los que regulan la actividad de cercamiento, y son ellos los encargados de indicar al algoritmo que ha culminado la rama recursiva y debe particionar o volver hacia su parte padre o abuelo (según sea el caso).

Veamos ahora, como ocurren las llamadas recursivas, varían los estados de variables y arreglo, a lo largo de la ejecución del programa “quicksor.cpp”:

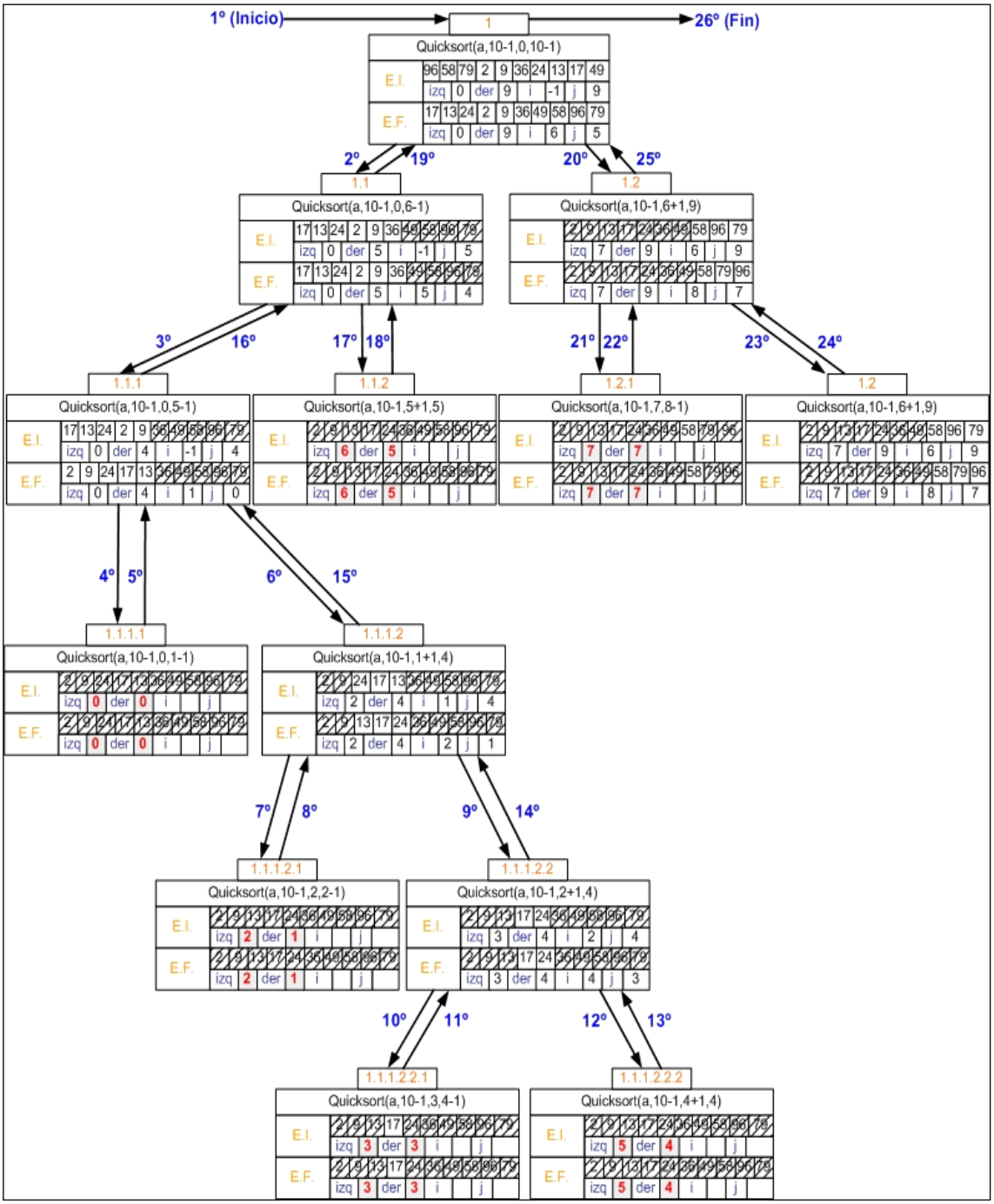


Figura 90: “Análisis de Llamadas Recursivas del Quicksort”

Heapsort

El Ordenamiento por "heapsort", también conocido como ordenamiento por montículo o apilamiento, se caracteriza por ser un algoritmo de dos fases; en la primera fase realiza un ordenamiento por grupo descendente del tamaño del arreglo, sobre el mismo, sin emplear recursos adicionales a los reservados. En la segunda fase, decapita la cabeza del arbol inicial (grupos de ordenamientos descendentes), y reagrupa el árbol encontrando una nueva cabeza de árbol. Esto último, lo realizará tantas veces como sea necesario hasta quedar solo un elemento en el árbol, decapitándolo o extrayéndolo, de forma que quede el arreglo completamente ordenado. Cada vez que se decapita un elemento del árbol, no se pierde, sino que se ubica dentro del mismo arreglo en su posición definitiva de orden, y deja de considerarse para los intercambios de elementos.

Note que se menciona "Decapitación", en otras palabras, se trunca siempre la cabeza del árbol, y es ésta quien pasa a ubicarse en la posición definitiva, ¿Casualidad? No, en absoluto, veremos en los apartados posteriores el motivo de dicha medida.

A continuación, se enunca el algoritmo básico del Heapsort codificado en C/C++:

Archivo: **heapsort.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       void heapsort(int[],int);
5
6       void main(void)
7       {
8        int aordenar[10]={96,58,79,2,9,36,24,13,17,49};
9
10       clrscr();
11       cout<<"Arreglo Inicial\n";
12       for(int i=0;i<10;i++)
13         cout<<aordenar[i]<<"-";
14
15       heapsort(aordenar,10);
16       cout<<"\n\nArreglo Ordenado\n";
17       for(i=0;i<10;i++)
18         cout<<aordenar[i]<<"-";
19      }
20
21      void heapsort(int v[],int tam)
22      {
23        int i,e,s,f,valori;
24
25        //CREA EL GRUPO INICIAL
26        for(i=1;i<tam;i++)
27        {
```

```
28          e=v[i];
29          s=i;
30          f=(s-1)/2;
31
32          while(s>0 && v[f]<e)
33          {
34           v[s]=v[f];
35           s=f;
36           f=(s-1)/2;
37          }
38
39          v[s]=e;
40        }
41        cout<<"\n\nGrupo Inicial\n";
42        for(i=0;i<tam;i++)
43             cout<<v[i]<<"-";
44        //Extrae la ra¡z y reordena el grupo
45        for(i=(tam-1);i>0;i--)
46        {
47         valori=v[i];
48         v[i]=v[0];
49         f=0;
50
51         if(i==1) s=-1;
52          else s=1;
53
54         if(i>2 &&  v[2]>v[1]) s=2;
55
56         while(s>=0 && valori<v[s])
57         {
58          v[f]=v[s];
59          f=s;
60          s=(2*f)+1;
61          if(((s+1)<(i-1)) && (v[s]<v[s+1])) s=s+1;
62          if(s>(i-1)) s=-1;
63         }
64         v[f]=valori;
65        }
66      }
67
```

Antes de ingresar en la explicación del funcionamiento del 'heapsort', deberemos analizar una serie de definiciones previas para comprender la lógica del algoritmo.

Se entiende por Arbol Binario *casi completo* a un árbol binario de altura 'h':

1. Donde todo nodo ubicado en un nivel menor que (h-1), tiene dos hijos
2. Las hojas (nodos del útlimo nivel), deben llenar dicho nivel, de izquierda a derecha sin dejar huecos o saltos.

Veamos la interpretación gráfica de estos requerimientos:

"Un Árbol Binario no es casi completo, si algún nodo del nivel menor que (h-1) no posee dos hijos"

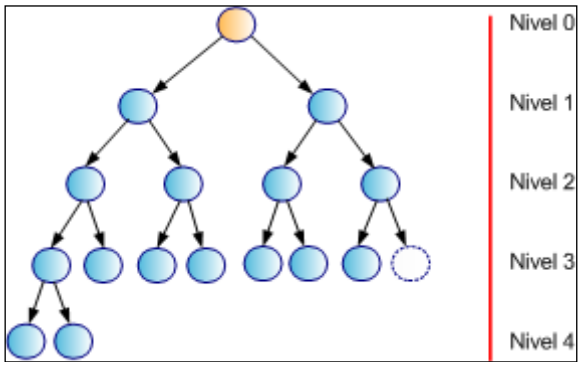


Figura 91: "Interpretación Gráfica de la Regla 1º de Árboles Binarios Casi Completos"

Note que en la figura 91, disponemos de un árbol binario de nivel cuatro, en el cual existe un nodo de nivel menor que (h-1), precisamente un nodo del nivel 2, que no posee dos hijos, sino tan solo uno (el nodo ausente se encuentra con el contorno punteado).

"Un Árbol Binario no es casi completo, si sus hojas no son llenadas en orden continuo de izquierda a derecha (poseen 'saltos' entre ellas)"

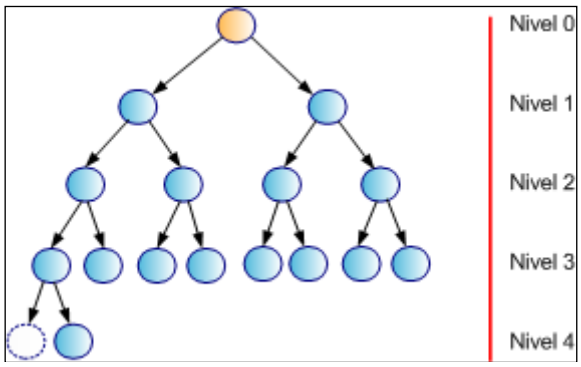


Figura 92: "Interpretación Gráfica de la Regla 2º de Árboles Binarios Casi Completos"

Note que en la figura 92, precisamente en el cuarto nivel, el orden de llenado del árbol binario de izquierda a derecha, no es continuo, por lo que existe un 'salto' en el primer nodo de izquierda a derecha del nivel. El Salto, hoja ausente, se representa con un contorno punteado, y claramente antecede a la siguiente hoja del nivel.

Ahora bien, veamos cómo quedaría la representación gráfica de un árbol binario casi completo:

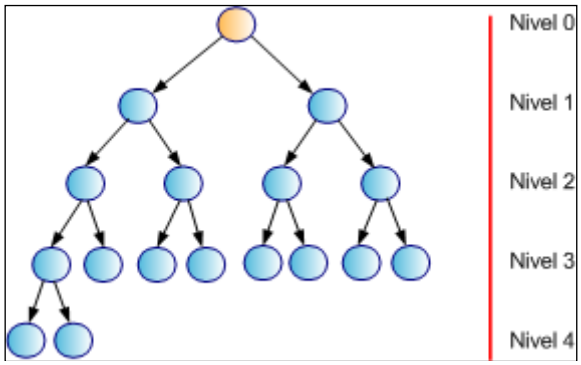


Figura 93: "Interpretación Gráfica del Concepto de Árbol Binario Casi Completo"

En un árbol Binario de Nivel 4, como el de la figura 93, donde todos los nodos situados en los niveles inferiores a 3 (h-1) poseen dos hijos, y sus hojas son continuas de izquierda a derecha en llenado, sin implicar necesariamente la completitud del nivel, se dice que es un árbol binario casi completo.

"Un Árbol Binario Casi completo se 'N' nodos, se puede implementar usando un vector de 'N' componentes". En otras palabras, si podemos probar empleando las reglas anteriores, que un árbol binario es casi completo, entonces se podrá implementar en un vector con igual longitud que la cantidad de nodos del árbol.

Cómo se podría incorporar un árbol binario casi completo en un arreglo es lo que repasaremos brevemente a continuación:

- La Raíz del árbol debe situarse en la posición '0' del arreglo o vector,
- El Hijo Izquierdo de v[i] se guarda en v[(i*2)+1],
- El Hijo Derecho de v[i] se guarda en v[(i*2)+2],

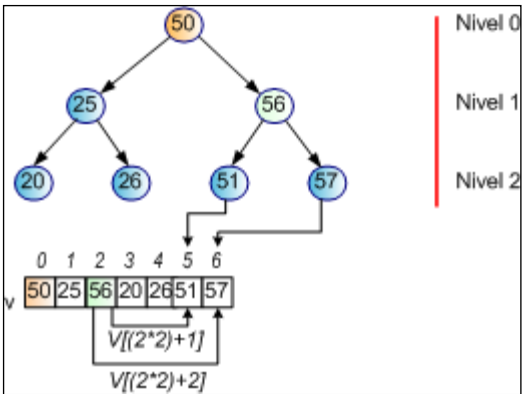


Figura 94: "Interpretación Gráfica del Concepto de Árbol Binario Casi Completo Implementado en un Arreglo de Tamaño 'N'"

El Árbol Binario Casi Completo de la figura 94, posee un total de siete nodos, lo cual determina el tamaño del vector que almacenará al mismo. Note que el hijo izquierdo del nodo 56, se puede rastrear perfectamente a través de v[(2*2)+1], mientras que el nodo derecho del mismos nodo, análogamente se rastrea a través de v[(2+2)+1], donde la posición 'i' del nodo con el valor '56' es '2'.

El último concepto que se deberá revisar, previo al análisis del 'heapsort', es lo que se entiende poor 'Grupo de Ordenamiento Descendente'.

"Un Grupo de Ordenamiento Descendente, es un Árbol Binario Casi Completo en el cua,l el nodo padre posee un valor superior al de sus dos hijos, sin importar la relación que exista entre sus dos hijos"

Simplemente la condición adicional que agrega el Grupo de Ordenamiento Descendente al Árbol Binario Casi completo, es que su padre posea un valor superior al de sus hijos, pero no le interesa el vínculo que exista entre ellos, en otras palabras, no le interesa si el hijo izquierdo es superior al hijo derecho o viceversa.

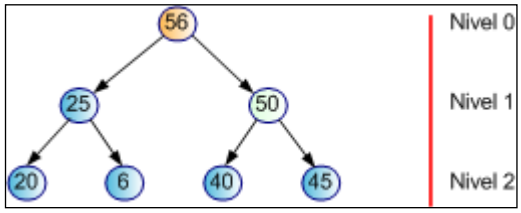


Figura 95: "Interpretación Gráfica del Concepto de Grupo de Ordenamiento Descendente"

El árbol de la figura 95, es un 'Grupo de Ordenamiento Descendente', note que todos los nodos que tienen hijos, poseen un valor superior a estos. ¿Dejaría de ser un Grupo de Ordenamiento Descendente si en lugar de la hoja con el 40 estuviese el 45 y viceversa?, En otas palabras, ¿El árbol Binario Casi Completo a continuación es un Grupo de Ordenamiento Descendente?

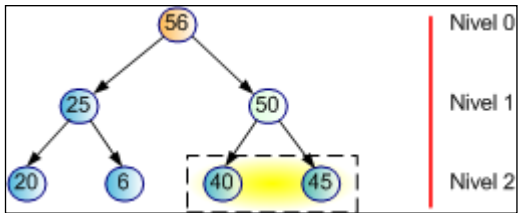


Figura 96: "Interpretación Gráfica del Concepto de Grupo de Ordenamiento Descendente, Posiciones Alternas de los Hijos"

El árbol de la figura 96, es un 'Grupo de Ordenamiento Descendente', ya que el orden o vínculo entre los hijos de un mismo padre (Hermanos) no repercute en la definición, recuerde: ¡¡¡La Definición exige que los nodos padres solamente, posean un valor superior al de sus nodos hijos, independientemente de la relación que exista entre los mismos!!!

Existen algunas propiedades muy valiosas que se desprenden del Análisis de un Grupo de Ordenamiento Descendente, como ser:

1. "El Mayor Valor está en la raíz": note que el valor '56' correspondiente al nodo raíz es el máximo del grupo.
2. "Todo Camino de la Raíz a una hoja es una lista ordenada descendentemente": ya de por sí una propiedad sumamente valiosa, note que siqueremos llegar a la hoja '45', el camino a recorrer será '56→50→45'.

Ahora se está en Condiciones de analizar la lógica del 'HeapSort'. Como se mencionó inicialmente, el Heapsort, a partir de un arreglo cualquiera de tamaño 'tam', genera un grupo inicial, que es ni más ni menos que un Grupo de Ordenamiento Descendente.

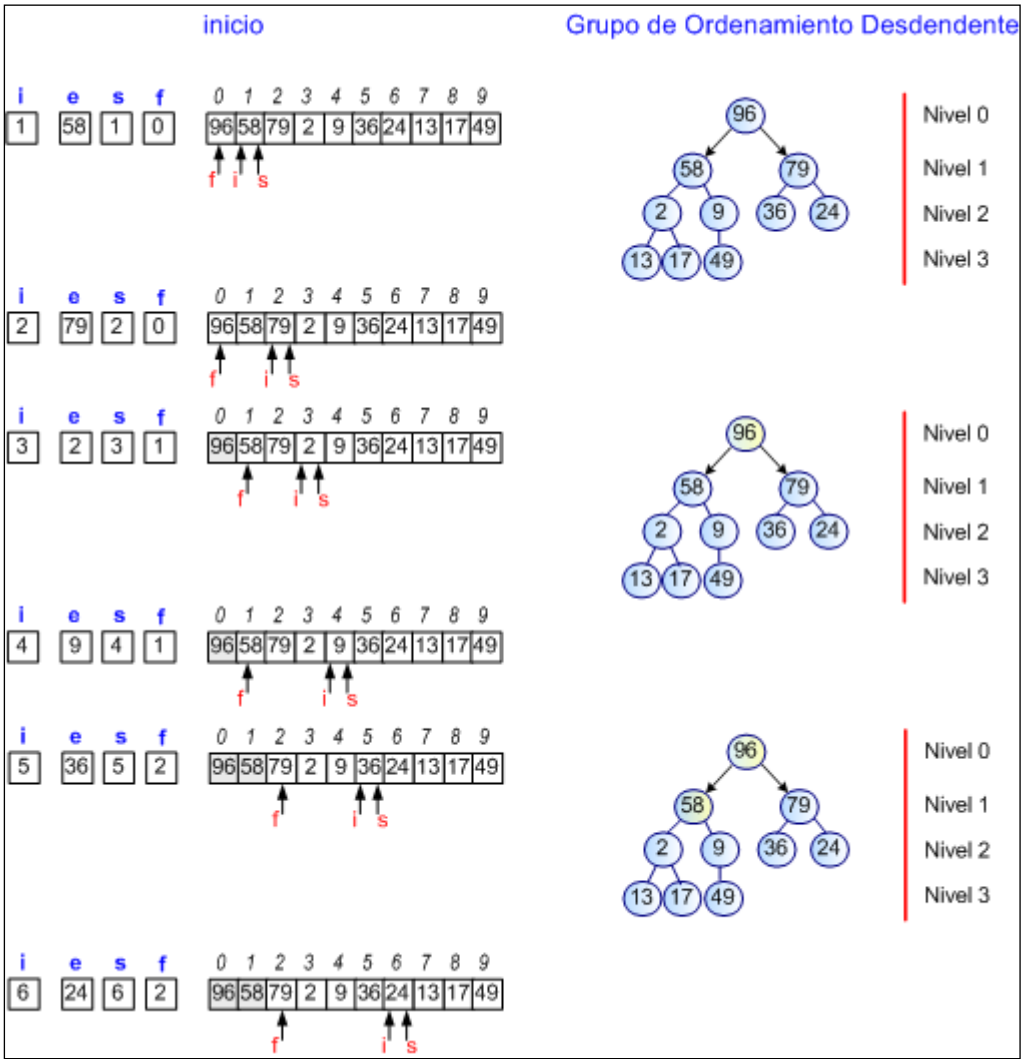


Figura 97: "Fase 1 del Heapsort: Generación del Grupo Inicial (Grupo de Ordenamiento Descendente)"

En la figura 97 se muestran los primeros pasos de la fase 1º del algoritmo heapsort. Note que 'f' apunta siempre a nodos que tienen hijos, su función es comparar si el padre es mayor a los hijos, ya analizaremos a continuación qué ocurre cuando esto sucede. El puntero 'i' recorre uno por uno la totalidad de los nodos, mientras que 's' es quien ubica a los hijos del correspondiente padre apuntado por 'f'.

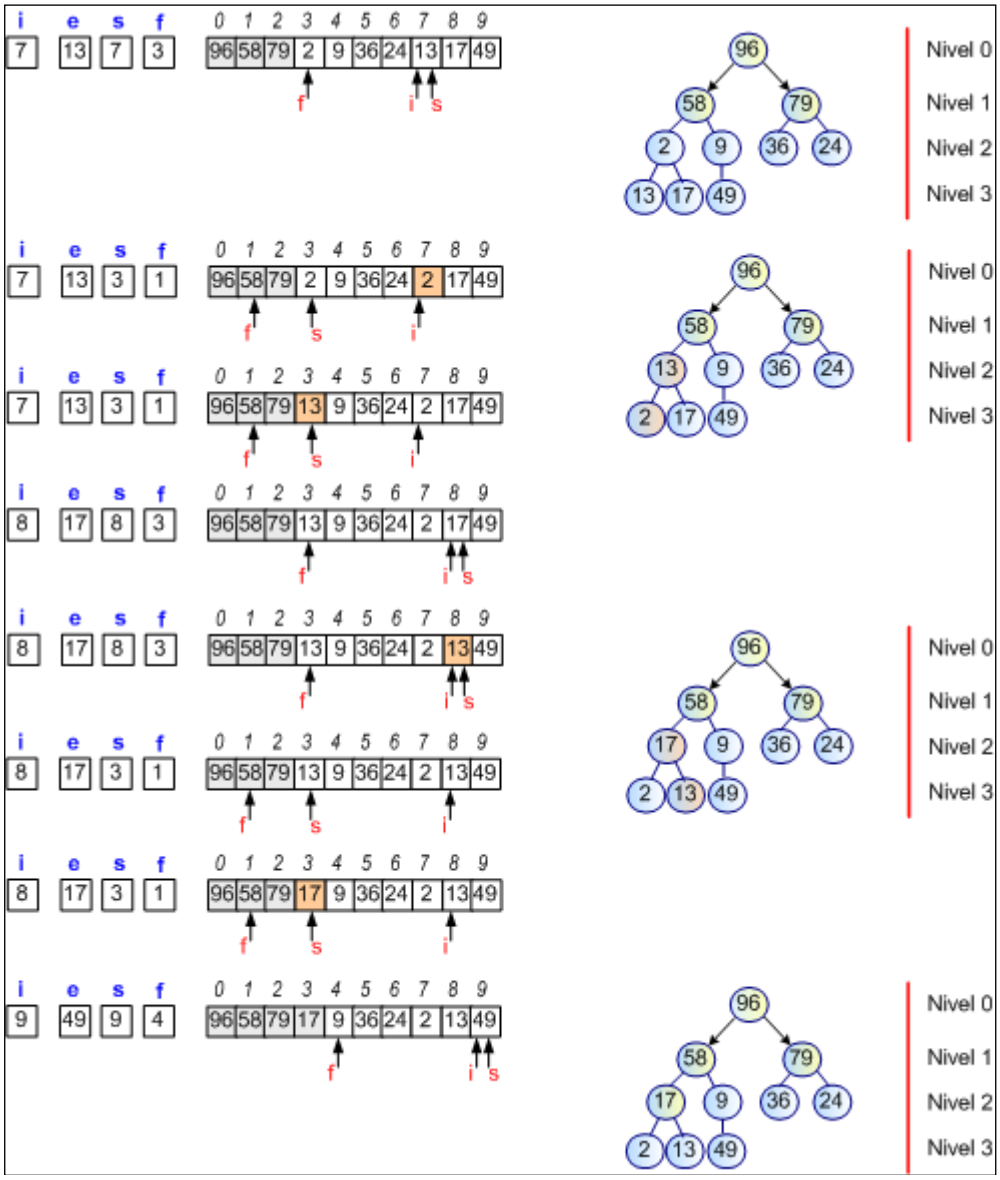


Figura 98: “Fase 1 del Heapsort: Generación del Grupo Inicial (Grupo de Ordenamiento Descendente)”

En la figura 98 se puede apreciar la lógica del heapsort al encontrar un hijo con un valor superior al del padre, vea puntualmente el primer caso, donde el ‘13’ está debajo del ‘2’, siendo éste último su padre. Al encontrar esta stuación, el ‘heapsort’ intercambia las posiciones del 2 con el 13, al puntero ‘s’ que apuntaba al actual hijo –previo al intercambio, el ‘13’- se hace que direccione a la posición del actual padre como si fuese un hijo (se refiere a la posición antigua del 2), mientras que el puntero ‘f’ que direccionaba al antiguo padre (el ‘2’), ahora direccionará al padre original de éste último, el ‘58’ ¿motivo? Averiguar si su nieto original (el ‘13’) es superior a él, de serlo repite el mismo procedimiento para ponerlo en su lugar, de no serlo (como en este caso), continúa analizando los demás hijos del ahora nuevo padre ‘13’, y de no tener más hijos, continúa con otros padres.

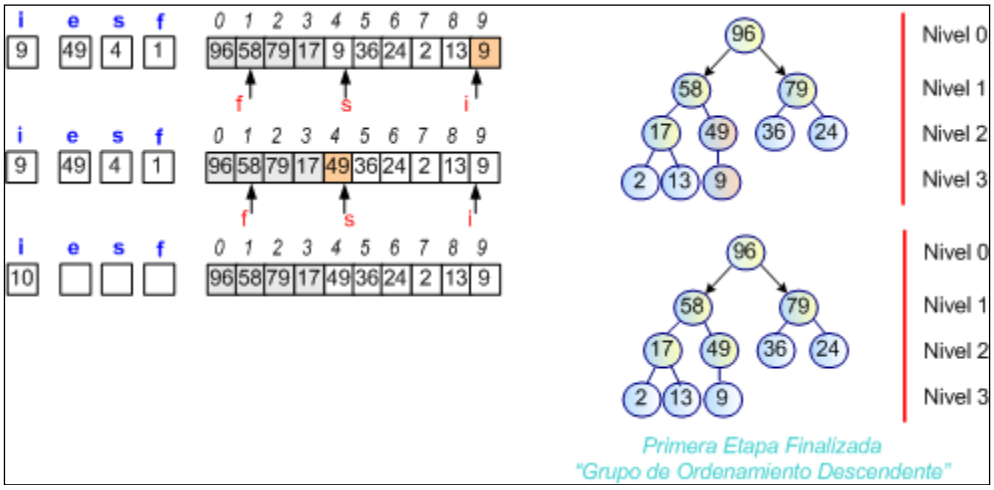


Figura 99: "Fase 1 del Heapsort: Generación del Grupo Inicial (Grupo de Ordenamiento Descendente)"

En la figura 99 se puede apreciar el estado final de la primera fase, como resultado de la operatoria se obtiene un grupo de ordenamiento descendente. Éste último, haciendo uso de las reglas y propiedades vistas anteriormente, se representa en un arreglo de tamaño 10 (Lo cual es equivalente a la cantidad de nodos en el árbol), y el Árbol Binario Representado es un Árbol Binario del Tipo "Casi Completo", lo que implica que todos sus nodos correspondientes a niveles inferiores al (h-1=3-1=2), tienen 2 hijos y todas sus hojas completan el último nivel de izquierda a derecha en forma contínua.

En la Segunda Fase, El heapsort por cada iteración, removerá la cabeza del Árbol Binario Descendientemente Ordenado, la enviará a la última posición del arreglo, almacenando previamente el valor de ésta última posición en una variable auxiliar (Recuerde que la última posición del arreglo en un Árbol Binario Descendientemente Ordenado, representa la última hoja de izquierda a derecha). Luego de decapitar el árbol, analiza cual de sus hijos es el mayor, transporta el mayor de sus hijos a su posición actual (recuerde que la información de la cabeza del árbol no se pierde ya que se transportó a la última hoja del árbol).

Una vez que decapitó el árbol, se traslada al hijo mayor a la posición del padre, en ese momento, existe dos veces el valor del mayor hijo en el árbol, el heapsort toma el valor correspondiente al mayor nivel (es decir el más cercano a las hojas), y pregunta si existe algún hijo de este nodo que sea mayor al valor almacenado en la variable auxiliar, de ser verdadera la respuesta transporta este hijo a la posición más profunda del hijo duplicado y reitera la pregunta para los hijos de éste último, de esta forma sucesivamente hasta que no exista un hijo mayor al padre, en ese momento, se coloca en el último hijo duplicado el valor de la variable auxiliar. Si por el contrario, no existen hijos mayores al padre, coloca en esa posición, el valor de la variable auxiliar y pasa a decapitar nuevamente el árbol reiterándose todo el proceso.

El Heapsort relatado es un poco tosco de comprender, es mucho más sencillo analizado gráficamente o a través del Análisis de Modificación de Variables, veamos éste último en primera medida.

La Segunda Fase consiste en ir extrayendo la raíz y reordenar el grupo, veamos como trabaja su lógica directamente sobre una tabla de variables del algoritmo

Variables				Arreglo									
i	valori	f	s	0	1	2	3	4	5	6	7	8	9
9				96	58	79	17	49	36	24	2	13	9
9	9	0	2	96	58	79	17	49	36	24	2	13	**96**
	9	2	5	**79**	58	79	17	49	36	24	2	13	96
	9	5	-1	79	58	**36**	17	49	**9**	24	2	13	96
8	13	0	1	79	58	36	17	49	9	24	2	**79**	96
		1	3	**58**	58	36	17	49	9	24	2	79	96
			4	58	**49**	36	17	49	9	24	2	79	96
		4	9	58	49	36	17	49	9	24	2	79	96
			-1	58	49	36	17	**13**	9	24	2	79	96
7	2	0	1	58	49	36	17	13	9	24	**58**	79	96
		1	3	**49**	49	36	17	13	9	24	58	79	96
		3	-1	49	**17**	36	17	13	9	24	58	79	96
				49	17	36	**2**	13	9	24	58	79	96
6	24	0	1	49	17	36	2	13	9	**49**	58	79	96
			2	49	17	36	2	13	9	49	58	79	96
		2	5	**36**	17	36	2	13	9	49	58	79	96
				36	17	**24**	2	13	9	49	58	79	96
5	9	0	1	36	17	24	2	13	36	49	58	79	96
			2	36	17	24	2	13	36	49	58	79	96
		2	5	**24**	17	24	2	13	36	49	58	79	96
			-1	24	17	**9**	2	13	36	49	58	79	96
4	13	0	1	24	17	9	2	**24**	36	49	58	79	96
		1	3	**17**	17	9	2	24	36	49	58	79	96
				17	**13**	9	2	24	36	49	58	79	96
3	2	0	1	17	13	9	**17**	24	36	49	58	79	96
		1	3	**13**	13	9	17	24	36	49	58	79	96
			-1	13	**2**	9	17	24	36	49	58	79	96
2	9	0	1	13	2	**13**	17	24	36	49	58	79	96
				9	2	13	17	24	36	49	58	79	96
1	2	0	-1	**2**	**9**	13	17	24	36	49	58	79	96
0				2	9	13	17	24	36	49	58	79	96

Figura 100: “Fase 2 del Heapsort: Decapitación y Reordenamiento del Grupo Descendentemente Ordenado”

Veamos a continuación el comportamiento de la segunda fase del algoritmo heapsort gráficamente:

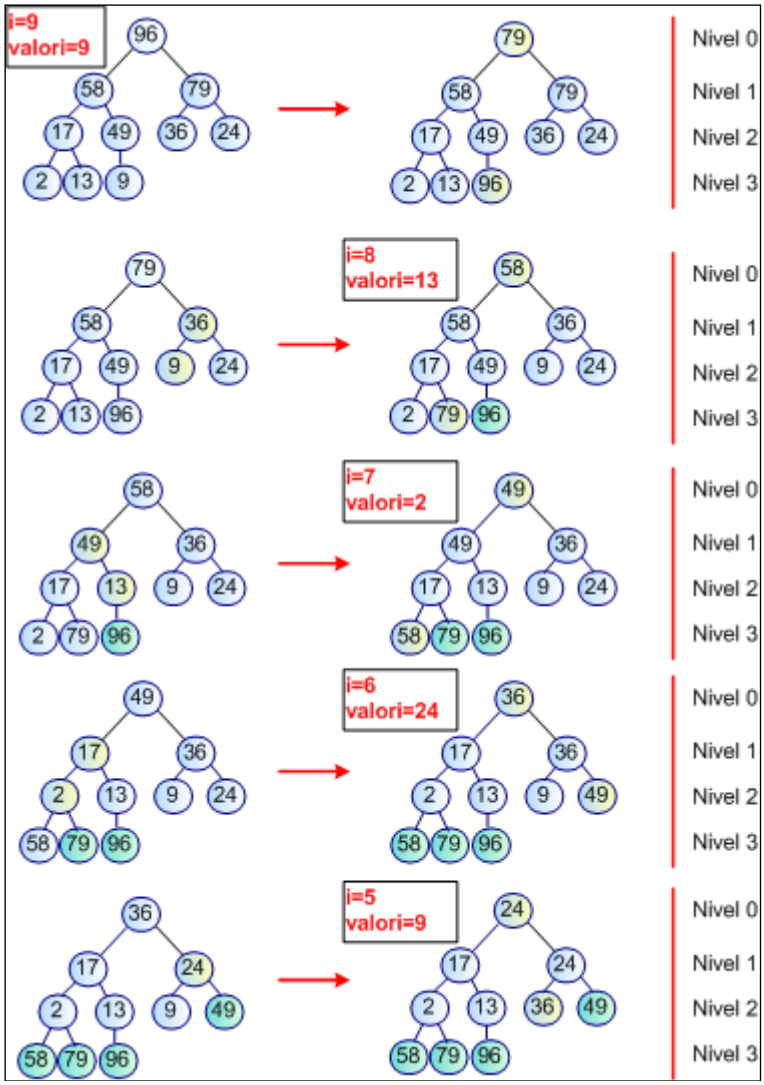


Figura 100: "Fase 2 del Heapsort: Decapitación y Reordenamiento del Grupo Descendentemente Ordenado"

Al Iniciar la segunda fase, como se mencionó, decapita el árbol, envando la raíz en la posición de la última hoja (el 96), luego traslada a la posición de la ra´z al mayor de sus hijos (el 79), el valor de la última hoja está temporalmente almacenado en 'valori' (valori tiene almacenado el 9 por el momento). Una vez trasladado el mayor de sus hijos (el 79), tenemos el valor del mismo duplicado en distintos niveles del árbol (nivel 0 y 1), se toma el de mayor profundidad (el nivel 1), y se vuelve a preguntar si tengo hijos, ¿Cuál es el más grande de mis hijos?, el resultado a esta pregunta es el '36', por lo que se traslada a la posición de su padre (el 79 de mayor profundidad). Luego ser trasladado el 36, tendremos dos veces nuevamente éste valor en el árbolpero en diferentes niveles, nuevamente se toma el de mayor profundidad y se pregunta, si tiene hijos ¿Cuál es el mayor?, como no tiene hijos, directamente en donde figura el último valor repetido de mayor profundidad se coloca el valor almacenado en 'valori' (el 9). Verá que no aparece en la figura 100 el 36 repetido, es por motivos de simplificación delos pasos y mayor claridad en la gráfica.

Una vez que culminó esta operatoria sigue la iteracón, ahora la nueva raíz o cabeza del árbol es el 79, entonces se decapita y se ejecuta el mismo procedimiento

descrito en el párrafo anterior, se ejecutará hasta que se haya logrado decapitar (tam-1) veces el árbol.

Veamos en la gráfica como continúa su comportamiento:

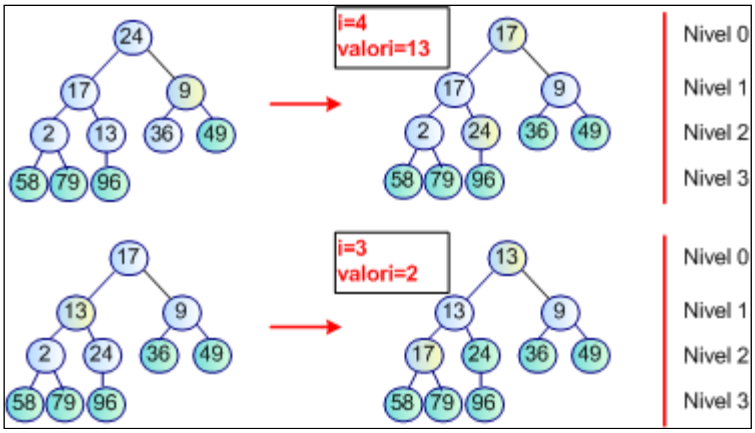


Figura 101: "Fase 2 del Heapsort: Decapitación y Reordenamiento del Grupo Descendentemente Ordenado"

Note que una vez decapitada la raíz o cabeza del árbol, y trasladada a su posición definitiva, no se vuelve a tocar la misma, y de hecho el heapsort, va acotando el intervalo de análisis, por lo que dicha sección, representada en las figuras 100, 101 y 102 con color verde, no hacen másque identificar las posiciones definitivamente ordenada del arreglo.

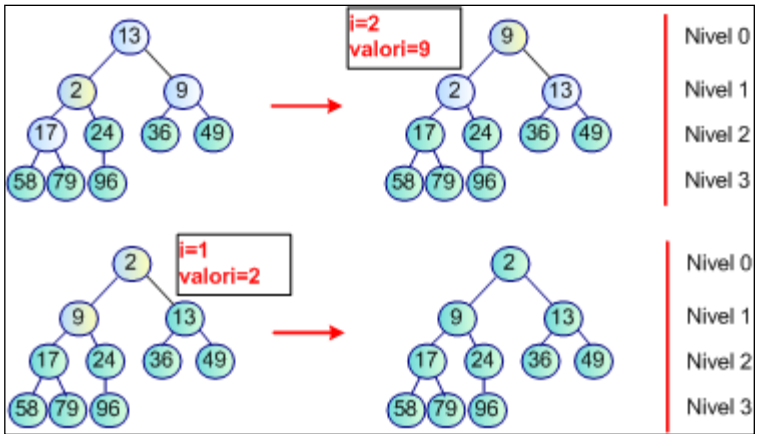


Figura 102: "Fase 2 del Heapsort: Decapitación y Reordenamiento del Grupo Descendentemente Ordenado"

Una vez finalizado el heapsort, el arreglo queda completamente ordenado en forma secuencial al vector, note que la misma situación en términos del último árbol binario de la figura 102, se da leyéndolo desde arriba hacia abajo y de izquierda a derecha.

Finalmente y, en absoluto por ello menos importante, queda analizar el rendimiento del heapsort. Debido a su bucle interno al for, el heapsort es dos veces más lento que el Quicksort, pero no por ello implica que es un mal algoritmo, sino que tenga en cuenta que independientemente del tamaño del arreglo, el heapsort ***sin usar recursos adicionales del sistema*** ordena un arreglo en no más de *2*n*lg(N)* comparaciones.

La comprobación de dicha cota superior no es tema de este capítulo, pero se recomienda fuertemente que tomando como base el capítulo de "Análisis de Algoritmos", compruebe la afirmación sobre el rendimiento del heapsort.

Combsort

El Ordenamiento por "Combsort", también conocido como ordenamiento por "Peine", fue desarrollado por Richard Box y Stephen Lacey. Publicado en sus comienzos en la revista Byte, edición de Abril de 1991 (USA), el Combsort supera al Quicksort en número de comparaciones necesarias, y no es ello motivo para no analizarlo, ya que por ejemplo, si dispone de mejor rendimiento que el Heapsort. El análisis de la eficacia y eficiencia del Algoritmo Combsort, para con los enunciados en el presente párrafo, no será analizado en el presente capítulo y se lo deja como tarea para el lector, teniendo en cuenta el Capítulo "Análisis de Algoritmos".

El combsort corresponde al orden:

$$k * N * \log(N)$$

$$\forall K : 1 < K < 2$$

A Continuación se plantea una codificación base del algoritmo en C/C++, y posteriormente se analizará la funcionalidad del mismo.

Archivo: **combsort.cpp**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       #define shrinkfactor 1.3
5
6       void combsort(int[],int);
7
8       void main(void)
9       {
10       int aordenar[10]={96,58,79,2,9,36,24,13,17,49};
11
12       clrscr();
13       cout<<"Arreglo Inicial\n";
14       for(int i=0;i<10;i++)
15         cout<<aordenar[i]<<"-";
16
17       combsort(aordenar,10);
18       cout<<"\n\nArreglo Ordenado\n";
19       for(i=0;i<10;i++)
20         cout<<aordenar[i]<<"-";
21      }
22
23      void combsort(int v[],int tam)
24      {
25       int switches, hold, i, j, top, gap;
26
27       gap=tam;
28
```

```
29      do{
30         gap=(int)((float)gap/shrinkfactor);
31
32         switch(gap)
33         {
34          case 0://el gap m s peque¤o es 1
35            gap=1;
36            break;
37          case 9://esto es lo que hace este combsort
38          case 10:
39            gap=11;
40            break;
41          default:
42            break;
43         }
44
45         switches=0; //bandera de paso sucio
46         top=tam-gap;
47
48         for(i=0;i<top;++i)
49         {
50          j=i+gap;
51
52          if(v[i]>v[j])
53          {//intercambio
54           hold=v[i];
55           v[i]=v[j];
56           v[j]=hold;
57           ++switches;
58          }
59         }
60        }while(switches || (gap>1));
61     }
62
```

Note que a primera vista, sin entrar en detalles funcionales, podemos apreciar que el Combsort adolece de una debilidad similar a la del heapsort, “Dos Ciclos Anidados”. Vea que el ciclo principal, el que dirige el algoritmo, es el *do-while* donde dentro del mismo y por cada iteración, se ejecuta un bucle *for*, esto es posiblemente lo que incrementa el número de comparaciones necesarias, que lo hace menor eficiente que el Quicksort.

Variables							Arreglo									
tam	switches	i	j	top	gap	shrinfactor	0	1	2	3	4	5	6	7	8	9
10					10	1,3	96	58	79	17	49	36	24	2	13	9
10	0	0	7	3	7	1,3	96	58	79	17	49	36	24	2	13	9
	1	1	8				2	58	79	17	49	36	24	96	13	9
	2	2	9				2	13	79	17	49	36	24	96	58	9
	3	3					2	13	9	17	49	36	24	96	58	79
	0	0	5	5	5		2	13	9	17	49	36	24	96	58	79
	0	1	6				2	13	9	17	49	36	24	96	58	79
	0	2	7				2	13	9	17	49	36	24	96	58	79
	0	3	8				2	13	9	17	49	36	24	96	58	79
	0	4	9				2	13	9	17	49	36	24	96	58	79
	0	5					2	13	9	17	49	36	24	96	58	79
	0	0	3	7	3		2	13	9	17	49	36	24	96	58	79
		1	4				2	13	9	17	49	36	24	96	58	79
		2	5				2	13	9	17	49	36	24	96	58	79
		3	6				2	13	9	17	49	36	24	96	58	79
		4	7				2	13	9	17	49	36	24	96	58	79
		5	8				2	13	9	17	49	36	24	96	58	79
		6	9				2	13	9	17	49	36	24	96	58	79
	0	7					2	13	9	17	49	36	24	96	58	79
	0	0	2	8	2		2	13	9	17	49	36	24	96	58	79
		1	3				2	13	9	17	49	36	24	96	58	79
		2	4				2	13	9	17	49	36	24	96	58	79
		3	5				2	13	9	17	49	36	24	96	58	79
		4	6				2	13	9	17	49	36	24	96	58	79
	1	5	7				2	13	9	17	24	36	49	96	58	79
		6	8				2	13	9	17	24	36	49	96	58	79
		7	9				2	13	9	17	24	36	49	96	58	79
	2	8					2	13	9	17	24	36	49	79	58	96
	0	0	1	9	1		2	13	9	17	24	36	49	79	58	96
	1	1	2				2	13	9	17	24	36	49	79	58	96
	1	2	3				2	9	13	17	24	36	49	79	58	96
	1	3	4				2	9	13	17	24	36	49	79	58	96
	1	4	5				2	9	13	17	24	36	49	79	58	96
	1	5	6				2	9	13	17	24	36	49	79	58	96
	1	6	7				2	9	13	17	24	36	49	79	58	96
	1	7	8				2	9	13	17	24	36	49	79	58	96
	2	8	9		1		2	9	13	17	24	36	49	58	79	96
					0		2	9	13	17	24	36	49	58	79	96
	0	0	1	9	1		2	9	13	17	24	36	49	58	79	96
		1	2				2	9	13	17	24	36	49	58	79	96
		2	3				2	9	13	17	24	36	49	58	79	96
		3	4				2	9	13	17	24	36	49	58	79	96
		4	5				2	9	13	17	24	36	49	58	79	96
		5	6				2	9	13	17	24	36	49	58	79	96
		6	7				2	9	13	17	24	36	49	58	79	96
		7	8				2	9	13	17	24	36	49	58	79	96
		8	9				2	9	13	17	24	36	49	58	79	96
	0	9			1		2	9	13	17	24	36	49	58	79	96

Figura 103: “Combsort: Decapitación y Reordenamiento del Grupo Descendentemente Ordenado”

La condición de corte del ciclo principal (*do-while*) del Combsort está regulada por las variables “switches” y “gap”.

- **switches:** Esta variable representa la cantidad de cambios totales efectuados durante una iteración del ciclo principal.
- **gap:** Representa la apertura entre los extremos de comparación, conocido también como “Distancia entre los dientes del peine”. El Gap determinará en primer lugar la cantidad de comparaciones en el buble interno de una iteración del bucle principal y, en segundo lugar, la distancia *constante* entre los valores a comparar. Note que la distancia entre los valores a comparar no se altera sino hastala próxima iteración del ciclo y la misma, tiene la particularidad de ser casi monótona decreciente ¿Porqué casi?, porque note que si gap es ‘1’ pero ocurrió un cambio (switches>0), implica que automáticamente se colocará gap en 1, para hacer una nueva pasada lineal por el arreglo, verificando el orden de los valores.

Note que la finalizació del Combsort está dada en función de las dos variables enunciadas con anterioridad, en consecuencia el mismo finalizará si:

- No Existieron cambios en la iteración **“y”**
- gap es menor o igual a 1

Vea que si gap es uno, implica que se efectuó una comparación lineal entre los valores del arreglo. Si hubiese existido un cambio (switches>1) y gap es uno, implica que al reingresar en una nueva iteración gap=(int)(1/1.3)=0 lo que lleva al gap a convertirse en 1 (uno) ¿Porqué? Por una cuestión muy obvia, un valor nunca será mayor y/o menor a sí mismo, si recordamos que el gap es la distancia entre los valores a comparar, implicaría el gap cero que ambos valores en realidad son los mismos, de allí que el *switch de la línea 32* lo coloca en 1.

Si el valor de gap fuese 9 o10, se actualiza el valor del mismo a 11, esto es debido a estudios empríricos de rendimiento del algoritmo efectuados por parte de los autores, al igual que el valor 1.3 de shrinfactor.

El valor constante shrinkfactor (Factor de Encogimiento), es un valor empírico que determina la magnitud de acortamiento del intervalo de comparación entre valores, por cada iteracción del ciclo principal, dado por gap. Éste último, al igual que los valores de gap 9/10, fueron estudiados y analizados por los autores del algoritmo. Hay algo garantizado por los autores del algoritmo, y es que el valor shrinkfactor de 1,3 es el que provoca los mejores resultados en términos de rendimiento del algoritmo, queda como tarea para el lector, analizar el incremento de la cantidad de comparaciones ante la variación de éste parámetro.

El bucle interno, hace el barrido “en peine”, comparando los valores a intervalos regulares, siempre partiendo desde la posición inicial del arreglo. Note que el único caso en que hace un barrido lineal del arreglo el algoritmo es cuando gap es 1. Quien define la cantidad de comparaciones por cada ciclo interno *for*, es tam y gap. De acuerdo al tamaño que disponga elarreglo y la distancia constante (por iteración) entre los valores a comparar, es que se obtiene la cantidad de comparaciones que se podrán efectuar partiendo desde el inicio del arreglo.

Intercambio de Residuos

Hasta el momento se han analizado los Algoritmos sin prestar demasiada atención a la naturaleza de la clave de ordenamiento, hemos tratado indistintamente los casos en donde la clave era entera, flotante, flotante de doble precisión, cadena de caracteres, entre otros. El Algoritmo de Ordenamiento del presente apartado, aprovecha el hecho de que algunas claves de ordenamiento puedan ser interpretadas como un valor numérico en un ***intervalo finito*** y así, poder efectuar comparaciones básicas a nivel de bits para lograr el ordenamiento definitivo sin necesidad de someterse a la naturaleza precisa de cada clave.

El nombre de "Residuos" proviene de que éste tipo de algoritmo toma la clave como un número en un sistema de numeración en base "B" (el residuo), y trabaja con las cifras que forman los números. En la mayoría de los computadores, la base será binaria (dos, M=2), o bien potencia de dos, debido a la naturaleza y facilidad de cálculo del computador.

El Algoritmo de Intercambio de Residuos, en la versión recursiva que se analizará en el presente apartado, posee un orden de N*Log(N) comparaciones de *bits*.

El Presente Algoritmo se basa en la interpretación de la clave como una cadena de *bits* de una longitud definida y especificada, al momento de la ejecución del algoritmo. Antes de ingresar en el análisis del algoritmo, haremos un breve repaso del complemento a uno en el álgebra binaria.

Complemento a Uno

Sea "A" un número binario, el complemento a uno de A se obtiene restando 1 a cada dígito de A.

Veamos un ejemplo Sencillo, si A fuese 11110000 el complemento a uno de A sería:

Se Resta 1 menos cada Dígito de A	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Número Binario "A"	1	1	1	1	0	0	0	0
Complemento a 1 de "A"	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

Operaciones Binarias Aplicables al Ordenamiento por Residuo

El Ordenamiento por residuo básicamente utiliza cuatro operaciones a nivel de bits, analizadas en el capítulo 1:

- Complemento a 1
- And Lógico
- Rotación a Izquierda
- Rotación a Derecha

El Algoritmo necesita estas operaciones binarias para poder contestar a la pregunta: Los "J" bits del valor "X", que se encuentra a "K" elementos del bit de mayor peso ¿Quiénes son?, La función que contesta a esta pregunta en el programa iresiduo.cpp (a Analizar posteriormente) es la siguiente:

```
26     unsigned JbitsAlaDerdeKBits(int valor ,int k, int j)
```

```
27      {
28        return (valor>>k) & ~(~0<<j);
29      }
```

Veamos como trabaja la función, suponga que valor es igual a 1000 (En base 10), k=4 y J=2, entonces la pregunta sería "Los 2 bits del valor 1000, que se encuentran a 4 elementos del bit de mayor peso (inclusive) ¿Quiénes son?"

Decimal	**Binario**									
Valor=1000	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**
Valor>>4	0	0	0	0	1	1	1	1	1	0
~ (~0<<2)	0	0	0	0	0	0	0	0	1	1
2 Dígitos Posteriores a los 4 Elementos iniciales en 1000 ((valor>>4)&(~(~0<<2)))	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Este mecanismo de análisis de bits se empleará de la siguiente forma en el algoritmo de residuos:

JbitsAlaDerdeKBits(v[i],b,1)

Donde:

- V[i]: Es el valor del arreglo v en la posición "i",
- B: Inicialmente representa la cantidad de bits que contiene como máximo la clave de ordenamiento, luego irá decrementándose de modo de analizar los bits de los valores del arreglo de izquierda a derecha (En efecto, desde los bits más significativos a los menos),
- 1: Implica que solo verá el valor de un bits posterior a los "B" elementos, iniciando la cuenta en el bits más significativo.

A continuación, se muestra una codificación base del algoritmo de ordenamiento por residuos recursivo:

```
Archivo: iresiduo.cpp

1       #include <iostream.h>
2       #include <conio.h>
3
4       #define shrinkfactor 1.3
5
6       void iresiduo(int[],int,int,int);
7
8       void main(void)
9       {
10       int aordenar[10]={96,58,79,2,9,36,24,13,17,49};
11
12       clrscr();
13
14       cout<<"Arreglo Inicial\n";
```

```
15      for(int i=0;i<10;i++)
16        cout<<aordenar[i]<<"-";
17
18      cout<<"\n\nBits de un Entero sobre la Arquitectura Actual:
       "<<(sizeof(int)*8);
19      int b=(sizeof(int)*8);//"b" es la cantidad de bits que compone la clave
20      iresiduo(aordenar,0,9,b);
21      cout<<"\nArreglo Ordenado\n";
22      for(i=0;i<10;i++)
23        cout<<aordenar[i]<<"-";
24     }
25
26     unsigned JbitsAlaDerdeKBits(int valor ,int k, int j)
27     {
28       return (valor>>k) & ~(~0<<j);
29     }
30
31     void iresiduo(int v[],int izq, int der, int b)
32     {
33      int i, j, t;
34
35      if(der>izq && b>=0)
36      {
37       i=izq;
38       j=der;
39
40       while(j!=i)
41       {
42        while(!JbitsAlaDerdeKBits(v[i],b,1) && i<j) i++;
43        while(JbitsAlaDerdeKBits(v[j],b,1) && j>i) j--;
44        int aux=v[i];
45        v[i]=v[j];
46        v[j]=aux;
47       }
48
49       if(!JbitsAlaDerdeKBits(v[der],b,1)) j++;
50       iresiduo(v,izq,j-1,b-1);
51       iresiduo(v,j,der,b-1);
52      }
53     }
54
```

El Ordenamiento por residuo efectúa los cambios en base al análisis de los bits de los valores que integran el arreglo. Inicia en el Bits de mayor peso (El 7 en un byte, el 15 en dos bytes –o un Word-) y desplaza todos aquellos valores que tienen activado (En 1) dicho bits al final del arreglo, dejando en el principio aquellos valores que lo tienen desactivado (En 0), una vez que efectuó este primer análisis, acorta el intervalo a aquellos valores que tenían el bit de mayor peso en 0 y vuelve a ordenarlos de la misma forma pero considerando el (bits-1) –Este es el comportamiento naturalde la rama

izquierda de la recursión-. Una vez que culmina este ordenamiento, la rama derecha de la recursión, analiza los valores en cada grupo, ordenándolos definitivamente. Esto último es lo siguiente, suponga que tres valores tienen el bit 6 en "1", el algoritmo luego de colocarlos al fnal, continúa analizando el bit 5 de los restantes, pero ¿Cómo sabe si estos tres valores que coincidían en 1 en el bit 6 son iguales, mayores o menores entre sí? Cuando se ejecuta la rama derecha del algoritmo recursivo, acota el intervalo de análisis solo a aquellos valores que coinciden en el peso de un bit dado (suponga el 6), y los reordena por el 5, 4…0 hasta lograr el orden definitivo de este sub grupo y continuar analizando luego otro subgrupo.

Veamos gráficamente y paso a paso el funcionamiento del algoritmo:

1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	96	0	1	1	0	0	0	0	0	8	0	9	0	9
1	58	0	0	1	1	1	0	1	0				1	10
2	79	0	1	0	0	1	1	1	1				2	
3	17	0	0	0	1	0	0	0	1				3	
4	49	0	0	1	1	0	0	0	1				4	
5	36	0	0	1	0	0	1	0	0				5	
6	24	0	0	0	1	1	0	0	0				6	
7	2	0	0	0	0	0	0	1	0				7	
8	13	0	0	0	0	1	1	0	1				8	
9	9	0	0	0	0	1	0	0	1				9	

1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	96	0	1	1	0	0	0	0	0	7	0	9	0	9
1	58	0	0	1	1	1	0	1	0					
2	79	0	1	0	0	1	1	1	1					
3	17	0	0	0	1	0	0	0	1					
4	49	0	0	1	1	0	0	0	1					
5	36	0	0	1	0	0	1	0	0					
6	24	0	0	0	1	1	0	0	0					
7	2	0	0	0	0	0	0	1	0					
8	13	0	0	0	0	1	1	0	1					
9	9	0	0	0	0	1	0	0	1					

1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	7	0	9	0	9
1	58	0	0	1	1	1	0	1	0				1	8
2	79	0	1	0	0	1	1	1	1				2	
3	17	0	0	0	1	0	0	0	1					
4	49	0	0	1	1	0	0	0	1					
5	36	0	0	1	0	0	1	0	0					
6	24	0	0	0	1	1	0	0	0					
7	2	0	0	0	0	0	0	1	0					
8	13	0	0	0	0	1	1	0	1					
9	96	0	1	1	0	0	0	0	0					

1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	7	0	9	0	9
1	58	0	0	1	1	1	0	1	0				1	8
2	79	0	1	0	0	1	1	1	1				2	
3	17	0	0	0	1	0	0	0	1					
4	49	0	0	1	1	0	0	0	1					
5	36	0	0	1	0	0	1	0	0					
6	24	0	0	0	1	1	0	0	0					
7	2	0	0	0	0	0	0	1	0					
8	13	0	0	0	0	1	1	0	1					
9	96	0	1	1	0	0	0	0	0					

1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	7	0	9	0	9
1	58	0	0	1	1	1	0	1	0				1	8
2	13	0	0	0	0	1	1	0	1				2	
3	17	0	0	0	1	0	0	0	1				3	
4	49	0	0	1	1	0	0	0	1				4	
5	36	0	0	1	0	0	1	0	0				5	
6	24	0	0	0	1	1	0	0	0				6	
7	2	0	0	0	0	0	0	1	0				7	
8	79	0	1	0	0	1	1	1	1				8	
9	96	0	1	1	0	0	0	0	0					

1.1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	6	0	7	0	7
1	58	0	0	1	1	1	0	1	0				1	
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	49	0	0	1	1	0	0	0	1					
5	36	0	0	1	0	0	1	0	0					
6	24	0	0	0	1	1	0	0	0					
7	2	0	0	0	0	0	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

Figura 104: "Intercambio de Residuos: Ordenamiento por el Bits de Mayor Peso al de Inferior Peso (Parte 1 de 5)"

En la gráfica se inició el análisis desde el bit 7, ya que en un byte perfectamente es posible representar los valores contenidos del arreglo y para no hacer tan extenso el análisis gráfico.

Note como inicialmente revisa el bits “7” el de mayor peso, y al no encontrar que desplazar, se reinvoca el algoritmo (en su rama izquierda), para continuar el análisis en el (bits-1), el 6. Al iniciar el Análisis del bits 6, note que encuentra un valor con el bit encendido al inicio del arreglo y busca su contrapartida para intercambiarlo ¿Quién será su contrapartida? El primer valor en las posiciones finales del arreglo (contando desde el final del mismo y decrementando de a una unidad) que posea éste bits 6 apagado; en el ejemplo es el valor “9” en la posición 9, note que efectúa el intercambio entre el 9 y el 96 y continua analizando a partir de la nueva posición del nueve (0 en el arreglo). Al continuar analizando, encuentra el bit 6 encendido nuevamente en la posición 2 del arreglo, entonces buscará la posición donde cambiarlo (será aquella posición que contando desde la posición 8 del arreglo -8 debido a que en 9 ya efectuó una rotación- a la cero, tenga el bits 6 apagado); el cambio lo realizará a la posición 8 que contiene el valor “13”. Continuará analizando el bits 6 hasta que j sea igual a i (lo que implica que se analizó el arreglo en el bits 6 completamente), o que se crucen (j<i).

1.1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	6	0	7	0	7
1	2	0	0	0	0	0	0	1	0				1	6
2	13	0	0	0	0	1	1	0	1				2	
3	17	0	0	0	1	0	0	0	1				3	
4	49	0	0	1	1	0	0	0	1				4	
5	36	0	0	1	0	0	1	0	0					
6	24	0	0	0	1	1	0	0	0					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	6	0	7	0	7
1	2	0	0	0	0	0	0	1	0				1	6
2	13	0	0	0	0	1	1	0	1				2	5
3	17	0	0	0	1	0	0	0	1				3	
4	24	0	0	0	1	1	0	0	0				4	
5	36	0	0	1	0	0	1	0	0				5	
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	5	0	4	0	4
1	2	0	0	0	0	0	0	1	0				1	3
2	13	0	0	0	0	1	1	0	1				2	
3	17	0	0	0	1	0	0	0	1				3	
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	9	0	0	0	0	1	0	0	1	4	0	2	0	2
1	2	0	0	0	0	0	0	1	0					1
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	4	0	2	0	2
1	9	0	0	0	0	1	0	0	1				1	1
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.1.1.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	3	0	0	0	2
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

Figura 105: “Intercambio de Residuos: Ordenamiento por el Bits de Mayor Peso al de Inferior Peso (Parte 2 de 5)”

Note que el tramo recorrido por el análisis de la variable “i” se denota con recuadros continuos, mientras que los recuadros punteados denotan el recorrido de la variable “j”. En caso de que las variables “j” e “i” se intersecten en la trayectoria en un punto dado, la misma se representará por la mitad del recuadro continuo y la otra mitad punteada.

1.1.1.1.1.2

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	3	1	2	1	2
1	9	0	0	0	0	1	$\boxed{0}$	0	1				2	
2	13	0	0	0	0	1	$\boxed{1}$	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.1.2.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	2	1	1		
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.1.2.2

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	2	1	2	1	2
1	9	0	0	0	0	1	0	$\boxed{0}$	1				2	
2	13	0	0	0	0	1	1	$\boxed{0}$	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.1.2.2.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	1	1	1		
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.1.2.2.2

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	1	2	2		
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.2

(Valores b, izq y der vienen del cuadro 1.1.1.1)

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	4	3	4	3	4
1	9	0	0	0	0	1	0	0	1				4	
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	$\boxed{0}$	0	0	1					
4	24	0	0	0	1	$\boxed{1}$	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

Figura 106: “Intercambio de Residuos: Ordenamiento por el Bits de Mayor Peso al de Inferior Peso (Parte 3 de 5)”

¿Notó que en la primera pasada del algoritmo j fue llevado a 10? Si, eso se debe a que como todos los bits 7 estaban apagados, implica que al reinvocarsela función se volverá a analizar *todos* los valores nuevamente del arreglo en el bits-1, es decir el bits 6. De no haberse efectuado el incremento, se hubiese producido un acortamiento del

intervalo sin conocer realmente si un elemento tortuga valor se encontraba en las últimas posiciones realmente.

1.1.1.1.2.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	3	3	3		
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.1.2.2

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	3	4	4		
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.2

(Valores b, izq y der vienen del cuadro 1.1.1)

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	5	5	7	5	7
1	9	0	0	0	0	1	0	0	1				6	6
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.2.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	4	5	5	5	6
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.2.2

(Valores b, izq y der vienen del cuadro 1.1.1.2)

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	4	6	7	6	7
1	9	0	0	0	0	1	0	0	1				7	
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.1.2.2.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	3	6	6		
1	9	0	0	0	0	1	0	0	1		XXXXXX			
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

Figura 107: "Intercambio de Residuos: Ordenamiento por el Bits de Mayor Peso al de Inferior Peso (Parte 4 de 5)"

El Truncamiento de las ramas, y el inicio del retorno de la misma se indican con cruces rojas, en el mismo momento que se produce el retorno en la figura.

1.1.1.2.2.2

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	3	7	7		
1	9	0	0	0	0	1	0	0	1	XXXXXX				
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.2

(Valores b, izq y der vienen del cuadro 1.1)

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	6	8	9	8	9
1	9	0	0	0	0	1	0	0	1				9	
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.2.1

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	5	8	8		
1	9	0	0	0	0	1	0	0	1	XXXXXX				
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.1.2.2

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	5	9	9		
1	9	0	0	0	0	1	0	0	1	XXXXXX				
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

1.2

(Valores b, izq y der vienen del cuadro 1)

Posición en el Arreglo		Bits 7	6	5	4	3	2	1	0	Variables b	izq	der	i	j
0	2	0	0	0	0	0	0	1	0	7	10	9		
1	9	0	0	0	0	1	0	0	1	XXXXXX				
2	13	0	0	0	0	1	1	0	1					
3	17	0	0	0	1	0	0	0	1					
4	24	0	0	0	1	1	0	0	0					
5	36	0	0	1	0	0	1	0	0					
6	49	0	0	1	1	0	0	0	1					
7	58	0	0	1	1	1	0	1	0					
8	79	0	1	0	0	1	1	1	1					
9	96	0	1	1	0	0	0	0	0					

Figura 108: "Intercambio de Residuos: Ordenamiento por el Bits de Mayor Peso al de Inferior Peso (Parte 5 de 5)"

En el recuadro "1.1.2" de la figura 108, puede apreciarse claramente el comportamiento de la rama derecha del algoritmo. En dicho recuadro, vea que solamente el recorrido se dio entre las posiciones 8 y 9 del arreglo, no es casualidad que esas mismas posiciones sean las únicas que tienen encendido el bit 6, ¿Pero el recorrido es en el bit 6? No, es en el bit 5 ¿Para qué? Para saber si el valor que actualmente se encuentra en la posición 9 del arreglo debe estar allí o es menor en realidad al de la posición 8 ¿Quién ayuda a saber si el valor es menor, mayor o igual? Los bits 5 a 0 en ese orden de inspección, como en nuestro ejemplo los dos valores difieren en el bit 5, y el bit de la posición 9 lo tiene encendido mientras que el de la posición 8 no, esto le indica al algoritmo que no debe efectuar cambios ya que el valor está bien ubicado, de

no haberlo estado, se haría el intercambio de los valores para que los mismos queden ordenados definitivamente.

En Síntesis, note que el comportamiento del algoritmo de ordenamiento, a rasgos gnerales es de la siguiente forma:

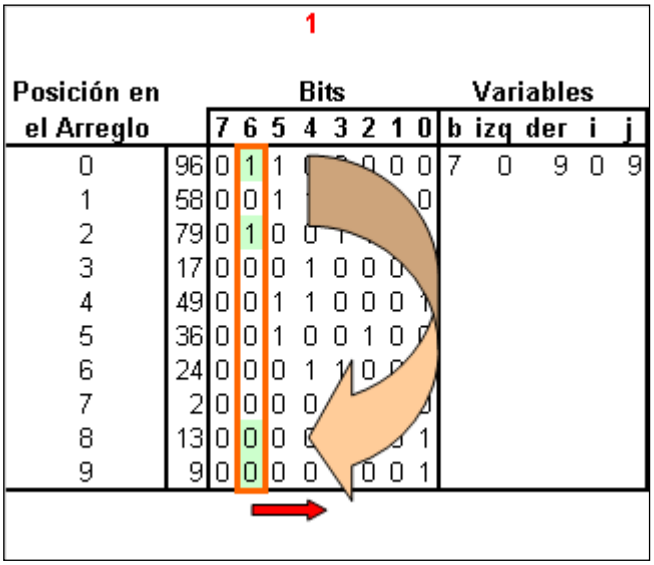


Figura 109: “Intercambio de Residuos: Interpretación General de su Comportamiento”

En este caso se tomo el bits 6 debido a que en él, existen efectivamente intercambios. Vea que el Algoritmo de Ordenamiento por Residuos Analiza de Izquierda a Derecha los bits de los valores que contiene el arreglo. La variable “i” conduce la búsqueda de un bits encendido, cuando lo encuentra, la variable “j” conduce la búsqueda (desde la posición final del arreglo y decrementando de a uno), de un valor con el mismo bit apagado, al ubicarlo se efectúa el intercambio. Note que en la figura 109, el primer bit encendido se intercambiará a la primera posición desde atrás hacia delante, con el mismo bit apagado (la posición 9). El segundo con el bit encendido, encontrado por el recorrido de “i”, se intercambiará a la primera posición desde atrás hacia delante, con el mismo bit apagado (la posición 8). De esta forma hasta culminar el análisis de los valores del arreglo para el bit 6, luego de lo cual se efectuará la misma operatoria para el bit 5, 4, 3, 2, 1 y 0. Recuerde que la rama derecha es la que analizará si los valores en los subgrupos formados (aquellos que poseen el mismo bit de mayor peso encendido) están ordenados definitivamente o no, que fue el último aspecto anañlizado en la página anterior.

Referencias

- "Estructuras de Datos con C y C++"; Langsam, Aungenstein y Tenenbaum; Editorial Prentice Hall; México, 1997.

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995.

- "Combsort"; Richard Box y Stephen Lacey; Revista Byte; Abril de 1991.

- "The Art of Computer Programming", Volumen 3: Sorting and Searching, Segunda Edición; D.E. Knuth, Editorial Addison-Wesley, 1975.

- "Sorting"; W.A. Martin; ACM Computing Surveys, Volumen 3, Edición 4; Diciembre de 1971.

6

Algoritmos y Estructuras para Búsqueda

Introducción a los Algoritmos y Estructuras para Búsqueda

Hasta el momento hemos tratados algoritmos de ordenamientos y estructuras de datos, pero no se ha dicho aún mucho sobre mecanismos de recupración y/o búsqueda de datos en sí.

Anteriormente, se han tratado las búsquedas binarias y secuenciales como ejemplos básicos de recuperación de datos, por lo que no serán vueltos a considerar en el presente capítulo, pero sí tenga en cuenta que los mismos corresponden a algoritmos de búsqueda de datos.

Así también, se abordó la temática de árboles binarios, en el apartado correspondiente a las Estructuras de Datos. Los Árboles Binarios eran estructuras que nos facilitaban la búsqueda o recuperación, pero sufrían de una fuerte debilidad asociada al orden en que se insertaban los datos en el mismo, ya que de acuerdo a este último, iba a resultar una estrutura desequilibrada o no. No se volverá sobre los árboles binarios en sí, pero si se abordará la problemática del desequilibrio en los árboles binarios a través de los árboles conocidos como “AVL”.

Posteriormente se analizará la aplicación de árboles de n-Claves como mecanismos de recuperación de datos, fuertemente utilizados en las indexaciones en disco de un sistema de administración de bases de datos.

Finalmente, y no por ello menos importante, se abordará la técnica de Hashing o Dispersión, con sus distintas variantes ante colisiones, como un mecanismo de recuperación de datos que tiende a un orden “k” de consecución de la información.

Árboles AVL

El material correspondiente a la temática de árboles AVL tratados en el presente capítulo, está basado en el material del curso "Tecnología de Software de Base", impartido por el Ing. Valerio Fritelli en la Universidad Tecnológica Nacional, Facultad Córdoba, durante el año lectivo 1999.

La problemática natural asociada a los árboles binarios, consistía en que eran fuertemente dependientes del orden de inserción de los valores en su estructura. La Peor de las situaciones, en términos de búsqueda, la tendríamos si la secuencia de valores a insertar se encuentra ordenada, lo que provocará una lista ordenada (caso especial de arbol binario) que provocaría, una búsqueda secuencial de los valores, ya que la longitud de la lista es equivalente a la altura del árbol binario.

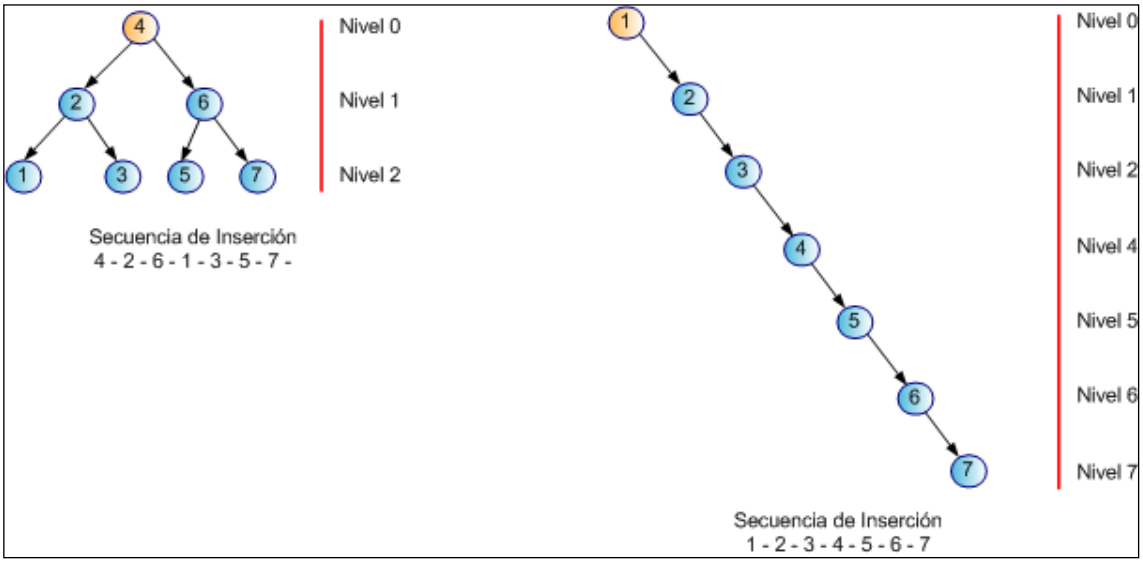


Figura 110: "Problema de Inserción en un Árbol Binario"

Qué mas quisiese un sistema, qe ante una búsqueda tenga un estructura de árbol como la primera, pero no siempre los datos en un árbol binario vienen con un orden tal que garantice dicha estructura, de hecho, es muy poco probable que podams garantizarlo. Con suerte, si nos esforzamos,con un árbol binario podamos evitar una estructura de árbol binario secuencial, ya que como se mencionó, note que la altura del árbol coincide con la longitud de la lista, loque provoca búsquedas secuenciales indefectiblemente.

Ahora bien, podríamos tratar por cada inserción y/o borrado del árbol binario, de reequilibrarlo completamente para lograr un equilibrio estricto, pero sería muy costoso en términos de comparaciones y tiempo adicional generado. M. **A**delson-**V**elskii y E.M. **L**audis, percatados de esta problemática de los árboles binarios, deciden incursionar en lo que denominaron árboles AVL (en honor a sus creadores) y mencionan:

"Un árbol binario se dice que está en equilibrio AVL, si y solo si para cada uno de sus nodos, ocurre que las alturas de sus subárboles difieren cuanto mucho en uno"

Analice los siguientes casos de árboles binarios:

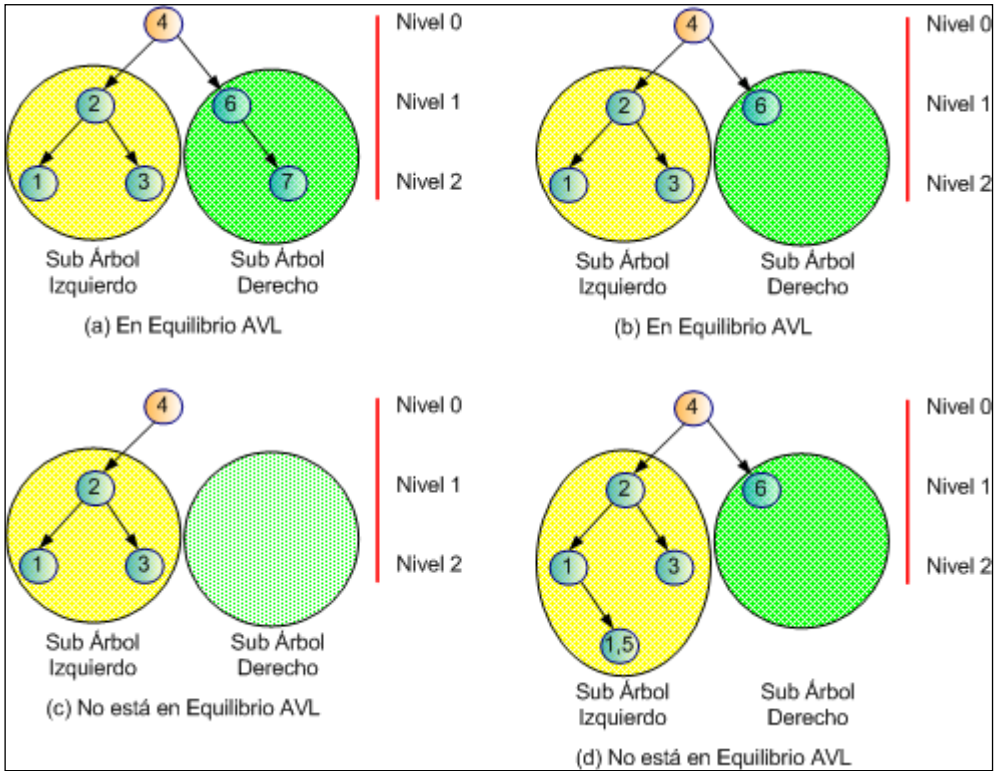


Figura 111: "Casos de Análisis de Equilibrio AVL"

En el caso "a" no hay diferencia de altura entre el subárbol izquierdo y derecho, y la diferencia de altura dentro del subárbol derecho es de uno, por lo que no viola la definición de equilibrio AVL y se puede concluir que cumple con tal definición, por lo que está en equilibrio AVL.

En el caso "b", si bien existe una diferencia de altura entre el subárbol izquierdo y derecho de la raíz del árbol, la diferencia es de uno, por lo que nuevamente no se viola la definición de equilibrio AVLy se puede concluir que se cumple con tal definición, por lo que está en equilibrio AVL.

El caso "c" rompe con elesquema que se traía en los puntos "a" y "b", en el c, la diferencia entre la altura del subárbol izquierdo y derecho, es de dos, esto conduce a que no se cumpla el equilibrio AVL.

El caso "d", también viola la definición de equilibrio AVL, debido a que la diferencia de altura entre los subárboles es de dos, esto implica que evidentemente no se encuentra en equilibrio AVL dicho árbol.

De acuerdo con lo menionado y demostrado por los autores del equilibrio AVL, un árbol AVL nunca será más alto que un 45% de su árbol perfectamente equilibrado. Esto pone a consideración, que una búsqueda en un árbol AVL si bien no será tan eficiente como una en su correspondiente árbol perfectamente equilibrado, no deja de ser eficaz y sacrifica parte de la eficiencia en la búsqueda, por ganar eficiencia en la inserción y borrado de valores, por lo que la convierte en una estructura de datos dinámica muy atrayente al momento en que los valores no sean demasiado estáticos.

Análisis de la Inserción de Valores en un árbol AVL

Sería tentador recalcular la situación de equilibrio cada vez que se inserta un valor, pero esto implicaría un sacrificio enorme en términos de tiempo adicional al proceso de inserción, por garantizar un equilibrio que puede ser sostenido a través de un campo adicional en cada nodo del árbol sin tanto sacrificio de procesamiento.

En cada nodo, se puede almacenar un campo denominado "equi", que contendrá la diferencia en la altura entre su subárbol derecho y su izquierdo, como sigue:

$$Equi = h_d - h_i$$

Donde:

h_d: Altura del Sub Árbol Derecho,
h_i: Altura del Sub Árbol Izquierdo

Veamos algunos ejemplos que como se aplicaría equi, para identificar si un árbol se encuentra o no en equilibrio AVL:

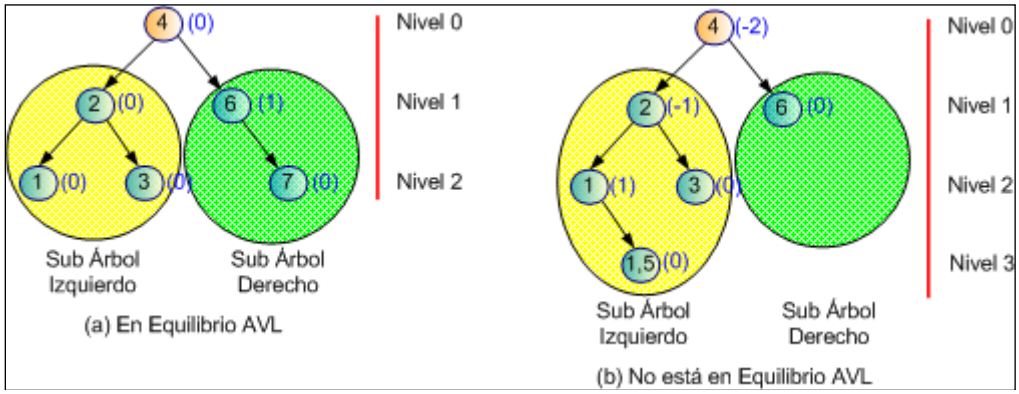


Figura 112: "Casos de Análisis de Equilibrio AVL con Cálculos de Alturas"

Analice el primer caso, el "a", el mismo se encuentra en equilibrio AVL,vea que la diferencia entre las alturas de sus subárboles no superan 1. Calculemos nodo a nodo su equi, para analizar como se obtienen los valores a la derechade cada nodo:

- **Nodo 4:** La Altura del Subárbol derecho es 2 (h_d), mientras que la altura del árbol izquierdo es 2 también (h_i). Entonces, equi será 2-2=0.
- **Nodo 2:** La Altura del Subárbol derecho es 1 (h_d), mientras que la altura del árbol izquierdo es 1 también (h_i). Entonces, equi será 1-1=0.
- **Nodo 6:** La Altura del Subárbol derecho es 1 (h_d), mientras que la altura del árbol izquierdo es 0 (h_i). Entonces, equi será 1-0=1.
- **Nodo 1:** La Altura del Subárbol derecho es 0 (h_d), mientras que la altura del árbol izquierdo es 0 (h_i). Entonces, equi será 0-0=0.
- **Nodo 3:** La Altura del Subárbol derecho es 0 (h_d), mientras que la altura del árbol izquierdo es 0 (h_i). Entonces, equi será 0-0=0.
- **Nodo 7:** La Altura del Subárbol derecho es 0 (h_d), mientras que la altura del árbol izquierdo es 0 (h_i). Entonces, equi será 0-0=0.

Pruebe usted de calcular los valores de equi para el caso "b", y podrá comrprobar la siguiente afirmación

"Un árbol en equilibrio AVL solo puede tener en su campo equi valores 0, 1 o -1"

Si equi es igual a cero, implica que h_i y h_d son iguales, por ende la altura de los subárbol izquierdo y derecho coinciden.

Si equi es -1, implica que h_i es mayor a h_d en una unidad, lo que no quiebra la regla de equilibrio AVL.

Si equi es 1, implica que h_d es mayor que h_i enana unidad, lo que tampoco quiebra la regla de equilibrio AVL.

Ahora bien, que ocurre si se insertase un nodo en el subárbol donde la situación fuese la siguiente:

Situación	Concepto
Si $h_i = h_d$	Al insertar un nuevo nodo, se produce una diferencia de alturas 1 o -1, dependiendo de si se inserta a la izquierda o a la derecha. No altera el equilibrio AVL, por lo que no es necesaria ninguna operación de reequilibrio.
Si $h_i > h_d$	Si se insertase el nodo en la derecha del subárbol, se igualan las alturas de los subárboles, por lo que no es necesario ningún reordenamiento del árbol. Pero si la inserción del valor se diese en el subárbol izquierdo, esto provocaría que la diferencia de altura entre subárboles sea superior a uno, lo que quiebra la regla de equilibro AVL y, son necesarias operaciones de reordenamiento para restaurar el equilibrio AVL.
Si $h_i < h_d$	Si se insertase el nodo en la izquierda del subárbol, se igualan las alturas de los subárboles, por lo que no es necesario ningún reordenamiento del árbol. Pero si la inserción del valor se diese en el subárbol derecho, esto provocaría que la diferencia de altura entre subárboles sea superior a uno, lo que quiebra la regla de equilibrio AVL y, son necesarias operaciones de reordenamiento para restaurar el equilibrio AVL.

Al momento de la inserción en un árbol AVL, y teniendo en cuenta la tabla anterior, se pueden dar básicamente dos tipos de desequilibrios: Un desequilibrio en Diagonal y un desequilibrio en ángulo.

El Desequilibrio en diagonal se soluciona con una rotación simple, a izquierda o derecha dependiendo en qué subárbol se de, veamos cómo sería este caso:

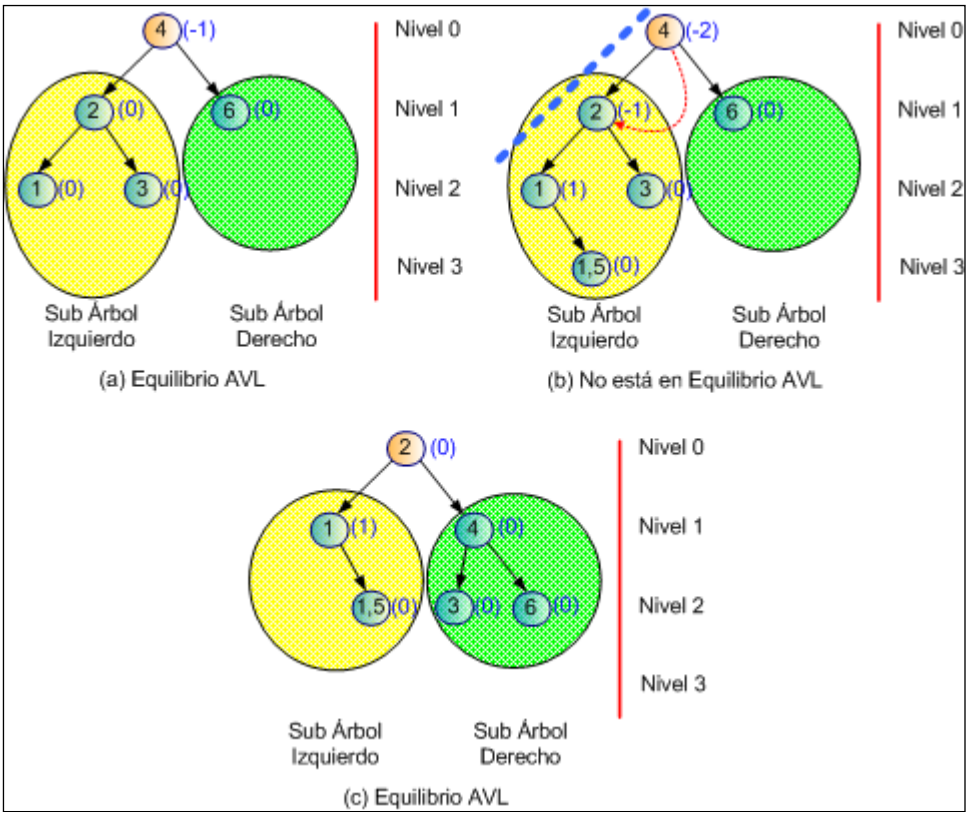


Figura 113: "Desequilibrio en Diagonal Izquierdo en un Árbol AVL"

Note que en la figura "a", el árbol AVL se encuentra en equilibrio AVL, cuando se inserta el valor "1,5" de no hacerse ningún reordenamiento, se perdería el equilibrio AVL quedando la raíz con una diferencia de dos niveles.

Note que el desquilibrio, indicado por "equi" se da en forma diagonal, donde originalmente el nodo 2 tenía un equi de "-1" (lo que ya indicaba la diferencia de un nivel) y, al insertarse el nodo 1,5 en el mismo subárbol izquierdo, el nodo 1 quedaría con equi igual a 1, mientras que el nodo 2 quedaría con un equi de -1, antes de llegar al nodo 4 y modificar su valor original de equi de -1 a -2, sabemos que con esta situación de equi en -1 en el nodo 2 y equi en -1 en el nodo 4, hay un desequilibrio diagonal. Esto nos lleva a que la solución tipo para el desequilibrio diagonal izquierdo sea:

1. Al Hijo derecho del nodo inferior con equi -1 (el nodo 3 es el hijo derecho) póngalo como hijo izquierdo del nodo superior con equi -1.
2. Rote el nodo superior con equi -1 (el nodo 4), como hijo derecho del nodo inferior con equi -1 (el nodo 2).
3. Direccione el puntero que direccionala al nodo superior con equi -1 (En este caso la raíz), al nodo inferior con equi -1 (al nodo 2) y ponga equi en cero.

En el caso del desequilibrio en diagonal derecho, es similar el procedimiento pero uso su simétrico, es decir, donde es izquierdo en el procedimiento anterior será derecho en el procedimiento de correción del desequilibrio de diagonal derecha.

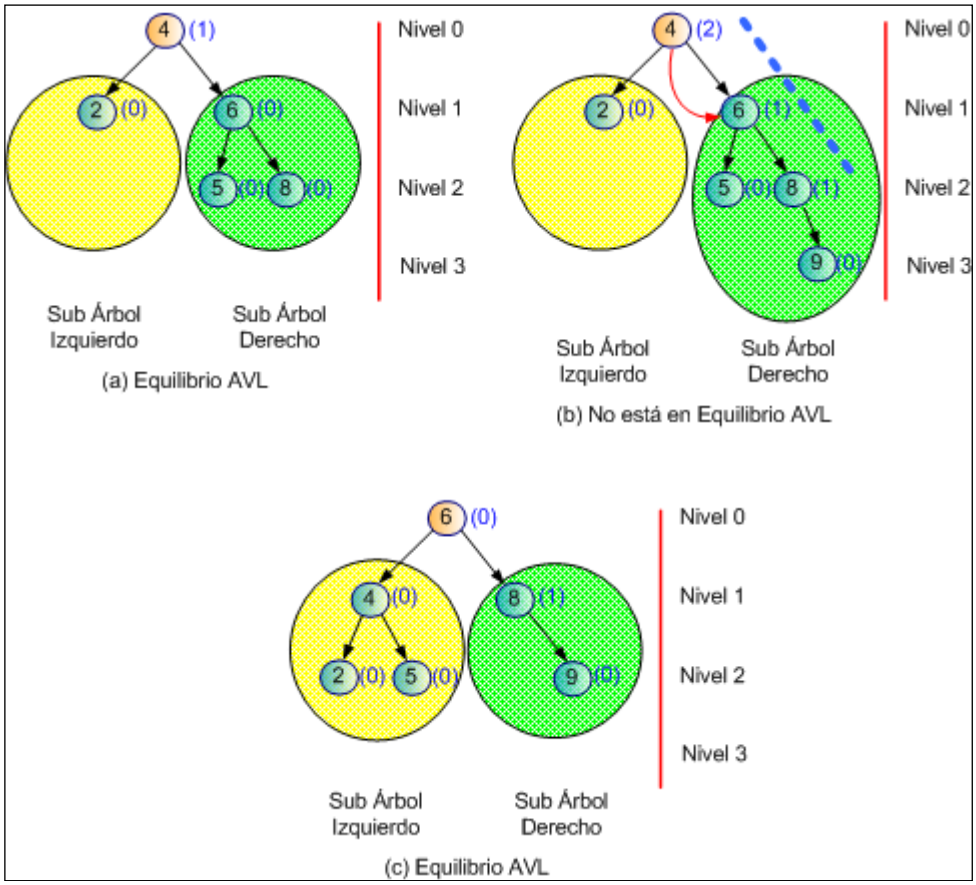


Figura 114: "Desequilibrio en Diagonal Derecho en un Árbol AVL"

Note que en la figura "a", el árbol AVL se encuentra en equilibrio AVL, cuando se inserta el valor "9" de no hacerse ningún reordenamiento, se perdería el equilibrio AVL quedando la raíz con una diferencia de dos niveles.

Note que el desquilibrio, indicado por "equi" se da en forma diagonal derecha, donde originalmente el nodo 4 tenía un equi de "1" (lo que ya indicaba la diferencia de un nivel) y, al insertarse el nodo 9 en el mismo subárbol derecho, el nodo 8 y 6 quedarían con equi igual a 1, mientras que el nodo 4 quedaría con un equi de 2, antes de llegar al nodo 4 y modificar su valor original de equi de 1 a 2, sabemos que con esta situación de equi en 1 en el nodo 8 y 6, y equi en 1 en el nodo 4, hay un desequilibrio diagonal derecho (note que en el desequilibrio diagonal derecho son valores positivos en equi, quienes denotan su desequilibrio). Esto nos lleva a que la solución tipo para el desequilibrio diagonal derecho sea:

1. Al Hijo izquierdo del nodo inferior con equi 1 (el nodo 5 es el hijo izquierdo) póngalo como hijo derecho del nodo superior con equi 1.
2. Rote el nodo superior con equi 1 (el nodo 4), como hijo izquierdo del nodo inferior con equi 1 (el nodo 6).
3. Direccione el puntero que direccionala al nodo superior con equi 1 (En este caso la raíz), al nodo inferior con equi 1 (al nodo 4) y ponga equi en cero.

Vea que el comportamiento de los procedimientos de rotación diagonal e izquierdo, son simétricos ente sí, lo único que cambia entre ellos es que donde en uno dice izquierdo, en el otro dice derechoy viceversa.

Pero note que hasta el momento solo hemos tratado el desequilibro diagonalñ, no se ha dicho nada aún sobre el desequilibrio en ángulo. El mismo consiste en la siguiente situación:

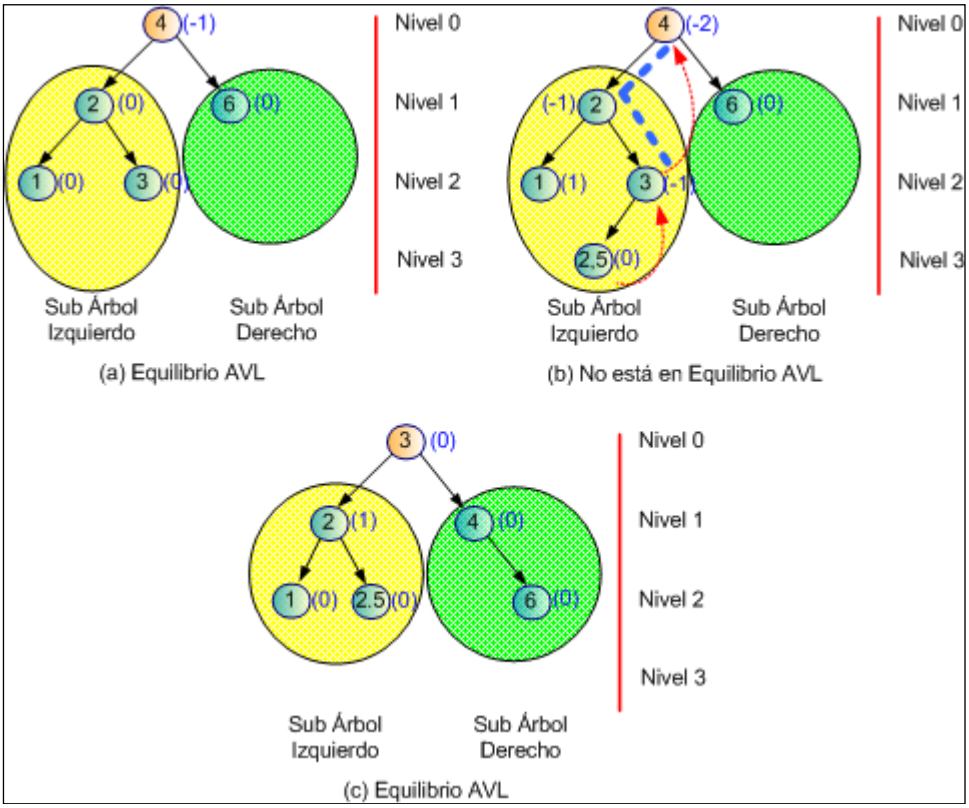


Figura 115: "Desequilibrio en Ángulo Izquierdo en un Árbol AVL"

Note que en la figura "a", el árbol AVL se encuentra en equilibrio AVL, cuando se inserta el valor "2.5" de no hacerse ningún reordenamiento, se perdería el equilibrio AVL quedando la raíz con una diferencia de dos niveles.

Note que el desquilibrio, indicado por "equi" se da en forma ángulo izquierdo, donde originalmente el nodo 4 tenía un equi de "-1" (lo que ya indicaba la diferencia de un nivel) y, al insertarse el nodo 2.5 en el mismo subárbol izquierdo, el nodo 3 y 2 quedarían con equi igual a -1, mientras que el nodo 4 quedaría con un equi de -2, antes de llegar al nodo 4 y modificar su valor original de equi de -1 a -2, sabemos que con esta situación de equi en -1 en el nodo 3 y 2, y equi en -1 en el nodo 4, hay un desequilibrio de ángulo izquierdo (note que en el desequilibrio en ángulo izquierdo son valores negativos en equi, quienes denotan su desequilibrio). Esto nos lleva a que la solución tipo para el desequilibrio de ángulo izquierdo sea como se muestra con flechas punteadas en el gráfico anterior.

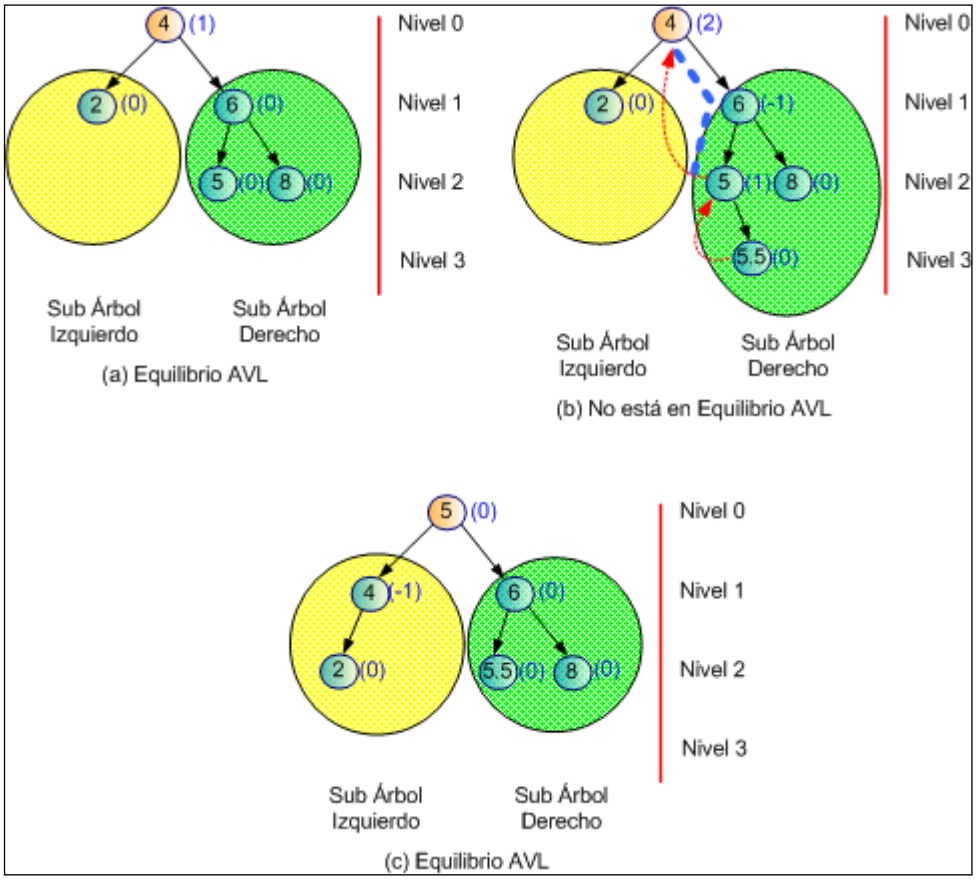


Figura 116: "Desequilibrio en Ángulo Derecho en un Árbol AVL"

Note que en la figura "a", el árbol AVL se encuentra en equilibrio AVL, cuando se inserta el valor "5.5" de no hacerse ningún reordenamiento, se perdería el equilibrio AVL quedando la raíz con una diferencia de dos niveles.

Note que el desquilibrio, indicado por "equi" se da en forma ángulo derecho, donde originalmente el nodo 4 tenía un equi de "1" (lo que ya indicaba la diferencia de un nivel) y, al insertarse el nodo 5.5 en el mismo subárbol derecho, el nodo 5 quedaría con equi igual a 1 y el nodo 6 con equi -1, mientras que el nodo 4 quedaría con un equi de 2, antes de llegar al nodo 4 y modificar su valor original de equi de 1 a 2, sabemos que con esta situación de equi en 1 y -1 en el nodo 5 y 6 respecitamente, y equi en 1 en el nodo 4, hay un desequilibrio de ángulo derecho. Esto nos lleva a que la solución tipo para el desequilibrio de ángulo derecho sea como se muestra con flechas punteadas en el gráfico anterior.

Análisis del Borrado de Valores en un árbol AVL

Naturalmente, y de la misma forma en que se vió cuando se trató el tema de árboles binarios, el borrado es una operación más compleja en este tipo de estructuras que la inserción misma.

Si ante el borrado de un determinado nodo, éste disponía de un hijo o ninguno, borrarlo no era tan complicado, el centro del asunto ocurrí cuando un nodo tenía sus dos hijos asociados. Cuando ocurría que un nodo tenía sus dos hijos, se reemplazaba a éste nodo a eliminar por su mayor descendiente izquierdo.

Ahora bien, hasta el momento lo dicho se corresponde con árboles binarios exclusivamente, no se ha mencionado nada sobre árbol AVL o equilibroAVL, en efecto, luego de que el nodo es eliminado de la lista y reemplazado por su mayor descendiente izquierdo, a través del indicador de equilibrio en el árbol (campo equi), se analizará si es necesaria alguna operación de equilibrio, de er así se lleva adelante para dejar el árbol en equilibrio AVL.

A diferencia del algoritmo de inserción AVL en el que a lo sumo se utilizaba una rotación por *operación de inserción*, en el borrado es posible que se necesite una rotación *por cada nodo del camino de búsqueda al nodo a eliminar*.

Las Rotaciones que se emplean dentro del algoritmo de borrado, son las mismas que se analizaron en el apartado "Análisis de la Inserción de Valores en un Árbol AVL", solo que posiblemente se utilizarán con mayor frecuencia.

A continuación, se tratará un ejemplo de eliminación sucesiva a partir de un árbol AVL, indicando el nodo a eliminar con una marca.

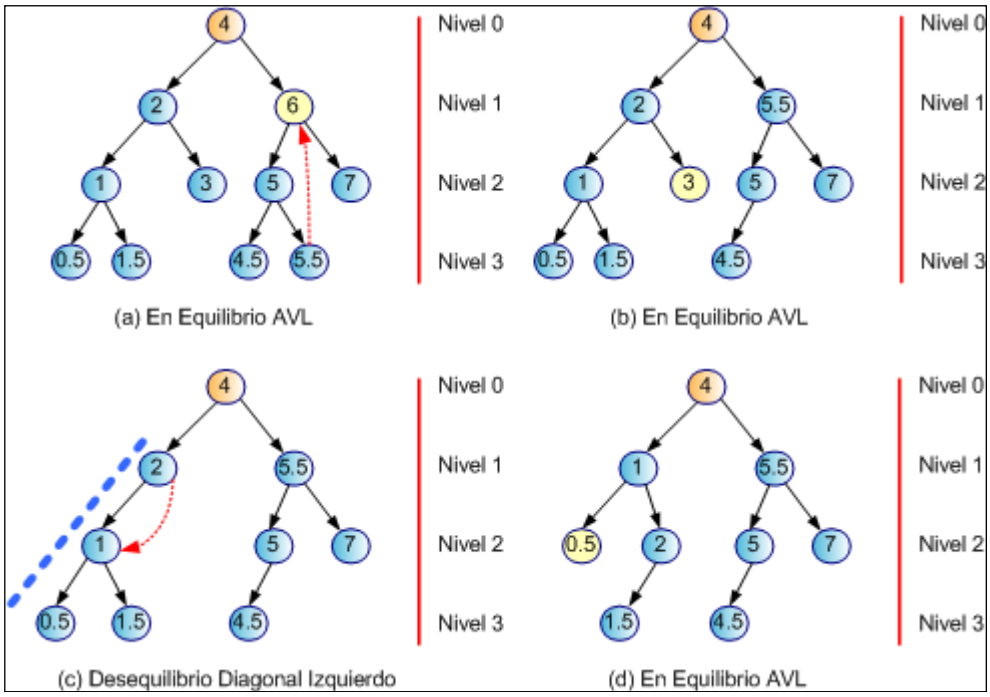


Figura 117: "Análisis del algoritmo de Borrado en un Árbol AVL (1 de 3)"

Note que en la etapa "c", una vez borrado el nodo 3, es necesario un reordenamiento diagonal izquierdo para reequilibrar el árbol AVL.

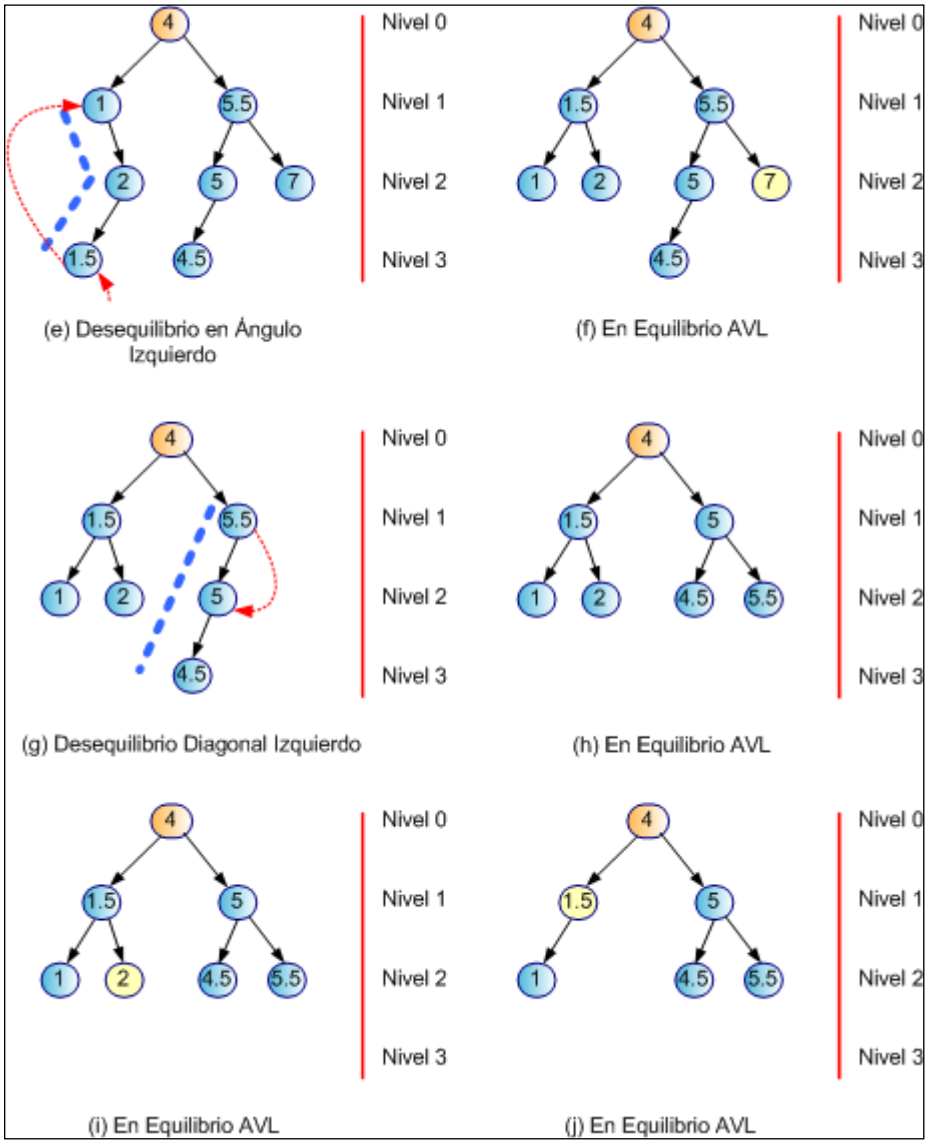


Figura 118: "Análisis del algoritmo de Borrado en un Árbol AVL (2 de 3)"

Vea que luego de eliminar el valor "0.5" en la etapa "d", se hizo necesaria una rotación doble para restablecer el equilibrio AVL. Análogamente al eliminar el valor 7, se hace necesaria una rotación simple para restablecer el equilibrio AVL.

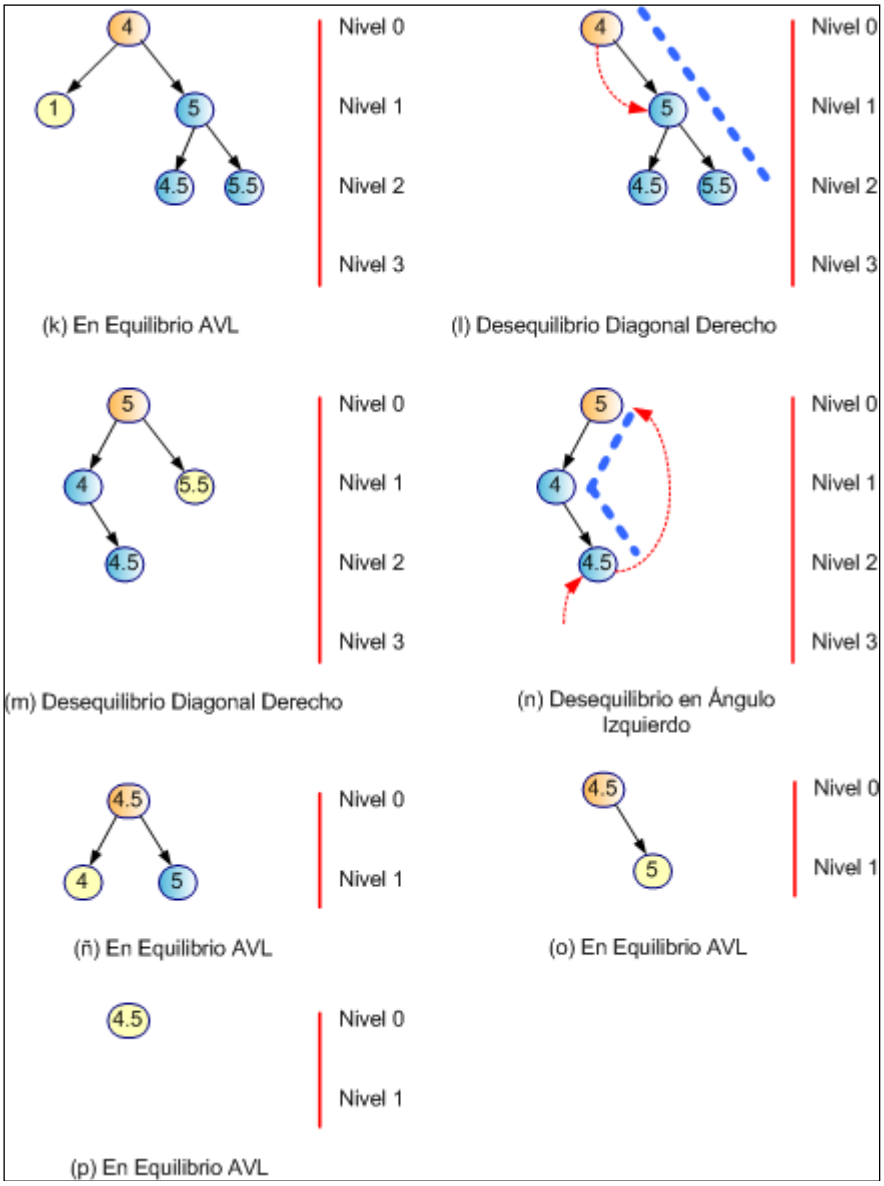


Figura 119: "Análisis del algoritmo de Borrado en un Árbol AVL (3 de 3)"

El Algoritmo de borrado AVL hace justamente lo descrito gráficamente, elimina al mejor estilo árbol binario, luego verifica que se mantenga el equilibrio a través del campo especial para tal fin, al que se denominó "equi". En caso de que el árbol se encuentre en desequilibrio AVL, el algoritmo de borrado efectúa las operaciones necesarias para llevarlo nuevamente a un estado de equilibrio AVL.

En el apartado posterior, se muestra el programa "avl.cpp" quien implementa los algoritmos de inserción y borrado en árboles AVL.

Implementación de un Árbol AVL en C/C++ Estructurado

A continuación se muestra el código estructurado que implementa los algoritmos para el manejo de árboles AVL, los cuales se adaptaron al paradigma estructurado a partir del programa orientado a objetos que implementa árboles AVL a través de la clase árbol, propiedad del Ing. Valerio Fritelli.

Archivo: **"avl.cpp"**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       enum bool {false, true};
5       struct nodo{
6               int info;
7               int equi;
8               nodo* izq, *der;
9               nodo(int x, int e, nodo *i, nodo *d)
10              {
11               info=x;
12               equi=e;
13               izq=i;
14               der=d;
15              }
16             };
17
18      void insertar(nodo *&,int);
19      void borrar(nodo*&, int);
20      void insavl(nodo *&, int,bool&);
21      void sacar(nodo *&, int, bool&);
22      void bor(nodo*&,bool &, nodo *&);
23      void equilibrar1(nodo*&,bool&);
24      void equilibrar2(nodo*&,bool&);
25      void mostrar(nodo *);
26
27      void main(void)
28      {
29       nodo* raiz=NULL;
30
31       clrscr();
32       insertar(raiz,96);
33       insertar(raiz,76);
34       insertar(raiz,56);
35       insertar(raiz,46);
36       insertar(raiz,66);
37       insertar(raiz,36);
38       mostrar(raiz);
39
40       cout<<"\n\n";
```

```
41      borrar(raiz,96);
42      cout<<"\n";
43      mostrar(raiz);
44      borrar(raiz,76);
45      cout<<"\n";
46      mostrar(raiz);
47      borrar(raiz,56);
48      cout<<"\n";
49      mostrar(raiz);
50      borrar(raiz,46);
51      cout<<"\n";
52      mostrar(raiz);
53      borrar(raiz,66);
54      cout<<"\n";
55      mostrar(raiz);
56      borrar(raiz,36);
57     }
58
59     void mostrar(nodo* p)
60     {
61      if(p==NULL) return;//arbol vacio
62
63      cout<<p->info<<".";
64      mostrar(p->izq);
65      mostrar(p->der);
66     }
67
68     void insertar(nodo *&raiz,int x)
69     {
70      bool h=false; //h=false implica que la altura no creci¢
71      insavl(raiz,x,h);
72     }
73     void insavl(nodo *&p, int x,bool& h)
74     {
75      nodo *p1, *p2;
76
77      if(p==NULL)
78      {//laclave no existe, insertarla
79       p=new nodo(x,0,NULL,NULL);
80       h=true;
81      }
82      else
83      {
84       if(x<p->info)
85        {
86         insavl(p->izq,x,h);
87
88         if(h==true)//la rama izquierda crecio
89         {
90          switch(p->equi)
```

```
91          {
92           case 1:
93                    p->equi=0;
94                    h=false;
95                    break;
96           case 0:
97                    p->equi=-1;
98                    break;
99           case -1:  //reequilibrar
100                   p1=p->izq;
101
102                   if(p1->equi==-1)
103                    {//izquierda simple
104                     p->izq=p1->der;
105                     p1->der=p;
106                     p->equi=0;
107                     p=p1;
108                    }
109                   else
110                    {//derecha doble
111                     p2=p1->der;
112                     p1->der=p2->izq;
113                     p2->izq=p1;
114                     p->izq=p2->der;
115                     p2->der=p;
116
117                     if(p2->equi==-1) p->equi=1;
118                      else p->equi=0;
119
120                     if(p2->equi==1) p1->equi=-1;
121                      else p1->equi=0;
122
123                     p=p2;
124                    }
125                   p->equi=0;
126                   h=false;
127                   break;
128         }//fin del switch
129        }//fin del if que contiene solo al switch
130       }//fin de if(x<p->info)
131      else
132       if(x>p->info)
133       {
134        insavl(p->der,x,h);
135
136        if(h==true)
137          switch(p->equi)
138          {
139            case -1: p->equi=0;
```

```
140                     h=false;
141                     break;
142            case 0: p->equi=1;
143                    break;
144            case 1: //reequilibrar
145                    p1=p->der;
146                    if(p1->equi==1)
147                    {//derecha simple
148                      p->der=p1->izq;
149                      p1->izq=p;
150                      p->equi=0;
151                      p=p1;
152                    }
153                    else
154                    {//izquierda doble
155                      p2=p1->izq;
156                      p1->izq=p2->der;
157                      p2->der=p1;
158                      p->der=p2->izq;
159                      p2->izq=p;
160
161                      if(p2->equi==1) p->equi=-1;
162                       else p->equi=0;
163
164                      if(p2->equi==-1) p1->equi=1;
165                       else p1->equi=0;
166
167                      p=p2;
168                    }//fin izq doble
169
170                    p->equi=0;
171                    h=false;
172                    break;
173           }//fin del switch
174        }//fin de if(x>p->info)
175       else //la clave ya existe
176          h=false; //la altura no varia
177      }//fin del else => p!=NULL
178     }//fin de la funcion
179
180     void borrar(nodo*& raiz,int x)
181     {
182      bool h=false;
183      sacar(raiz,x,h);
184     }
185
186     void sacar(nodo *&p, int x, bool& h)
187     {
188      if(p==NULL)
189       {
```

```
190        cout<<"La Clave No Existe";
191        h=false;
192       }
193      else
194       if(x<p->info)
195        {
196         sacar(p->izq,x,h);
197         if(h==true) equilibrar1(p,h);
198        }
199       else
200        if(x>p->info)
201         {
202          sacar(p->der,x,h);
203          if(h==true) equilibrar2(p,h);
204         }
205        else
206         {
207          nodo* q=p;
208
209          if(q->der==NULL)
210           {
211            p=q->izq;
212            h=true;
213           }
214          else
215           if(q->izq==NULL)
216            {
217             p=q->der;
218             h=true;
219            }
220           else
221            {
222             bor(q->izq,h,q);
223             if(h==true) equilibrar1(p,h);
224            }
225
226          delete q;
227         }
228     }
229
230     void bor(nodo*&d,bool &h, nodo *&q)
231     {
232      if(d->der!=NULL)
233       {
234        bor(d->der,h,q);
235        if(h==true) equilibrar2(d,h);
236       }
237       else
238       {
```

```
239        q->info=d->info;
240        q=d;
241        d=d->izq;
242        h=true;
243       }
244     }
245     void equilibrar1(nodo*&p,bool&h)
246     {
247      nodo *p1,*p2;
248      int e1,e2;
249
250      switch(p->equi)
251      {
252       case -1: p->equi=0;
253                break;
254       case 0: p->equi=1;
255               h=false;
256               break;
257       case 1:
258               p1=p->der;
259               e1=p1->equi;
260               if(e1>=0)
261                  {
262                    p->der=p1->izq;
263                    p1->izq=p;
264                    if(e1==0)
265                     {
266                      p->equi=1;
267                      p1->equi=-1;
268                      h=false;
269                     }
270                    else
271                     {
272                      p->equi=0;
273                      p1->equi=0;
274                     }
275                    p=p1;
276                  }
277                else
278                  {
279                   p2=p1->izq;
280                   e2=p2->equi;
281                   p1->izq=p2->der;
282                   p2->der=p1;
283                   p->der=p2->izq;
284                   p2->izq=p;
285                   if(e2==1) p->equi=-1; else p->equi=0;
286                   if(e2==-1) p1->equi=1; else p1->equi=0;
287
288                   p=p2;
```

```
289                 p2->equi=0;
290                }
291               break;
292     }
293    }
294
295    void equilibrar2(nodo*&p,bool&h)
296    {
297     nodo *p1, *p2;
298     int e1, e2;
299
300     switch(p->equi)
301     {
302      case 1:
303            p->equi=0;
304            break;
305      case 0:
306            p->equi=-1;
307            h=false;
308            break;
309      case -1:
310            p1=p->izq;
311            e1=p1->equi;
312
313            if(e1<=0)
314             {
315              p->izq=p1->der;
316              p1->der=p;
317              if(e1==0)
318               {
319                p->equi=-1;
320                p1->equi=1;
321                h=false;
322               }
323              else
324               {
325                p->equi=0;
326                p1->equi=0;
327               }
328
329              p=p1;
330             }
331            else
332             {
333              p2=p1->der;
334              e2=p2->equi;
335              p1->der=p2->izq;
336              p2->izq=p1;
337              p->izq=p2->der;
```

```
338             p2->der=p;
339
340             if(e2==-1) p->equi=1; else p->equi=0;
341             if(e2==1) p->equi=-1; else p1->equi=0;
342
343             p=p2;
344             p2->equi=0;
345            }
346            break;
347     }//fin del switch
348    }
349
```

Árboles Multidireccionales

Introducción

Hasta el momento se han venido tratando aqellos árboles que solo disponían de una clave (*key*) por nodo, y a partir de allí se decidía, en caso de que el valor en *key* no fuese el deseado, si continuaba por el camino izquierdo ó derecho, a lo cual, era de público conocimiento que se disponía de dos caminos, por lo que el nombre con el que se reconocía a esta estructura especial de árbol, era "Binario".

Se analizó el caso de un árbol binario en su concepto puro, allí se resalto la dificultad natural que tenían frente al desequilibrio que se producía ante cierta secuencia de inserción de valores, lo cual dio origen a los árboles AVL.

El foco del presente apartado, serán aquellos árboles que disponen de más de una clave en un nodo dado; a estos tipos de árboles se los conoce como multidireccionales, ya que dispondrán de (llaves +1) camino por donde derivar en la búsqueda del valor deseado.

Aunque a primera vista parece algo complicado, la estructura de árbol multidireccional es muy utilizada en la gestión de búsqueda en almacenamientos externos, como ser Discos Rígidos, Redes de Almacenamiento, entre otros mecanismos.

Los Aspectos vinculados a conceptos previos y/o rendimientos de los árboles en la gestión de datos en almacenamientos externos, se abordará en el siguiente apartado.

Conceptos Previos

En un árbol de búsqueda multidireccional, se denomina "***orden del árbol***", a la cantidad máxima de subárboles que en un nodo se pueden almacenar. La Cantidad de Claves que un nodo puede contener será una menos que la cantidad de subárboles, en otras palabras: "Número de Subárboles - 1".

Entonces en un árbol multidireccional de orden "n", dispondremos de s_0, s_1, s_2, s_3, … , $s_{(n-1)}$ subárboles y k_0, k_1, k_2, k_3, … , $k_{(n-2)}$ claves. Veamos lo dicho en el siguiente gráfico:

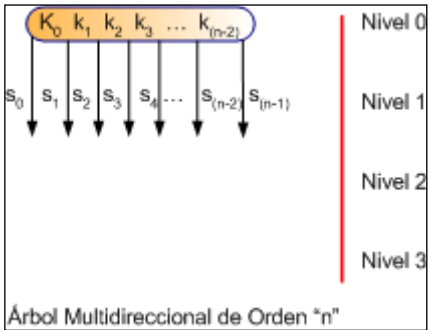


Figura 120: "Concepto de Claves y Subárboles en un Árbol Multidireccional"

Note que el subárbol $s_{(n-1)}$, conduce a otro nodo del árbol que contendá valores mayores a todos los presentes en el actual nodo, muchas veces se lo nombra como árbol de caída derecha o de caída superior. El Subárbol s_0, conduce a otros nodo del árbol que contendrá valores inferiores a todos los presentes en el actual nodo, por lo que se lo suele denominar árbol de caída inferior o izquierdo del nodo.

En forma análoga, note que entre k_0 y k_1 existe un subárbol, el mismo contendrá valores mayores a k_0 pero inferiores a k_1, el análisis es el mismo para los restantes subárboles de las siguientes claves. Así podemos decir que, s_1 es el subárbol izquierdo a la clave k_1, y que s_2 es el subárbol derecho a la misma clave.

Cuando un nodo se encuentra con la máxima cantidad de llaves, se lo denomina "***nodo ocupado***". Claro está que si el nodo posee la máxima cantidad de llaves que puede almacenar, también contedrá la máxima cantidad de subárboles factible de almacenamiento. Veamos un caso:

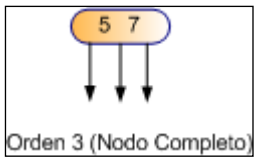


Figura 121: "Nodo Completo"

Note que el nodo de la figura anterior, considerando que sea de orden 3 el árbol, contiene los tres apuntadores a sus correspondientes subárboles, y las (Orden-1) claves, en otras palabras, las dos claves que completarían el nodo. A partir de este concepto de nodo ocupado, podemos decir que:

"Un árbol de búsqueda multidireccional de arriba hacia debajo de orden "n" es tal, si y solo si todos sus nodos son 'nodos ocupados' y, de no serlo, deben ser hojas"

Analicemos algunos ejemplos, para consolidar esta definición:

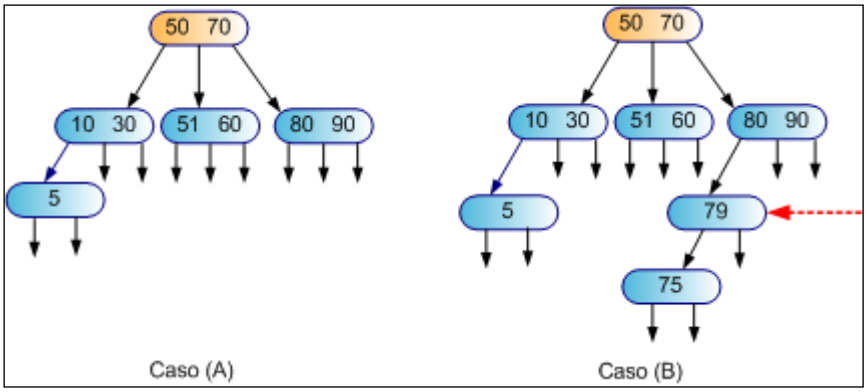


Figura 121: "Análisis de Caso de Árbol Multidireccional de Arriba hacia Abajo"

El caso "A", se puede decir que es un árbol de orden 3 de arriba haca abajo, ya que todos sus nodos poseen tres subárboles y dos claves y, los que no lo poseen, están situados como hojas.

En el caso "B", el árbol no puede decirs que sea de arriba hacia abajo, ya que posee un nodo no completo, que no se encuentra situado como hoja, éste es el caso del nodo que contiene la llave "79". Note que éste último nodo, posee dos subárboles, cuando el orden del árbol es 3 y además, posee subárbol inferior, lo que nos manifiesta que no es hoja, sino que se encuentra en el camino de una.

Si el nodo con la llave "75", no fuese hijo del nodo con la llave "79", es decir no existiese, el nodo con la llave 79 sería en realidad una hoja y, el caso "B" podría ser un

árbol de arriba hacia abajo, ya que si bien no estaría completo el nodo, se encontraría como una hoja en realidad.

Podemos definir como "***semihoja***", a un nodo con al menos un subárbol vació, el ejemplo de la figura anterior, el nodo que contenía la llave 79, era una semihoja, ya que un subárbol direccionaba al nodo con la llave "75", mientras que su otro subárbol se encontraba libre.

A partir de este concepto previo, podemos definir el siguiente aspecto de un árbol multidireccional:

"Se dice que un árbol está balanceado, si y solo si sus semihojas se encuentran en el último nivel del árbol"

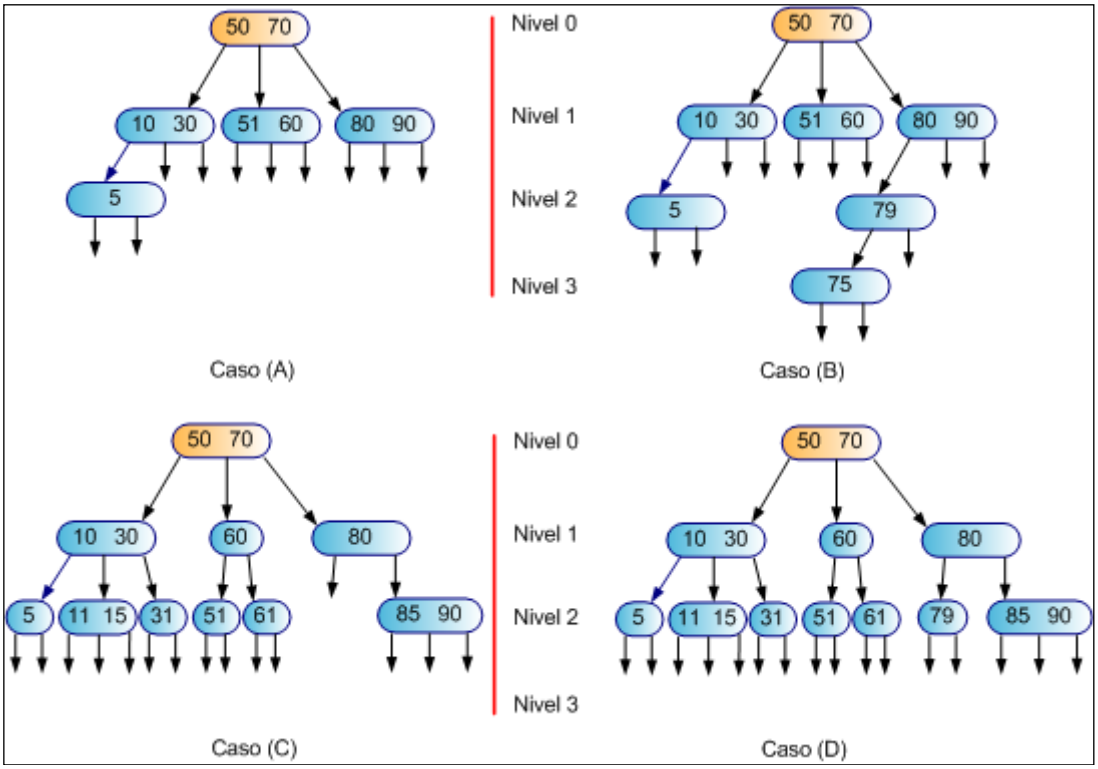


Figura 122: "Árboles Multidireccionales Balanceados"

Se Asume que los árboles de la figura anterior, corresponden al orden 3. Entonces, el Caso "A" claramente no es balanceado, ya que posee semihojas disponibles en el anteúltimo nivel.

El Caso "B", nuevamente posee semihojas disponibles en los niveles uno y dos, donde en realidad debieran estar todas en el tres para poder ser balanceado. Por ende el caso "B" no es un árbol multidireccional balanceado.

En el caso "C" casi llega a ser balanceado si no fuese por el nodo que contiene la llave "80" en el ante último nivel, el cual dispone de un subárbol disponible (semihoja).

Finalmente, el caso "D" si es un árbol multidireccional balanceado, ya que note que todas las semihojas (Recuerde que por semihoja se entiende que tiene libre al menos un subárbol), se encuentran en el último nivel.

Preferencia por el Orden

El aspecto del Orden en un árbol multidireccional merece un apartado por separado, ya que es lo que sustenta uno de los principales motivos de su empleo en los dispositivos externos como los discos rígidos.

Cuando un Árbol multidireccional es utilizado en dispositivos externos, la lectura de un *bloque* de datos del disco, representa la lectura continua y secuencial en disco y un nodo completo en el árbol.

A raíz de esto último, es que en un Árbol Multidireccional se prefiere que el orden del mismo (cantidad de claves por nodo) sea tan grande como se pueda (entre 400 claves aproximadamente en Sistemas de Gestión de Base de Datos) disminuyendo al mínimo posible el nivel del mismo. ¿Porqué? El acceso a un nodo en un nivel implica la lectura del bloque completo, por ende, cuanto mayor sea el nivel del árbol, mayor cantidad de lecturas mecánicas en disco se necesitan, lo que provoca un incremento de tiempo considerable. Usted se preguntará ¿si pero en memoria tengo 400 claves por ejemplo, en un nodo determinado? ¿Las busco secuencialmente? ¿No perdería mucho tiempo? Es cierto en parte dicho punto de vista, pero ocurre que la comparación de las claves en el nodo se efectúa en la memoria, con lo cual los tiempos de procesamiento son electrónics no mecánicos, en cuanto a magnitud total de tiempo, la diferencia es muy grande. Además del procesamiento electrónico, los árboles multidireccionales, implementan estructuras de búsqueda en memoria dentro de sus propios nodos como ser:

- Dispersión (tema que se tratará posteriormente en el presente capítulo),
- Árboles Binarios representados como arreglos,
- Entre otras estructuras.

Estas estructuras, a costa del empleo de una proporción adicional de memoria por nodo, mejoran los tiempos de respuesta ante la búsqueda en el nodo.

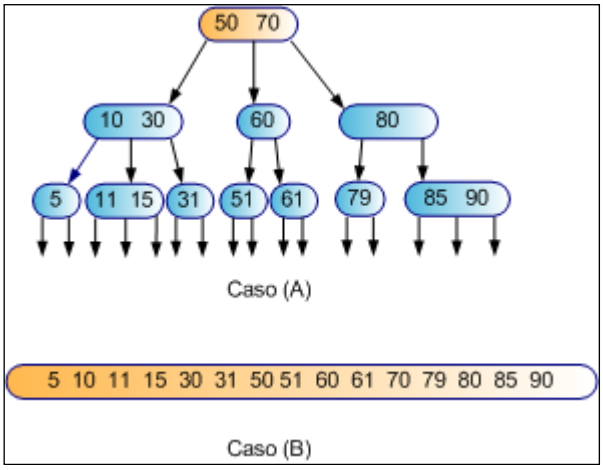


Figura 123: “Importancia del Orden en un Árbol Multidireccional”

En la figura anterior tenemos dos árboles distintos, ambos tienen 15 llaves almacenadas en su estructura, pero el árbol del Caso “A” es de orden 3, mientras que el árbol del caso “B” es de orden 16. Mire como impacta el orden en el acceso a disco, si usted tiene qu buscar la llave “90” en el disco, ¿Cuántos bloques leerá en el Caso A? tres, contra un solo bloque que se lee en el caso B. Dirá entonces ¿Si pero el caso B lee más información?,el problema de los discos y dispositivos de almacenamiento en

general, no es hacer la lectura del dato en sí, sino el movimiento del dispositivo de lectura. Note que en el caso "B" se efectúa un solom movimiento mecánico y se lee la información en un bloque y se lo almacena en un buffer de memoria para su tratamiento; mientras que en el caso "A" debe efectuar la misma operación tres veces para poder concretar el mismo objetivo.

Otro factor importante asociado a los Árboles multidireccionales y los dispositivos externos es el siguiente, ¿El registro con el dato lo almaceno en el mismo nodo o solo almaceno en éste, la dirección al registro de dato? La Respuesta a esta pregunta desencadena dos vertientes de almacenamiento:

- Almacenamiento Interno del Dato: es la situación en que se decide almacenar el dato de cada registro en el nodo, para aprovechar la lectura del bloque completo y cargar las claves y los datos,
- Almacenamiento Externo de Datos Direccionados: es la situación en que se decide almacenar el dato de cada registro en forma externa y guardar en éste la dirección física para obtenerlo (muchas veces a esta obtención a través de la memoria física, se lo denomina "Mapeo", ya que en realidad representa una coordenada dentro del dispositivo externo).

¿Cuál es la correcta? Depende de la situación y aplicación que se le de puntualmente al árbol. En el caso de Almacenamiento Interno se incrementa sustancialmente la cantidad de memoria electrónica requerida para hacer frente a la carga de datos, lo que tenga en cuenta que si se desea almacenar cuantrocientas claves por nodo, con registros de 10kb y cada nodo ocupando en encabezados (los campos enunciados, arreglo de punteros a subárboles, etc) unos 2418 bytes aproximadamente, necesitaríamos para montar el nodo en memoria unos 4002,35Kb. Tenga en cuenta que el tamaño de los registros varía notablemente dependiendo de la aplicación a la que se oriente una base de datos.

Los 2418 bytes aproximados del encabezado del nodo en memoria surgen de la estructura a analizar a continuación. Para el cálculo, se tiene en cuenta:

- (Cantidad de Claves (400) * Tamaño de un Entero) + Tamaño del Puntero a Entero. Esto es Equivalente a decir 400 * 2 bytes + 4 bytes = 804 bytes.
- (Orden del Árbol * Tamaño de un puntero Tipo Nodo) + Tamaño del Puntero al Arreglo de Punteros. Esto es equivalente a decir 401 * 4 bytes + 4 bytes = 1608 bytes.
- (Cantidad de Registros * Tamaño). Esto es equivalente a 400 * 10 kb = 4000 kb
- Las tres variables enteras del campo nodo, suman aproximadamente 6 bytes.

De esta forma, si se suman el espacio requerido por las llaves del nodo (804 bytes), más el requerido por los apuntadores de los subárboles (1608 bytes), más el requerido por los campos enteros del nodo (6 bytes) y el requerido por los campos de datos; el nodo requiere aproximadamente 4002,3613 Kbytes (3,91Mb). Como verá, el espacio por nodo no es un tema menor de análisis al momento de decidir si los datos residirán o no en el nodo. También es cierto que la contrapartida de esta técnica, sería soportar el incremento de tiempo que se produce por el mapeo a dsco en la búsqueda de los registros almacenados externamente.

Todo lo anterior se resume en un análisis costo/beneficio que el Desarrollador de Software de Base deberá contrastar al momento de la elección de la técnica a utilizar.

Estructura de Soporte al Nodo de un Árbol Multidireccional

La Estructura que da soporte a la implementación del árbol multidireccional a considerar en apartados posteriores, es la siguiente.

```
13      struct nodo{
14                  int *key;
15                  nodo **sons;
16                  int qArbolOcupados;
17                  int qKeysOcupadas;
18                  int qKeysMaximas;
19
20                  nodo(const int orden,int keyInicial)
21                  {
22                   qKeysMaximas=(orden-1);
23                   qKeysOcupadas=0;
24                   qArbolOcupados=2;
25
26                   key=new int[(orden-1)];
27
28                   sons=new nodo*[orden];
29                   inicializaNodo(orden,(void**)sons,key);
30                   key[0]=keyInicial;//en la posici¢n a la izquierda, coloco
31                                     //el primer elemento
32                   qKeysOcupadas++;//incremento el n£mero de llaves en el nodo
33                  };
34              };
```

Figura 124: “Fracción del Código de Definición de la Estructura de Nodo, del programa ‘AMultdir.cpp’”

A primera vista se puede observar diferencias sustanciales con la implementación de un nodo en un árbol binario. Se Analizará cada una a continuación:

- El Campo que almacena la clave, es un puntero del tipo entero, ya que direccionará un arreglo dinámico de tamaño “N-1”, donde “N” será el orden del árbol multidireccional. Recuerde que si el árbol multidireccional era de orden 3 por ejemplo, el mismo podría almacenar como máximo, dos claves por nodo.
- El Campo que almacena la dirección de los subárboles, en realidad es un puntero a puntero del tipo nodo. ¿Qué significa esto? Que definido el orden de un árbol, el nodo tendrá como máximo un arreglo de tamaño ‘n’ (el orden), de punteros de tipo nodo, que apuntarán a los subárboles respectivos. Éste arreglo de punteros del tipo nodo, a su vez, estará direccionado por un puntero del tipo nodo, que es “sons”, de forma que por ejemplo sons[0], nos da la dirección del nodo inferior del nodo en curso.
- El campo “qArbolOcupados”, contendrá la cantidad de subárboles que actualmente se está direccionando desde el nodo actual. Este campo se actualiza

en cada eliminación o inserción, de forma tal de evitarel recálculo cada vez que se desee conocer este dato.

- El campo "qKeysOcupadas", contendrá la cantidad de llavesque actualmente está almacenando en el nodo actual. Este campo, al igual que "qArbolOcupados", se actualiza en cada inserción o eliminación, de forma tal de evitar el recálculo cada vez que se lo necesite.
- El campo "qKeyMaximas" almacena la máxima cantidad de llaves que puede almacenar el nodo. Note que implícitamente, también almacena la máxima cantidad de subárboles que pueden resguardarse en el nodo, ya que recuerde que la cantidad máxima de llaves en un nodo era (Orden-1), mientras que la cantidad máxima de subárboles en el nodo, estaba dado por el Orden del mismo.

La Estructura, emplea el inicializador por defecto de la misma con dos parámetros:

- El orden del árbol en el cual se insertará el nodo. Lo que automáticamente tendrá un impacto directo en el arreglo de llaves del nodo, el arreglo de punteros a los subárboles hijos y los campos enunciados anteriormente.
- La Llave Inicial del Nodo, a la cual se le agregarán sus dos primeros subárboles derecho e izquierdo.

El Inicializador de la estructura, facilita enormemente la tarea de generaciónde hojas, ya que permite abstraer la lógica de inicialización del nodo en el nodo mismo, sin mezclar la misma, con la lógica funcional de las rutinas del árbol multidireccional.

Recorrido de un Árbol Multidireccional

La implementación del recorrido de un árbol multidirecional, se caracteriza por efectuar una lectura en orden de las llaves que lo integran. El resultado que se persigue con esto último, es la lectura en orden de acuerdo a sus llaves, del árbol multidireccional. Existen técnicas que analizaremos posteriormente, por el cual se vinculan las hojas almacenadas externamente de un árbol multidireccional, lo que permite que para el mismo objetivo, en lugar de recorrer el árbol, se efectúe un recorrido secuencial por las hojas *con datos*, vinculadas entre sí (En esta última técnica, en los nodos se almacenan solo las llaves, y los datos en las hojas).

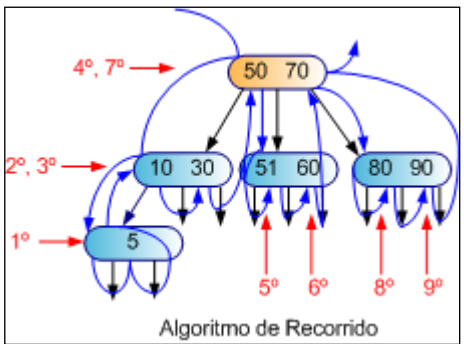


Figura 125: "Recorrido de un Árbol Multidireccional"

Note como el algoritmo de recorrido inicialmente localiza el menor elemento, luego sube y baja de nivel dinámicamente localizando los subsiguientes elementos en

orden y secuencia del nodo, hasta que retorna finalmente al nodo raíz luego de recorrer el subárbol superior de éste, culminando el recorrido.

El Algoritmo que ejecua esta acción, parte del programa "amultdir.cpp" es el siguiente:

```
163    void recorrido(nodo* p)
164    {
165     if(p!=NULL)
166      {
167       int nroArboles=p->qArbolOcupados;
168
169       for(int i=0;i<(nroArboles-1);i++)
170       {
171        recorrido(hijoDe(p,i));
172        cout<<"-"<<keyEn(p,i)<<"-";
173       }
174
175       recorrido(hijoDe(p,(nroArboles-1)));
176      }
177    }
```

Figura 126: "Fracción del Código que implementa el Algoritmo de Recorrido en un Árbol Multidireccional, del programa 'AMultdir.cpp'"

Búsqueda en un Árbol Multidireccional

La Búsqueda en un árbol multidireccional puede ser llevada adelante en forma recursiva, en forma análoga al algoritmo de recorrido, o bien en forma no recursiva, tal y como se analizará en el presente apartado.

El Algoritmo de Búsqueda implementadoe en el programa "AMuldir.cpp" es el siguiente:

```
134    nodo* buscar(nodo* arbol,int key)
135    {
136     nodo* p=arbol;
137
138     while(p!=NULL)
139     {
140      int i=buscarClaveEn(p,key);
141
142      if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
143      {
144       return p;
145      }
146
147      p=hijoDe(p,i);
148     }
149
150     return NULL;
151    }
152
```

```
153     nodo* hijoDe(nodo* p,int pos)
154     {
155      if(p==NULL)return NULL;
156      if(p->sons==NULL) return NULL;
157      if(pos<0 || pos>(p->qKeysMaximas+1)) return NULL;
158      if(p->sons[pos]==NULL) return NULL;
159
160      return p->sons[pos];
161     }
```

Figura 127: "Fracción del Código que implementa el Algoritmo de Búsqueda en un Árbol Multidireccional, del programa 'AMultdir.cpp'"

Hay funciones auxiliares del algoritmo que mercen ser explicadas para comprender el algoritmo de búsqueda, a saber:

- keyEn(nodo* p, int i): Retorna la llave almacenada en la posición "i" del nodo diseccionado con "p".
- buscarClaveEn(nodo* p, int key): Busca la *menor* posición j tal que "key" (el valor buscado) sea menor o igual a KeyEn(p,j).
- hijoDe(nodo* p, int i): Retorna un puntero a la posición del hijo de "p" direccionado por la posición "i" del arreglo d punteros. Es en realidad, el subárbol izquierdo de "p".

El Ciclo principal inicia con la dirección de la raíz del árbol, si la llave no existe en el nodo actual, la función buscarClaveEn, retorna la posición donde key<=KeyEn(p,j), por lo que se conoce porqué subárbol deberá descender al siguiente nodo. Una vez que descendió, efectúa la misma operatoria, buscará la posición "j" donde la llave buscada sea menor o igual a KeyEn(p,j). ¿Cuándo termina? En dos situaciones:

- La llave buscada coincide con KeyEn(p,j), lo que provoca que se retorne el nodo p (que es quien contiene a la llave en su posición "j").
- Cuando p es nulo, lo que indica que el valor no existe en el árbol.

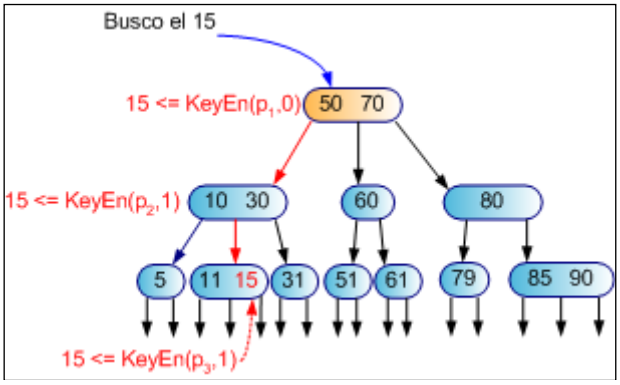


Figura 128: "Interpretación del Algoritmo de Búsqueda"

Note como al buscar el 15 en la primera iteración del algoritmo (donde p_1 significa primer valor de "p", p_2 significa segundo valor de "p" y así sucesivamente), el

resultado de buscarClaveEn devuelve el entero "0", en efecto 15 es menor o igual a la clave contenida en la posición cero del nodo y ésta última, es la menor llave (de las mayores a la buscada) del nodo completo. Con el valor devuelto por "buscarClaveEn", el algoritmo se mueve al hijo izquierdo y accede al subárbol izquierdo de 50.

Una vez que p asume el valor p_2 (apunta al subárbol izquierdo de 50), nuevamente pide de los mayores valores al buscado (o el buscado si existiese), devuelva la posición del menor, con lo que se retorna la posición 1, se desciende por el hijo izquierdo de "30" y, al efectuar la misma acción en el nodo p_3, el algoritmo encuentra el valor buscado en la posición 1 (recuerde que la cuenta inicia en cero).

La búsqueda que efectúa el algoritmo del presente apartado dentro de las claves del nodo, es una búsqueda secuencial implementada en la rutina "buscarClaveEn", se desarrollo a los fines de facilitar la comprensión inicial de la problemática de árboles multidireccionales. Sepa que es factible, como se mencionó anteriormente, implementar algoritmos más sofisticados de búsqueda dentro del nodo (intranodo), como ser árboles binarios equilibrados, dispersión, entre otros.

Algoritmo de Inserción

El Algoritmo de inserción implementado en el presente apartado, está adecuado para la construcción de un árbol de arriba hacia abajo, esto implica, que primero se llenarán los niveles superiores del árbol para recién luego generar nodos descendientes.

Se Analizará la lógica de inserción gráficamente, y se deja al lector que efectue una prueba de escritorio con el algoritmo correspodiente de inserción en el apartado "Implementación de un Árbol Multidireccional".

Suponga que el árbol a construir tiene un orden de 3 y la secuencia de valores a insertar en el mismo es la siguiente: 2, 5, 0, 1, 6, 70 y 3.

Hay algunas cuestiones previas al análisis gráfico como ser:

- Note que el Algoritmo de Inserción busca el nodo inicialmente con la función buscar, luego ubica la posición adecuada a insertar, y en caso de ser necesario efectua un corrimiento de las llaves pre existentes y sus punteros asociados, para dejatr la nueva llave en su posición definitiva,
- El Algoritmo de Inserción para árboles de arriba hacia abajo, solo crea nuevos nodos si y solo si, coexiste más espacio donde almacenar la nueva llave. Recuerde que en un árbol de arriba hacia abajo, solo las hojas eran nodos semi ocupados, mientras que aquellos nodos no hojas, debías ser nodos ocupados, en otras palabras, tener completamente ocupada su capacidad de llaves y subárboles,
- El Algoritmo inicialmente al crearse el nodo, carga la llave inaugural al mismo como así también inicializa sus dos primeros apuntadores a subárboles. Esta es la única situación en la que habilita dos subárboles por llave insertada. Las llaves que se inserten con posterioridad, solo provocarán que se habilite un puntero a subárbol adicional.

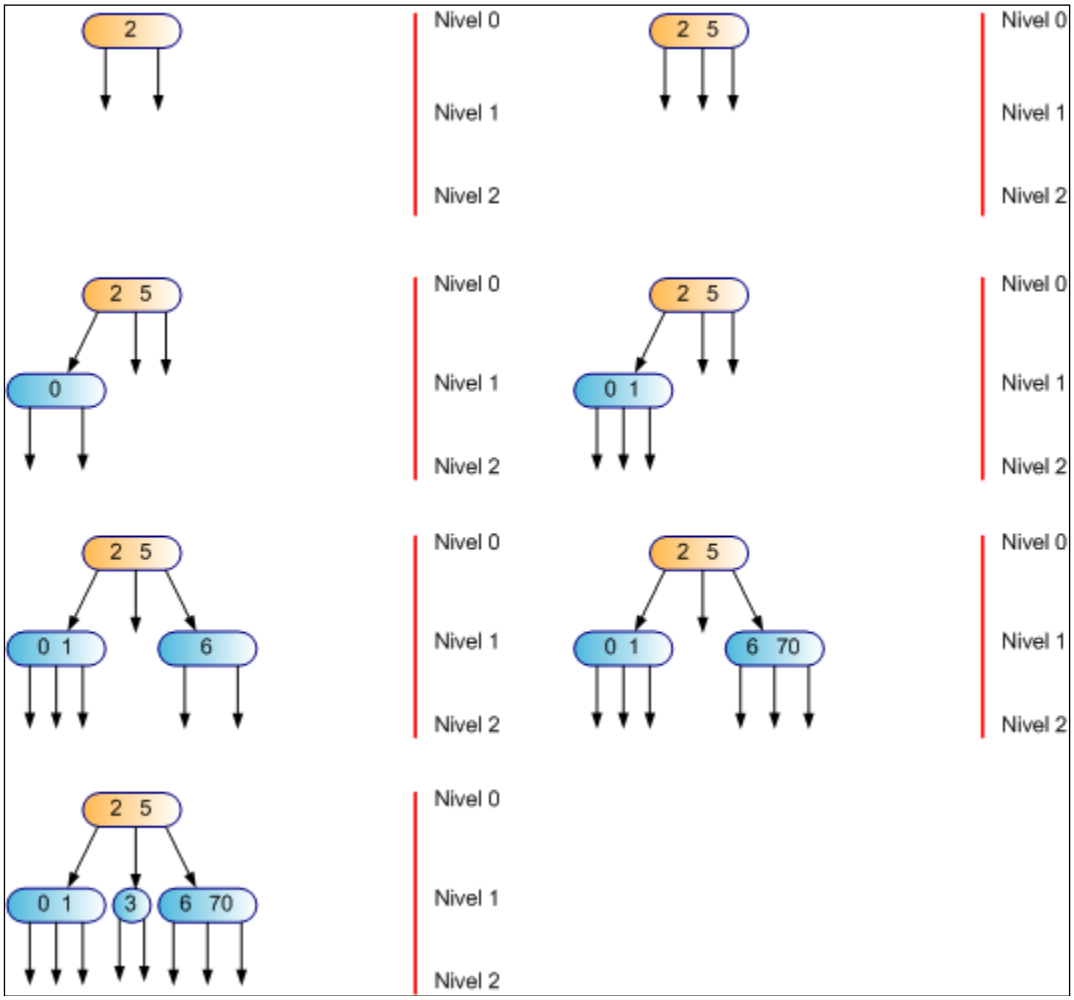


Figura 129: “Interpretación del Algoritmo de Inserción”

Note que al insertar la llave “2” y no existir árbol, inaugura el árbol y el nodo, habilitando para éste último dos punteros a subárboles. De todas formas no utilizará estos punteros a subárboles, hasta tanto no llene el nodo actual. La siguiente clave a insertar es la “5”, como su valor se inserta a la derecha de la existente, no es necesario un corrimiento a izquierda (distinto hubiese sido si la inserción fuese a la izquierda de 2, donde se deberían haber trasladado el 2 a la posición 1 del nodo, junto con sus punteros a subárbol).

A continuación se inserta la llave “0”, como detecta que el nodo actual se encuentra lleno, identifica por qué camino debe descender, entoces crea un nuevo nodo con sus respectivos subárboles y lo cuelga de la posición de descenso. Si presta atención al gráfico, el 0 generó un nuevo nodo con dos subárboles y se colgó del puntero a la izquierda de la llave “2” (en su nodo padre).

Al insertar la llave 1, tan solo se busca el nodo donde insertarla, como existe espacio solo se agrega, y al agregarse a la derecha no es necesario corrimiento de llaves y punteros. Volveremos sobre los corrimientos y veremos algunos ejemplos, en el algoritmo de eliminación.

Cuando se desea insertar la llave “6”, se identifica que la misma debiera ir colgando del subárbol superior de la ráiz, lo que provoca que se cree un nuevo nodo (ya

que no se dispone de espacio en la raíz) y se prenda/cuelgue del nodo derecho o de caída de su padre.

Al Insertarse el "70", no existen mayores inconvenientes, ya que en el nodo donde debiera ir, existe espacio y se inserta a la derecha. Finalmente, la llave 3, provoca que se genere un nuevo nodo, colgado del puntero a la izquierda de 5 (En efecto, 2<3<5).

Veamos ahora como trabajaría la lógica de eliminación para sostener los nodos no ocupados como hojas del árbol.

Algoritmo de Borrado

En la operación de borrados de llaves, en árboles multidireccionales, es común hablar de eliminación lógica o eliminación física. El Proceso de Eliminación Lógica de un nodo, consiste básicamente en localizar el nodo a insertar, y una vez encontrado, utilizando un campo adicional en la estructura (Ej: bool estaBorrado), marcarlo como eliminado. Los algoritmos de búsqueda y recorrido deberían tener en cuenta esta marca lógica a los fines de identificar si la llave está borrada o no.

El Proceso de Eliminación Física, consiste en un proceso análogo al anterior, solo que al localizar la llave en el nodo, se la elimina físicamente y si el nodo quedase sin llaves, se libera la memoria y se anulan los punteros del padre.

En muchas situaciones se prefiere un borrado lógico y posteriormente, se ejecutan actividades de "mantenimiento", donde se depuran las estructuras de datos. Imagine en un sistema de gestión de base de datos, cada vez que elimina un registro que deba hacer una escritura física y un reordenamiento, si fuese necesario, de los nodos remanentes del árbol. Esta última situación sería sumamente costosa en términos de eficiencia, recursos y tiempo del sistema, por lo que normalmente se prefiere sacrificar el espacio adicional que representa este campo lógico de eliminado por el del tiempo de procesamiento implicado en la reestructuración del árbol.

Ahora bien, en algún momento se deberá efectuar la eliminación física, de lo contrario estaríamos llenando de "basura" (datos inválidos o muertos) nuestro árbol, retrasando los procesos de búsqueda y corriendo el riesgo de corrupción de datos. Esta eliminación física es la que normalmente se lleva adelante con cierta periodicidad en los sistemas de gestión de base de datos para depurar sus índices.

La implementación del borrado lógico es sumamente sencillo por lo que no se aboradará en el presente capítulo. Con lo dicho anteriormente y la implementación de árboles multidireccionales del último apartado, es suficiente para que el lector sin mayores problemas pueda implementarla. Tenga en cuenta que solo debe agregarun campo por llave del tipo "Bit" (0 o 1) que indique si la llave está borradao activa nada más, y modificar levemente los algoritmos mencionados.

En cambio, se implementará el mecanismo de eliminación físico en memoria de un árbol multidireccional. Este mecanismo es mucho más complejo que la eliminación lógica, ya que supone procesos de corrimiento de llaves cuando sea necesario como de compactación de nodos cuando se lo necesite.

A continuación se analizará la lógica funcional de trabajo del algoritmo de borrado de árboles multidireccionales, con el objeto de tratar de no alterar la reglamentación impuesta por la definición de "Árboles de Arriba hacia abajo".

La Secuencia de Eliminación a utilizar será 5, 2, 0, 3, 1, 6 y 70.

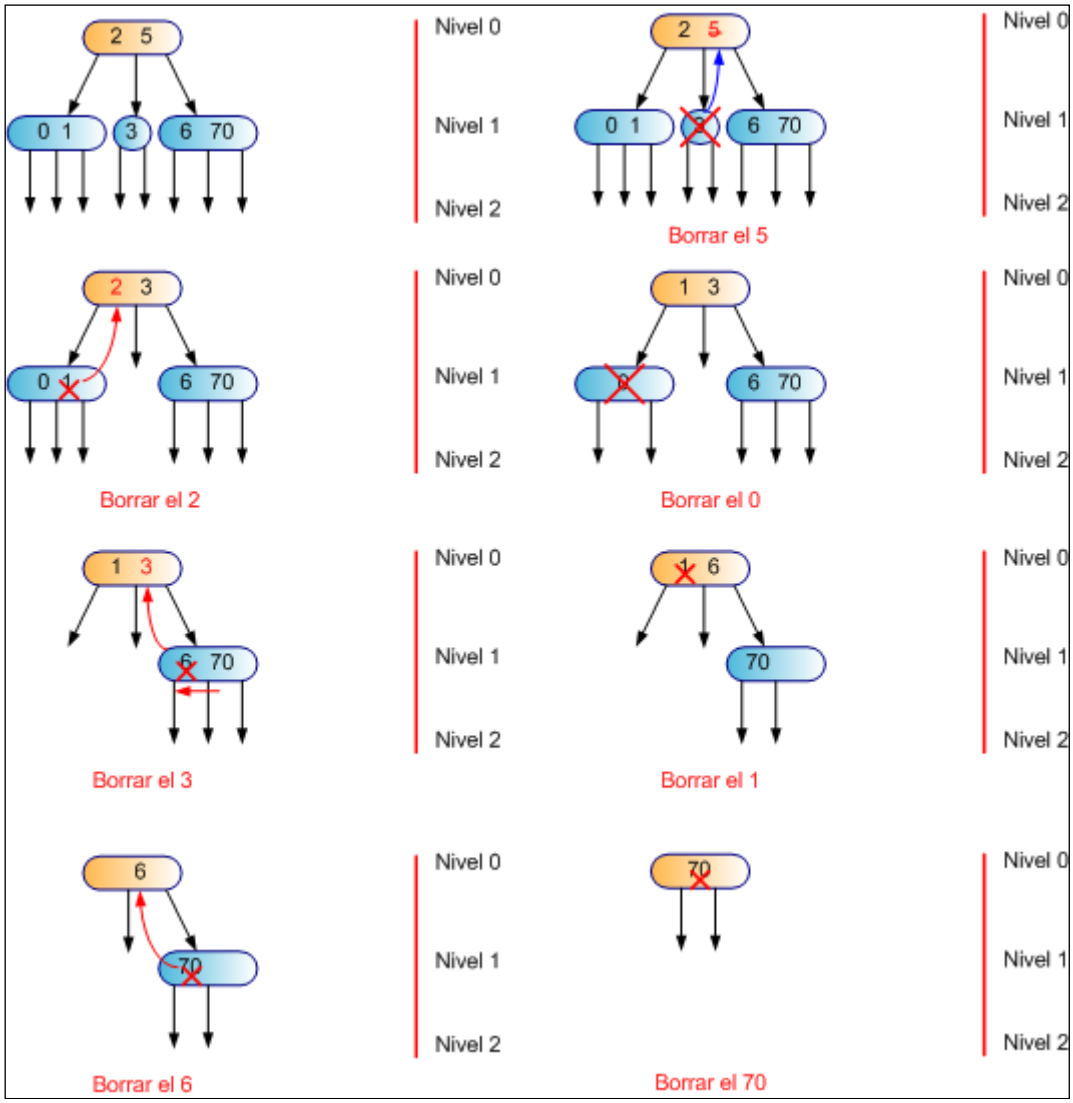


Figura 130: "Interpretación del Algoritmo de Borrado"

Del Algoritmo de Borrado básicamente note estos puntos:

- Cuando la llave a eliminar tiene hijo izquierdo, se busca el mayor para reemplazarlo en su lugar y en caso de que el mayor, hay sido la última llave en la hoja, se elimina la misma llevando el hijo izquierdo de la hoja (si tuviese) como hija del padre del nodo a eliminar. (Esta situación puede verla cuando se elimina el 6).
- En caso de que elimine una llave que, al no tener hija izquierda busque el menor valor a la derecha, puede que sea necesario un corrimiento en el nodo de donde se extrae el menor valor de los mayores para mantener el direccionamiento de los punteros y la cantidad de llaves ocupadas de dicho nodo. (Esta Situación puede verla al eliminarse la llave 3).

Este Algoritmo funciona muy bien en esas situaciones, pero existe una situación que queda por analizar, la que dejaremos para apartados posteriores. Esta situación que queda analizar es aquella donde al eliminar una llave de un nodo intermedio, éste no

tiene hijos izquierdo o derecho, por ende solo elimina la llave y se produce el corrimiento, pero debiera existir una fución auxiliar donde se produzca una rotación de llaves de cara a mantener como nodos ocupados a los intermedios y no ocupados a las hojas. Esta Situación es la que se dio al eliminar el 1, donde para mantener el nodo raiz completo se debería de haber subido el 70 a la raiz y eliminado el nodo del nivel 1.

Implementación de un Árbol de Búsqueda Multidireccional

A continuación se enuncia el programa completo "amultdir" con el algoritmo de borrado físico, en apartados posteriores se abordará nuevamente el algoritmo de borrado para incorporar la rotación de clave y manutención de nodos ocupados en la parte superior del árbol.

Archivo: **"amultdir.cpp"**

```
1       //Implementacion de un Arbol Multidireccional
2       //Antesala de Arboles B
3
4       #include <iostream.h>
5       #include <conio.h>
6
7       const short int VACIO=-1;
8       const short int true=1;
9       const short int false=0;
10
11      void inicializaNodo(int,void**,int*);
12
13      struct nodo{
14                  int *key;
15                  nodo **sons;
16                  int qArbolOcupados;
17                  int qKeysOcupadas;
18                  int qKeysMaximas;
19
20                  nodo(const int orden,int keyInicial)
21                  {
22                   qKeysMaximas=(orden-1);
23                   qKeysOcupadas=0;
24                   qArbolOcupados=2;
25
26                   key=new int[(orden-1)];
27
28                   sons=new nodo*[orden];
29                   inicializaNodo(orden,(void**)sons,key);
30                   key[0]=keyInicial;//en la posici¢n a la izquierda, coloco
31                                     //el primer elemento
32                   qKeysOcupadas++;//incremento el n£mero de llaves en el nodo
33                  };
34              };
35
```

```
36      int nroArbolesEn(nodo*);
37      int keyEn(nodo*,int);
38      int buscarClaveEn(nodo*,int);
39      nodo* buscar(nodo*,int);
40      nodo* hijoDe(nodo*,int);
41      void recorrido(nodo*);
42      nodo* verLugarInsert(nodo*,int,int&,int&);
43      void insertarRegistro(nodo*,int,int);
44      nodo* crearNodo(int,int);
45      void insertarNodo(nodo*&, int,int);
46      void insertarHoja(nodo*,int,int);
47      void borrarNodo(nodo*&,int);
48      nodo* buscarPadre(int,int&,nodo*&);
49
50      void main(void)
51      {
52       clrscr();
53       int ordenArbol=3;//El Inicialzador de Arbol
54       nodo *raiz=NULL;
55       insertarNodo(raiz,ordenArbol,2);
56       insertarNodo(raiz,ordenArbol,5);
57       insertarNodo(raiz,ordenArbol,0);
58       insertarNodo(raiz,ordenArbol,1);
59       insertarNodo(raiz,ordenArbol,6);
60       insertarNodo(raiz,ordenArbol,70);
61       insertarNodo(raiz,ordenArbol,3);
62       recorrido(raiz);
63       borrarNodo(raiz,5);
64       cout<<"\n";
65       recorrido(raiz);
66       borrarNodo(raiz,2);
67       cout<<"\n";
68       recorrido(raiz);
69       borrarNodo(raiz,0);
70       cout<<"\n";
71       recorrido(raiz);
72       borrarNodo(raiz,3);
73       cout<<"\n";
74       recorrido(raiz);
75       borrarNodo(raiz,1);
76       cout<<"\n";
77       recorrido(raiz);
78       borrarNodo(raiz,6);
79       cout<<"\n";
80       recorrido(raiz);
81       borrarNodo(raiz,70);
82       cout<<"\n";
83       recorrido(raiz);
84      }
```

```
85
86      void inicializaNodo(int orden,void** voidsons, int* key)
87      {
88       nodo** sons=(nodo**)voidsons;
89
90       for(int i=0;i<orden;i++)
91       {
92         sons[i]=NULL;//anulo todos los punteros a hijo
93         if(i<(orden-1)) key[i]=VACIO;
94       }
95      }
96
97      int nroArbolesEn(nodo* p)
98      {
99       if(p==NULL) return 0;
100
101      return p->qArbolOcupados;
102     }
103
104     //Retorna la clave del nodo p en la posici¢n j
105     int keyEn(nodo* p, int j)
106     {
107      if(p==NULL) return -1;
108      if(j<0) return -1;
109      if(j>p->qKeysOcupadas) return -1;
110
111      return p->key[j];
112     }
113
114     //Retorna el entero j mas peque¤o, tal que key<= k(p,j), donde
115     //k(p,j) retorna la clave en la posicion j para el nodo p o
116     int buscarClaveEn(nodo* elnodo, int key)
117     {
118      if(elnodo==NULL) return (-1);
119
120      if(elnodo->qKeysOcupadas==0) return (-1);
121      if(elnodo->key==NULL) return (-1);
122
123      //Se hace una B£squeda Secuencial de la posicion "j", que
124      //contiene una Clave >= a key (o, key<=clave en pos. j)
125
126      for(int i=0;i<elnodo->qKeysOcupadas;i++)
127      {
128       if(key<=elnodo->key[i]) return i;
129      }
130
131      return elnodo->qKeysOcupadas;//es lo mismo que (numArbol-1)
132     }
133
134     nodo* buscar(nodo* arbol,int key)
```

```
135     {
136      nodo* p=arbol;
137
138      while(p!=NULL)
139      {
140       int i=buscarClaveEn(p,key);
141
142       if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
143       {
144        return p;
145       }
146
147       p=hijoDe(p,i);
148      }
149
150      return NULL;
151     }
152
153     nodo* hijoDe(nodo* p,int pos)
154     {
155      if(p==NULL)return NULL;
156      if(p->sons==NULL) return NULL;
157      if(pos<0 || pos>(p->qKeysMaximas+1)) return NULL;
158      if(p->sons[pos]==NULL) return NULL;
159
160      return p->sons[pos];
161     }
162
163     void recorrido(nodo* p)
164     {
165      if(p!=NULL)
166       {
167        int nroArboles=p->qArbolOcupados;
168
169        for(int i=0;i<(nroArboles-1);i++)
170        {
171         recorrido(hijoDe(p,i));
172         cout<<"-"<<keyEn(p,i)<<"-";
173        }
174
175        recorrido(hijoDe(p,(nroArboles-1)));
176       }
177     }
178
179     nodo* verLugarInsert(nodo* arbol,int key,int& posicion, int& encontrado)
180     {
181       nodo* q=NULL;
182       nodo* p=arbol;
183
```

```
184      int i=0;
185      while(p!=NULL)
186      {
187       i=buscarClaveEn(p,key);
188       q=p;
189
190       if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
191       {
192        encontrado=true;
193        posicion=i;
194        return p;
195       }
196
197       p=hijoDe(p,i);
198      }
199
200      encontrado=false;
201      posicion=i;
202      return q;
203    }
204
205    void insertarRegistro(nodo* s,int posicion,int key)
206    {
207     s->key[posicion]=key;
208     s->qKeysOcupadas++;//se ocupo una posicion del nodo
209     s->qArbolOcupados++;//se agrega un arbol mas
210    }
211
212    nodo* crearNodo(int orden,int key)
213    {
214     nodo* nuevonodo=new nodo(orden,key);
215
216     return nuevonodo;
217    }
218
219    void insertarNodo(nodo*& arbol, int orden,int key)
220    {
221     if(arbol==NULL)
222     {
223      arbol=crearNodo(orden,key);//crea el arbol
224      return;
225     }
226
227     int posicion=0;
228     int encontrado=false;
229     nodo* s=verLugarInsert(arbol,key,posicion,encontrado);
230
231     if(encontrado==true) return;//ya existe
232
233     if(nroArbolesEn(s)<orden)
```

```
234      {
235       insertarHoja(s,posicion,key);
236       return;
237      }
238
239      nodo*p=crearNodo(orden,key);
240      s->sons[posicion]=p;
241      return;
242     }
243
244     void insertarHoja(nodo *s, int posicion, int key)
245     {
246      for(int i=(s->qArbolOcupados-1);i>posicion;i--)
247      {
248       s->key[i]=s->key[i-1];
249       s->sons[i+1]=s->sons[i];
250       s->sons[i]=((i-1)>0)?s->sons[i-1]:NULL;
251      }
252
253      s->key[posicion]=key;
254      s->sons[posicion]=NULL;
255      insertarRegistro(s,posicion,key);
256     }
257
258     void borrarNodo(nodo*& arbol,int key)
259     {//Se implementa el borrado fisico del nodo
260      nodo* p=arbol;
261
262      nodo* res=buscar(p,key);
263
264      if(res==NULL)
265      {
266        cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
267        return;
268      }
269
270      int posicion=buscarClaveEn(res,key);
271
272      if(keyEn(res,posicion)!=key)
273      {
274        cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
275        return;
276      }
277
278      //Si se esta Aqui, es porque en el nodo direccionado por "res"
279      //en el lugar "posicion" se encuentra el valor a eliminar
280      nodo* hizq=hijoDe(res,posicion);
281
282      if(hizq!=NULL)
```

```
283     {//Busco el mayor descendiente derecho del hijo izquierdo
284      int a;
285
286      int   veoPadrePosMayor=posicion;
287      nodo* veoPadre=res;
288      nodo* veo=hizq;
289
290      while(veo!=NULL)
291      {//Encuentro el mayor descendiente izquierdo
292       int posMayor=buscarClaveEn(veo,key);
293
294       if(veo->sons[posMayor]==NULL)
295       {//Se Encontro el mayor descendiente
296          res->key[posicion]=veo->key[(posMayor-1)];
297          veo->qKeysOcupadas--;
298          veo->qArbolOcupados--;
299
300          if(veo->qKeysOcupadas>=1)
301          {//solo anulo la posicion de la antigua clave
302           veo->key[(posMayor-1)]=VACIO;
303           return;
304          }
305          else
306          {//Si nuestro mayor descendiente, era el £nico en el nodo
307           veoPadre->sons[veoPadrePosMayor]=veo->sons[0];
308           delete veo;
309           veo=NULL;
310           return;
311          }
312       }
313
314       veoPadre=veo;
315       veoPadrePosMayor=posMayor;
316       veo=hijoDe(veo,posMayor);
317      }//fin del while
318     }//fin de si hizq!=NULL
319
320
321     nodo* hder=hijoDe(res,(posicion+1));
322     if(hder!=NULL)
323     {//Busco el mayor descendiente derecho del hijo izquierdo
324      int   veoPadrePosMenor=posicion+1;
325      nodo* veoPadre=res;
326      nodo* veo=hder;
327
328
329      while(veo!=NULL)
330      {//Encuentro el mayor descendiente izquierdo
331       int posMenor=buscarClaveEn(veo,key);
332
```

```
333        if(veo->sons[posMenor]==NULL)
334        {//Se Encontro el mayor descendiente
335           res->key[posicion]=veo->key[(posMenor)];
336           veo->qKeysOcupadas--;
337           veo->qArbolOcupados--;
338
339           if(veo->qKeysOcupadas>=1)
340           {//solo anulo la posicion de la antigua clave
341            veo->key[(posMenor)]=VACIO;
342
343            //Efectuo un Corrimiento a Izquierda de las Claves Remanentes
344            int z=0;
345            for(;z<veo->qKeysOcupadas;z++)
346            {
347             veo->key[z]=veo->key[z+1];
348             veo->sons[z]=veo->sons[z+1];
349            }
350            veo->key[z]=VACIO;//El antiguo extremo derecho que se traslado
351            veo->sons[z]=veo->sons[z+1];//Corrimiento del Extremo Derecho
352
353            return;
354           }
355           else
356           {//Si nuestro mayor descendiente, era el £nico en el nodo
357            veoPadre->sons[veoPadrePosMenor]=veo->sons[0];
358            delete veo;
359            veo=NULL;
360            return;
361           }
362        }
363
364        veoPadre=veo;
365        veoPadrePosMenor=posMenor;
366        veo=hijoDe(veo,posMenor);
367       }//fin del while
368      }//fin de si hder!=NULL
369
370      //Si se llega a esta instancia => que hder=hizq=NULL
371      //Efectuo el Corrimiento a Izquierda Necesario
372      int posicionResEnPadre=0;
373      nodo* padreDeRes=buscarPadre(key,posicionResEnPadre,arbol);
374
375      int pz=posicion;
376      for(;pz<res->qKeysOcupadas;pz++)
377       {
378        res->key[pz]=res->key[pz+1];
379        res->sons[pz]=res->sons[pz+1];
380       }
381
```

```
382     res->key[pz]=VACIO;//El antiguo extremo derecho que se traslado
383     res->qKeysOcupadas--;
384     res->qArbolOcupados=(res->qKeysOcupadas==0)?0:res->qArbolOcupados-1;
385     res->sons[pz]=res->sons[pz+1];//Corrimiento del Extremo Derecho
386
387     if(res->qKeysOcupadas>0)
388      {
389        return;
390      }
391      else
392      {//Debo eliminar el nodo y anular el puntero del padre al mismo
393       if(padreDeRes!=NULL)
394        {//Si fuese NULL, res es la raiz del arbol
395         if(posicionResEnPadre!=-1)
396            padreDeRes->sons[posicionResEnPadre]=NULL;
397         else
398            arbol=NULL;//el padre era el puntero cabeza del arbol(main)
399        }
400
401       delete res;
402       res=NULL;
403      }
404
405    }
406
407    nodo* buscarPadre(int key,int& posicionPadre,nodo*& arbol)
408    {
409     nodo* p=arbol;
410     nodo* padre=arbol;
411     posicionPadre=buscarClaveEn(padre,key);
412
413     int i=0;
414     while(p!=NULL)
415     {
416      i=buscarClaveEn(p,key);
417
418      if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
419      {
420       //padre; es el actualmente direccionado
421       return padre;
422      }
423
424      posicionPadre=i;
425      padre=p;
426      p=hijoDe(p,i);
427     }
428
429     padre=NULL;
430     return padre;
431    }
```

Algoritmo de Rotación en Árboles Multidireccionales

Como se mencionó en el apartado anterior, existen casos en los que para mantener la condición de árbol de arriba hacia abajo (todos los nodos no ocupados sean hojas en sus correspondientes subárboles), deben de efectuarse una serie de rotaciones.

Analizamos el caso en que, cuando un clave a eliminar en un nodo determinado poseía subárbol izquierdo y/o derecho, no existían mayores inconvenientes ya que se extraía la mayor llave del subárbol izquierdo a la llave a eliminar ó, si se utilizaba el subárbol derecho, se extraía la menor de las mayores llaves; para colocarla en la posición de llave eliminada.

La Situación Compleja, se presentaba cuando la llave a eliminar, no disponía de hijo izquierdo o derecho y no era una hoja, sino un nodo que originalmente era ocupado. Esto nos lleva a intuir que para mantener la condición de árbol de arriba hacia abajo, alguna llave debía subirse para completar el nodo y re completarse sucesivamente hasta los niveles inferiores del árbol, de cara a mantener los nodos no ocupados como hojas.

El Algoritmo presentado en este apartado, resuelve la situación problemática, y garantiza que los nodos no ocupados estarán situados como hojas. El Algoritmo al que se denomino "Rotación", se dispara en el caso en que la llave eliminada en un nodo *no tenga subárboles*. Hay que destacar, que es una rutina de soporte para un caso especial de eliminación de árboles Multidireccionales.

La rutina "Rotación", emplea una rutina de soporte denominada "buscarClaveRotar"; ésta última tiene por objetivo, localizar la clave del árbol que reemplazará a aquella a ser eliminada, como así también al padre del nodo que la contiene.

Localizada la llave a eliminar, se tiene en cuenta que se está extrayendo la misma, de un nodo que podría no ser una hoja y haber estado completo; por lo que se debe reinvocar la función rotación, luego del primer re orden, sobre el nodo del que se extrajo la clave de manera de garantizar su completitud. La función "Rotación" se irá re invocando recursivamente tantas veces haga falta para garantizar la completitud de los nodos intermedios del árbol.

A partir de una situación simple en el borrado del apartado anterior, que desencadenó "El inicio" de esta compleja situación, se explicará la lógica de rotación.

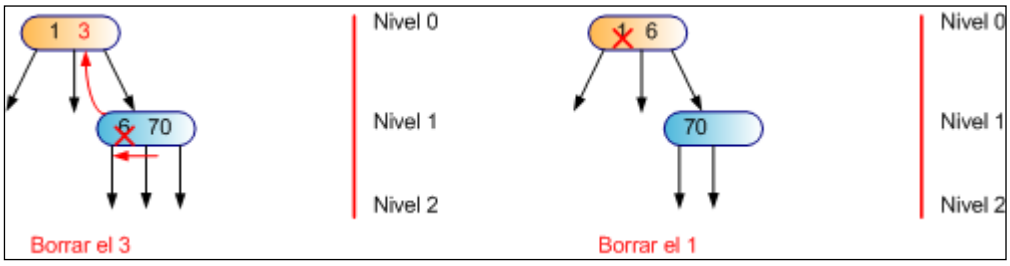


Figura 131: "Detección de la Situación Problemática"

La Situación se detecta debido a que el nodo superior no está completo, esto implica que la cantidad de claves en el mismo es inferior a (Orden – 1). Detectada la situación, se elimina la clavey se produce el corrimiento normalmente, quedando la llave 6 en el nodo superior y la llave "70" en el inferior.

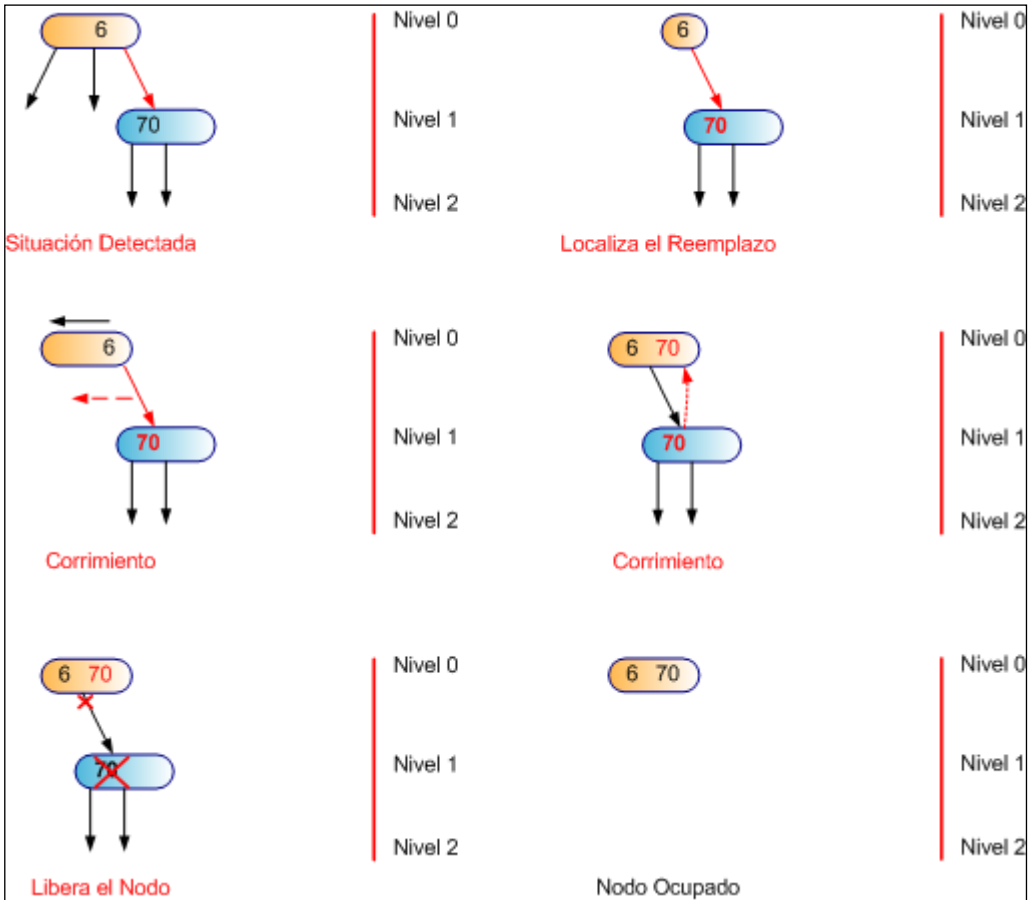


Figura 132: "Análisis de Rotación en el Subárbol Superior"

Cuando entra en acción el algoritmo de "rotación", identifica de derecha a izquierda, el primer subárbol no nulo del nodo principal; en nuestro ejemplo, el subárbol que cumple dicha condición es el derecho de la llave "6", en la posición 1 del arreglo "sons".

Detectado el subárbol a analizar, busca el menor de entre los mayores tomando como referencia la clave en la misma posición de descenso. Descenderá las veces que haga falta hasta encontrar el menor; el final del recorrido lo identifica a través del puntero de descenso igual a "NULL". Esto último ocurre con el 70, ¿Cómo sabe que el 70 es el menor de los mayores? Porque al tomar como referencia el 6 (Aquel que está a la izquierda del camino de descenso –solo por tratarse del árbol superior, porque si el subárbol fuese el inferior o uno intermedio, se toma como referencia el valor a la derecha del mismo-), se sabe que el menor de los mayores a 6, en el nodo actual es el 70, con lo que se retorna la posición "0", pero al intentar descender por sons[0], nos encontramos con que es "nulo", lo que nos indica que 70 es el menor disponible y candidato a ocupar el lugar ausente del nodo principal, junto con el 6.

Se agrega espacio al final del nodo que contiene al 6, para una clave y subárbol que completen el nodo, y se copia el 70 en esta posición adicional. Note que en ese preciso momento el 70 se encuentra duplicado en el árbol. Antes de borrar el 70, se fija si el nodo desde donde provino el 70 posee claves y subárboles adicionales, de tenerlos, provoca un corrimiento a la izquierda del mismo y, luego de completar al nodo superior, generará una nueva invocación de la rutina "Rotación" que trabajará sobre el nodo que contenía el 70 (solo si tuviese más claves). Como en nuestro caso, la llave "70" era la

única en el nodo, se libera la memoria y se anula el puntero del padre que direccionala a la misma.

En esta situación sencilla enunciada anteriormente, la rotación nos permitiría dejar el nodo ocupado en las posiciones superiores, y no en las hojas.

Si el subárbol no nulo estuviese no en el extremo superior como en el caso anterior, sino que fuese uno intermedio o inferior, se toma como referencia el valor a la derecha del subárbol de descenso y, se buscará el menor de los mayores. El comportamiento es análogo al anteriomente enunciado, solo que en lugar de producir un corrimiento a la izquierda, se produce un corrimiento a la derecha, dejando libre una posición inferior o intermedia, para que pueda subirse la mayor de las llaves. Una vez que se subió la llave (al subirse no es necesario corrimientos en los nodos inferiores, ya que tomo la mayor llave), se reinvoca la rutina "Rotación" desde el nodo de donde proviene la llave rotada.

Así trabaja el algoritmo para cada nodo que interviene en el trayecto de rellenamiento, hasta alcanzar el nivel de la hoja lo que corta la rutina.

A continuación, se lista el nuevo programa "amultdir.cpp", actualizado con la rutina de rotación y sus auxiliares:

Archivo: **"amultdir.cpp"**

```
1       //Implementacion de un Arbol Multidireccional
2       //Antesala de Arboles B
3
4       #include <iostream.h>
5       #include <conio.h>
6
7       const short int VACIO=-1;
8       const short int true=1;
9       const short int false=0;
10
11      void inicializaNodo(int,void**,int*);
12
13      struct nodo{
14                  int *key;
15                  nodo **sons;
16                  int qArbolOcupados;
17                  int qKeysOcupadas;
18                  int qKeysMaximas;
19
20                  nodo(const int orden,int keyInicial)
21                  {
22                   qKeysMaximas=(orden-1);
23                   qKeysOcupadas=0;
24                   qArbolOcupados=2;
25
26                   key=new int[(orden-1)];
27
28                   sons=new nodo*[orden];
29                   inicializaNodo(orden,(void**)sons,key);
```

```
30                 key[0]=keyInicial;//en la posici¢n a la izquierda, coloco
31                                   //el primer elemento
32                 qKeysOcupadas++;//incremento el n£mero de llaves en el nodo
33                };
34            };
35
36     int nroArbolesEn(nodo*);
37     int keyEn(nodo*,int);
38     int buscarClaveEn(nodo*,int);
39     nodo* buscar(nodo*,int);
40     nodo* hijoDe(nodo*,int);
41     void recorrido(nodo*);
42     nodo* verLugarInsert(nodo*,int,int&,int&);
43     void insertarRegistro(nodo*,int,int);
44     nodo* crearNodo(int,int);
45     void insertarNodo(nodo*&, int,int);
46     void insertarHoja(nodo*,int,int);
47     void borrarNodo(nodo*&,int);
48     nodo* buscarPadre(int,int&,nodo*&);
49     void rotacion(nodo*&);
50     nodo* buscoClaveRotar(int,int&,nodo*,nodo*&,int&);
51
52
53     void main(void)
54     {
55      clrscr();
56      int ordenArbol=3;//El Inicialzador de Arbol
57      nodo *raiz=NULL;
58      insertarNodo(raiz,ordenArbol,2);
59      insertarNodo(raiz,ordenArbol,5);
60      insertarNodo(raiz,ordenArbol,0);
61      insertarNodo(raiz,ordenArbol,1);
62      insertarNodo(raiz,ordenArbol,6);
63      insertarNodo(raiz,ordenArbol,70);
64      insertarNodo(raiz,ordenArbol,3);
65      recorrido(raiz);
66      borrarNodo(raiz,5);
67      cout<<"\n";
68      recorrido(raiz);
69      borrarNodo(raiz,2);
70      cout<<"\n";
71      recorrido(raiz);
72      borrarNodo(raiz,0);
73      cout<<"\n";
74      recorrido(raiz);
75      borrarNodo(raiz,3);
76      cout<<"\n";
77      recorrido(raiz);
78      borrarNodo(raiz,1);
79      cout<<"\n";
```

```
80      recorrido(raiz);
81
82     /*
83      borrarNodo(raiz,6);
84      cout<<"\n";
85      recorrido(raiz);
86      borrarNodo(raiz,70);
87      cout<<"\n";
88      recorrido(raiz);*/
89     }
90
91     void inicializaNodo(int orden,void** voidsons, int* key)
92     {
93      nodo** sons=(nodo**)voidsons;
94
95      for(int i=0;i<orden;i++)
96      {
97        sons[i]=NULL;//anulo todos los punteros a hijo
98        if(i<(orden-1)) key[i]=VACIO;
99      }
100    }
101
102    int nroArbolesEn(nodo* p)
103    {
104     if(p==NULL) return 0;
105
106     return p->qArbolOcupados;
107    }
108
109    //Retorna la clave del nodo p en la posici¢n j
110    int keyEn(nodo* p, int j)
111    {
112     if(p==NULL) return -1;
113     if(j<0) return -1;
114     if(j>p->qKeysOcupadas) return -1;
115
116     return p->key[j];
117    }
118
119    //Retorna el entero j mas peque¤o, tal que key<= k(p,j), donde
120    //k(p,j) retorna la clave en la posicion j para el nodo p o
121    int buscarClaveEn(nodo* elnodo, int key)
122    {
123     if(elnodo==NULL) return (-1);
124
125     if(elnodo->qKeysOcupadas==0) return (-1);
126     if(elnodo->key==NULL) return (-1);
127
128     //Se hace una B£squeda Secuencial de la posicion "j", que
```

```
129     //contiene una Clave >= a key (o, key<=clave en pos. j)
130
131     for(int i=0;i<elnodo->qKeysOcupadas;i++)
132     {
133      if(key<=elnodo->key[i]) return i;
134     }
135
136     return elnodo->qKeysOcupadas;//es lo mismo que (numArbol-1)
137    }
138
139    nodo* buscar(nodo* arbol,int key)
140    {
141     nodo* p=arbol;
142
143     while(p!=NULL)
144     {
145      int i=buscarClaveEn(p,key);
146
147      if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
148      {
149       return p;
150      }
151
152      p=hijoDe(p,i);
153     }
154
155     return NULL;
156    }
157
158    nodo* hijoDe(nodo* p,int pos)
159    {
160     if(p==NULL)return NULL;
161     if(p->sons==NULL) return NULL;
162     if(pos<0 || pos>(p->qKeysMaximas+1)) return NULL;
163     if(p->sons[pos]==NULL) return NULL;
164
165     return p->sons[pos];
166    }
167
168    void recorrido(nodo* p)
169    {
170     if(p!=NULL)
171      {
172       int nroArboles=p->qArbolOcupados;
173
174       for(int i=0;i<(nroArboles-1);i++)
175       {
176        recorrido(hijoDe(p,i));
177        cout<<i<<":-"<<keyEn(p,i)<<"-\n";
178       }
```

```
179
180        recorrido(hijoDe(p,(nroArboles-1)));
181       }
182     }
183
184     nodo* verLugarInsert(nodo* arbol,int key,int& posicion, int& encontrado)
185     {
186       nodo* q=NULL;
187       nodo* p=arbol;
188
189       int i=0;
190       while(p!=NULL)
191       {
192        i=buscarClaveEn(p,key);
193        q=p;
194
195        if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
196        {
197         encontrado=true;
198         posicion=i;
199         return p;
200        }
201
202        p=hijoDe(p,i);
203       }
204
205       encontrado=false;
206       posicion=i;
207       return q;
208     }
209
210     void insertarRegistro(nodo* s,int posicion,int key)
211     {
212      s->key[posicion]=key;
213      s->qKeysOcupadas++;//se ocupo una posicion del nodo
214      s->qArbolOcupados++;//se agrega un arbol mas
215     }
216
217     nodo* crearNodo(int orden,int key)
218     {
219      nodo* nuevonodo=new nodo(orden,key);
220
221      return nuevonodo;
222     }
223
224     void insertarNodo(nodo*& arbol, int orden,int key)
225     {
226      if(arbol==NULL)
227      {
```

```
228      arbol=crearNodo(orden,key);//crea el arbol
229      return;
230     }
231
232     int posicion=0;
233     int encontrado=false;
234     nodo* s=verLugarInsert(arbol,key,posicion,encontrado);
235
236     if(encontrado==true) return;//ya existe
237
238     if(nroArbolesEn(s)<orden)
239     {
240      insertarHoja(s,posicion,key);
241      return;
242     }
243
244     nodo*p=crearNodo(orden,key);
245     s->sons[posicion]=p;
246     return;
247    }
248
249    void insertarHoja(nodo *s, int posicion, int key)
250    {
251     for(int i=(s->qArbolOcupados-1);i>posicion;i--)
252     {
253      s->key[i]=s->key[i-1];
254      s->sons[i+1]=s->sons[i];
255      s->sons[i]=((i-1)>0)?s->sons[i-1]:NULL;
256     }
257
258     s->key[posicion]=key;
259     s->sons[posicion]=NULL;
260     insertarRegistro(s,posicion,key);
261    }
262
263    void borrarNodo(nodo*& arbol,int key)
264    {//Se implementa el borrado fisico del nodo
265     nodo* p=arbol;
266
267     nodo* res=buscar(p,key);
268
269     if(res==NULL)
270     {
271       cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
272       return;
273     }
274
275     int posicion=buscarClaveEn(res,key);
276
277     if(keyEn(res,posicion)!=key)
```

```
278     {
279       cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
280       return;
281     }
282
283     //Si se esta Aqui, es porque en el nodo direccionado por "res"
284     //en el lugar "posicion" se encuentra el valor a eliminar
285     nodo* hizq=hijoDe(res,posicion);
286
287     if(hizq!=NULL)
288     {//Busco el mayor descendiente derecho del hijo izquierdo
289      int a;
290
291      int   veoPadrePosMayor=posicion;
292      nodo* veoPadre=res;
293      nodo* veo=hizq;
294
295      while(veo!=NULL)
296      {//Encuentro el mayor descendiente izquierdo
297       int posMayor=buscarClaveEn(veo,key);
298
299       if(veo->sons[posMayor]==NULL)
300       {//Se Encontro el mayor descendiente
301          res->key[posicion]=veo->key[(posMayor-1)];
302          veo->qKeysOcupadas--;
303          veo->qArbolOcupados--;
304
305          if(veo->qKeysOcupadas>=1)
306          {//solo anulo la posicion de la antigua clave
307           veo->key[(posMayor-1)]=VACIO;
308           return;
309          }
310          else
311          {//Si nuestro mayor descendiente, era el £nico en el nodo
312           veoPadre->sons[veoPadrePosMayor]=veo->sons[0];
313           delete veo;
314           veo=NULL;
315           return;
316          }
317       }
318
319       veoPadre=veo;
320       veoPadrePosMayor=posMayor;
321       veo=hijoDe(veo,posMayor);
322      }//fin del while
323     }//fin de si hizq!=NULL
324
325
326     nodo* hder=hijoDe(res,(posicion+1));
```

```
327     if(hder!=NULL)
328     {//Busco el mayor descendiente derecho del hijo izquierdo
329      int   veoPadrePosMenor=posicion+1;
330      nodo* veoPadre=res;
331      nodo* veo=hder;
332
333
334      while(veo!=NULL)
335      {//Encuentro el mayor descendiente izquierdo
336       int posMenor=buscarClaveEn(veo,key);
337
338       if(veo->sons[posMenor]==NULL)
339       {//Se Encontro el mayor descendiente
340          res->key[posicion]=veo->key[(posMenor)];
341          veo->qKeysOcupadas--;
342          veo->qArbolOcupados--;
343
344          if(veo->qKeysOcupadas>=1)
345          {//solo anulo la posicion de la antigua clave
346           veo->key[(posMenor)]=VACIO;
347
348           //Efectuo un Corrimiento a Izquierda de las Claves Remanentes
349           int z=0;
350           for(;z<veo->qKeysOcupadas;z++)
351           {
352            veo->key[z]=veo->key[z+1];
353            veo->sons[z]=veo->sons[z+1];
354           }
355           veo->key[z]=VACIO;//El antiguo extremo derecho que se traslado
356           veo->sons[z]=veo->sons[z+1];//Corrimiento del Extremo Derecho
357
358           return;
359          }
360          else
361          {//Si nuestro mayor descendiente, era el £nico en el nodo
362           veoPadre->sons[veoPadrePosMenor]=veo->sons[0];
363           delete veo;
364           veo=NULL;
365           return;
366          }
367       }
368
369       veoPadre=veo;
370       veoPadrePosMenor=posMenor;
371       veo=hijoDe(veo,posMenor);
372      }//fin del while
373     }//fin de si hder!=NULL
374
375     //Si se llega a esta instancia => que hder=hizq=NULL
376     //Efectuo el Corrimiento a Izquierda Necesario
```

```
377     int posicionResEnPadre=0;
378     nodo* padreDeRes=buscarPadre(key,posicionResEnPadre,arbol);
379
380     int pz=posicion;
381     for(;pz<res->qKeysOcupadas;pz++)
382      {
383       res->key[pz]=res->key[pz+1];
384       res->sons[pz]=res->sons[pz+1];
385      }
386
387     res->key[pz]=VACIO;//El antiguo extremo derecho que se traslado
388     res->qKeysOcupadas--;
389     res->qArbolOcupados=(res->qKeysOcupadas==0)?0:res->qArbolOcupados-1;
390     res->sons[pz]=res->sons[pz+1];//Corrimiento del Extremo Derecho
391
392
393     if(res->qKeysOcupadas>0)
394      {//Aqu¡ es donde debo subir una clave de izq o derecha
395       rotacion(res);
396       return;
397      }
398      else
399      {//Debo eliminar el nodo y anular el puntero del padre al mismo
400       if(padreDeRes!=NULL)
401        {//Si fuese NULL, res es la raiz del arbol
402         if(posicionResEnPadre!=-1)
403            padreDeRes->sons[posicionResEnPadre]=NULL;
404         else
405            arbol=NULL;//el padre era el puntero cabeza del arbol(main)
406        }
407
408       delete res;
409       res=NULL;
410      }
411
412    }
413
414    nodo* buscarPadre(int key,int& posicionPadre,nodo*& arbol)
415    {
416     nodo* p=arbol;
417     nodo* padre=arbol;
418     posicionPadre=buscarClaveEn(padre,key);
419
420     int i=0;
421     while(p!=NULL)
422     {
423      i=buscarClaveEn(p,key);
424
425      if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
```

```
426      {
427       //padre; es el actualmente direccionado
428       return padre;
429      }
430
431      posicionPadre=i;
432      padre=p;
433      p=hijoDe(p,i);
434     }
435
436     padre=NULL;
437     return padre;
438    }
439
440    //Funcion prototipo de rotacion de claves
441
442    void rotacion(nodo*& p)
443    {
444    //Localizo la Rama del Sub rbol distinta de NULL desde derecha a Izq
445     int posicion;
446     int ready=false;
447
448     for(posicion=(p->qArbolOcupados-1);posicion>=0 && !ready;posicion--)
449            if(p->sons[posicion]!=NULL)
450            {
451             ready=true;
452             posicion++;//incremento 1 porque luego viene posicion--, previo a
        evaluar la condicion
453            }
454
455     if(posicion<0) return;//es una hoja
456
457     nodo* padreARotar=p;//originalmente es el mismo nodo donde se elimina
458     nodo* arotar=NULL;
459     int claveARotar=-1;
460     int pospadre=posicion;//originalmente es por donde desciendo
461
462     if((posicion+1)==1)//es el subarbol inferior, implica corrimiento a la
        derecha
463      {
464       arotar=buscoClaveRotar(p->key[posicion],claveARotar,p-
        >sons[posicion],padreARotar,pospadre);
465       claveARotar--;//ya que retorna la pos del subarbol superior nulo
466      }
467
468     if((posicion+1)>p->qKeysOcupadas)
469     {
470      arotar=buscoClaveRotar(p->key[(p->qKeysOcupadas-1)],claveARotar,p-
        >sons[posicion],padreARotar,pospadre);
471      if(arotar==NULL) return;//no hay que rotar
```

```
472
473       p->qKeysOcupadas++;
474       p->qArbolOcupados++;
475
476       //Se agrega la clave al final del nodo
477       p->key[(p->qKeysOcupadas-1)]=arotar->key[claveARotar];
478
479       if(arotar->qKeysOcupadas==1)//pregunto antes de decrementar porque
        puede ser necesario alg£n corrimiento
480       {
481        arotar->qKeysOcupadas--;
482        arotar->qArbolOcupados--;
483        delete arotar;
484        arotar=NULL;
485        if(padreARotar!=NULL) padreARotar->sons[pospadre]=NULL;
486
487        p->sons[(p->qArbolOcupados-1)]=p->sons[(p->qArbolOcupados-2)];
488        p->sons[(p->qArbolOcupados-2)]=NULL;
489        return;//la clave rotada era el unico valor del nodo origen, fin.
490       }
491       //Si estoy aqu¡, hace falta un corrimiento a la izq. del nodo origen
492       //de la clave rotada
493
494       int corr=0;
495
496       for(;corr<p->qArbolOcupados;corr++)
497       {
498        arotar->sons[corr]=arotar->sons[corr+1];
499        if((corr+1)<p->qKeysOcupadas) arotar->key[corr]=arotar->key[corr+1];
500       }
501
502       arotar->sons[(p->qArbolOcupados-1)]=NULL;
503       arotar->key[(p->qKeysOcupadas-1)]=VACIO;
504       arotar->qArbolOcupados--;
505       arotar->qKeysOcupadas--;
506
507       rotacion(arotar);//el nodo en que se efectuo el corrimiento debe ser
        completado
508       return;
509      }//fin  del if que nos decia que estabamos parados en la puerta del
        subarbol superior
510      else
511      {//es un subarbol intermedio, sera necesario un corrimiento a la derecha
512       arotar=buscoClaveRotar(p->key[posicion],claveARotar,p-
        >sons[posicion],padreARotar,pospadre);
513       claveARotar--;
514      }
515
516      if(arotar==NULL) return;//no hay que rotar
```

```
517
518     p->qKeysOcupadas++;
519     p->qArbolOcupados++;
520
521     int corr=p->qArbolOcupados-1;
522
523     for(;corr>posicion;corr--)
524     {
525      if((corr-1)!=posicion)
526      {
527       p->sons[corr]=p->sons[corr-1];
528       p->key[corr-1]=p->key[corr-2];
529      }
530     }
531
532     p->sons[posicion+1]=NULL;
533     p->key[posicion]=arotar->key[claveARotar];
534     arotar->qKeysOcupadas--;
535     arotar->qArbolOcupados--;
536
537     if(arotar->qKeysOcupadas==0)
538     {
539      delete arotar;
540      arotar=NULL;
541      if(padreARotar!=NULL) padreARotar->sons[pospadre]=NULL;
542      return;
543     }
544
545     rotacion(arotar);//completo el nodo origen de la clave rotada
546    }
547
548    nodo* buscoClaveRotar(int key, int& posicion, nodo* bolsa, nodo*&
       padreARotar, int& pospadre)
549    {
550     nodo* p=bolsa;
551     nodo* aux=padreARotar;
552
553     while(p!=NULL)
554     {
555      int bajo=buscarClaveEn(p,key);
556
557      if(p->sons[bajo]==NULL)
558      {//encontre
559       padreARotar=aux;
560       posicion=bajo;
561       return p;
562      }
563
564      pospadre=bajo;
565      aux=p;
```

```
566        p=p->sons[bajo];
567       }
568
569       return p;// aqui p es NULL
570      }
```

Árboles B

Un Árbol Multidireccional Balanceado, en el que cada nodo no raíz contiene al menos (n-1)/2 claves (siendo "n" el orden del árbol) se denomina Árbol B de orden n. Tenga en cuenta que un árbol B de orden 'n', también se lo conoce por arbol n-(n-1) o un árbol (n-1)-n. El Árbol B tiene un orden de búsqueda de *O(log(n))*, lo que para búsquedas en dispositivos externos es para tener muy en cuenta a la hora de seleccionar el mecanismo de indexación.

Un árbol B posee la enorme bondad de disponer sus nodos medios llenos, salvo la raíz, lo que permite aprovechar con cierto grado de eficiencia los dispositivos de almacenamiento externo. Normalmente en este tipo de árboles, empleados para indexación masiva, no es frecuente realizar borrados físicos, ya que la operatoria que se generaría para re equilibrar el árbol, considerando que posiblemente el 90% del mismo se encuentre en disco, sería muy costosa en terminos de recursos. Este último factor, lleva a que sean frecuentes los borrados lógicos y, cada cierto período, la regeneración completa del índice.

Algoritmo de Inserción en Árboles B

Lo más potable en este tipo de algoritmos es, al menos inicialmente, analizar su lógica de inserción mediante un seguimiento gráfico de un árbol hipotético: Suponga la siguiente secuencia de inserción "2, 5, 0, 1, 6, 70, 3, 80, 90, 95, 99, 69 y 68" en un árbol de orden 3.

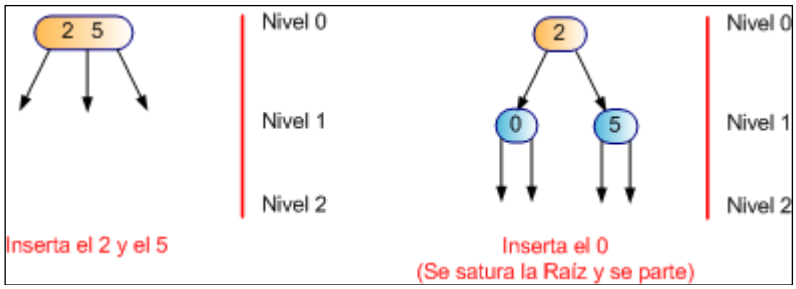


Figura 133: "Análisis de la Partición de la Raíz de un Árbol B"

Note que en un árbol B de orden dos, al insertar los dos primero valores, se completa el nodo raíz disponible y, cuando se desea insertar el valor "0", lo que provocaría que se sobrepase la capacidad del nodo, y en particular del raíz, se produce un quiebre que incrementará el nivel del árbol B. Este quiebre, provoca que el valor intermedio entre las claves actualmente en el nodo y el valor a insertar, sea colocado como el nuevo elemento de la raíz, generando dos nodos hijos (izquierdo y derecho) con los valores restantes, que pasarán a depender o "colgar" del nuevo nodo raíz. Este es el único caso donde se incrementa la altura de un árbol B, por ende se puede decir que:

"La altura de un Arbol B solo se incrementará, al momento en que la capacidad de almacenamientos de claves del nodo raíz, sea excedida"

En el ejemplo de la figura anterior, note que los valores pre existentes en el nodo eran el 2 y el 5, en ese orden. Al intentar insertarse el valor "0", los valores deberían ir en la siguiente secuencia "0, 2 y 5"; como el 2 es el valor intermedio en la secuencia, es tomado como referencia para formar la nueva raíz del árbol, mientras que los valores 0 y 5, a izquierda y derecha del 2 respectivamente, se toman como base para la conformación de los hijos izquierdo y derecho de la raíz.

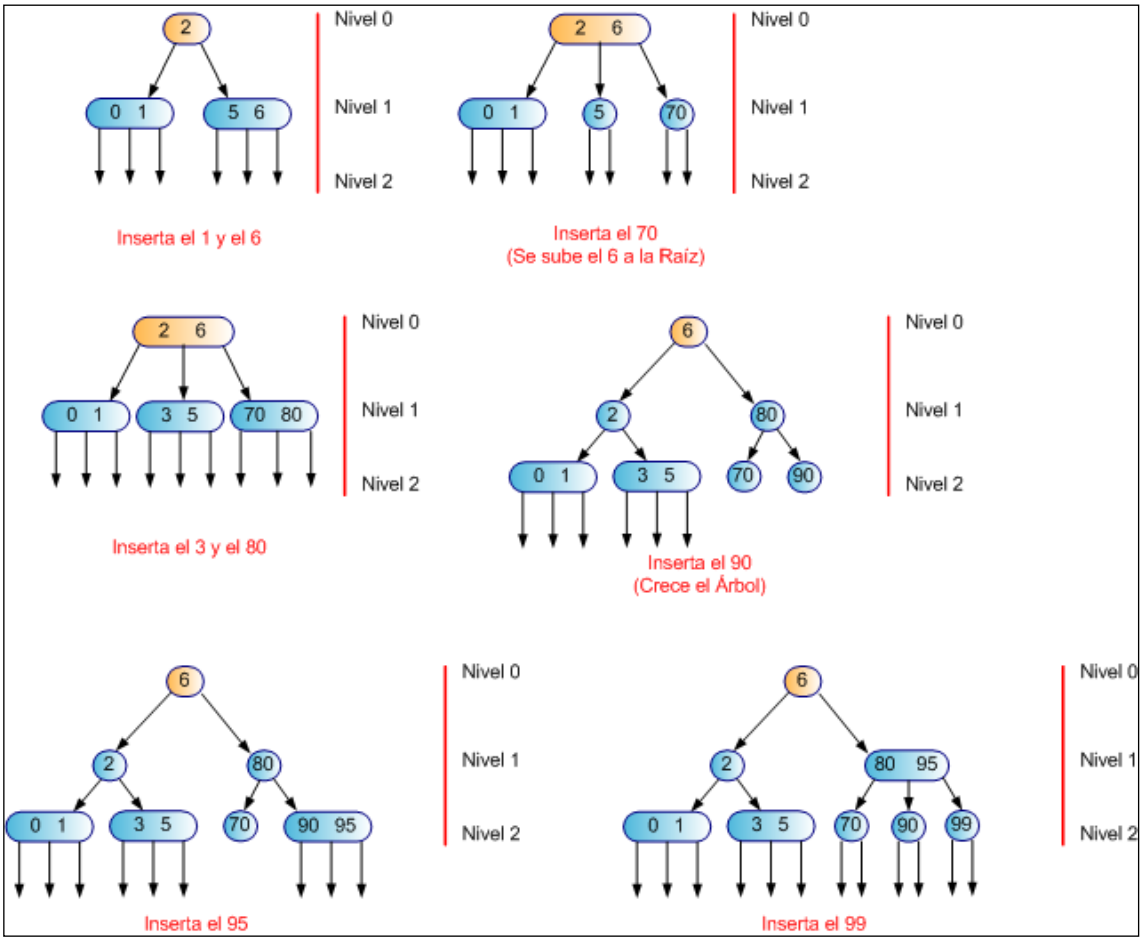


Figura 134: "Análisis de la Partición de Nodos Intermedios de un Árbol B y su incidencia en la Raíz"

Vea que, siguiendo con la secuencia de inserción comentada inicialmente, si se agrega al árbol los valores 1 y 6, no ocurre expansión alguna en términos de nodos en el árbol, ya que existe espacio disponible en los nodos inferiores para almacenarlos. Si luego se inserta el valor "70", al no existir espacio en el nodo destino (quien contiene el 5 y 6 originalmente), se "parte" este nodo de la siguiente forma:

- La secuencia de claves, previa a la partición del nodo, sería 5, 6 y 70.
- El valor medio de la anterior secuencia es el '6', éste es trasladado a la raíz del árbol (más adelante se verá que ocurre si el padre no tiene espacio para almacenar la clave trasladada),
- Y los valores 5 y 70, se utilizan para generar los hijos izquierdo y derecho de la llave '6'.

Note que si bien se particionó el nodo, debido a la falta de espacio en el nodo originalmente destinatario de la clave '70', no se creó un nuevo nodo como se hubiese efectuado en un árbol de arriba hacia abajo, sino que se trasladó la clave a un nodo superior y *se mantuvo el nivel del árbol*. Esta última característica es sumamente interesante en la indexación de grandes volúmenes de datos, ya que recuerde que estas estructuras de datos se emplean generalmente en la gestión de almacenamiento en dispositivos externos, por lo que un arbol de menor nivel implica menor cantidad de lecturas de disco, lo que se traduce en una mayor eficiencia al momento del recupero de los datos.

La inserción de los valores 3 y 80 no respresentan mayores inconvenientes, debido a que existe espacio suficiente donde almacenarlos en sus correspondientes destinos asociados. Lo Interesante sucede cuando desea insertar el valor "90", siga la siguiente secuencia:

- El 90 debiera ubicarse en el nodo que contiene el 70 y 80. Por ende la secuencia del nodo sería 70, 80 y 90. El valor 80 es la clave candidata a subir a su padre, mientras que el 70 y 80 colgarán a izquierda y derecha de éste.
- El 80 debería subirse al nodo que contiene 2 y 6. Por ende la secuencia sería 2, 6 y 80. El valor 6 es la clave candidata a subir como raíz, mientras que los valores 2 y 80 colgarán a izquierda y derecha de éste, y con ello sus hijos respectivamente.

Note que creció nuevamente el nivel del árbol, pero solo porque la raíz del mismo se vió excedida. Si ahora se inserta el valor 95, este es perfectamente almacenable a la derecha del 90 pero, ¿Si luego insertoel 99? Éste debiera situarse a la derecha del 95, pero al no existir espacio debe partirse el nodo. La secuencia de valores de éste último sería 90, 95 y 99, con esto se sabe que el candidato a subir al padre es el 95, mientras que el 90 y el 99 serían el hijo izquierdo y derecho de éste. Como el padre del nodo a partirse solo contiene una clave, la llave 80, puede almacenar un valor adicional, lo que provoca que el valor 95 se almacene a la derecha del 80.

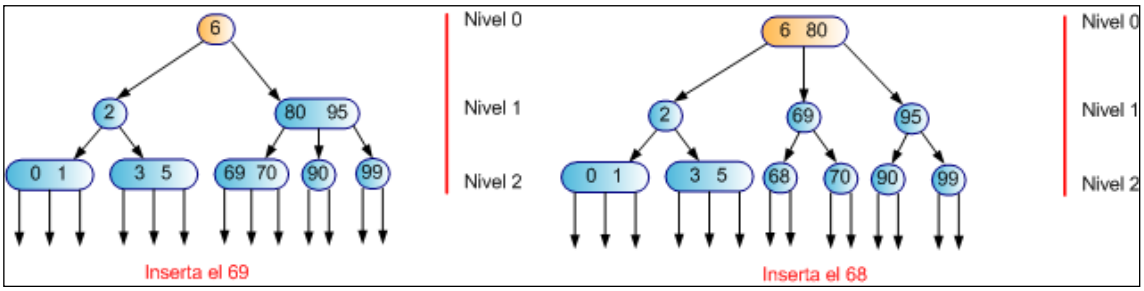


Figura 135: "Análisis de la Partición de Nodos Intermedios de un Árbol B y su incidencia en la Raíz"

Si ahora se inserta el valor 69, éste se puede almacenar perfectamente a la izquierda del 70, pero si luego se inserta el 68 es necesario un particionamiento. De la secuencia 68, 69 y 70, el candidato a subir es el '69'. Alsubir éste último se encuentra con que el nodo padre no tiene espacio, y quedaría la secuencia 69, 80 y 95, esto indica que el candidato a subir es el 80. Como el padre del 80 si tiene espacio, se almacena a la derecha del 6, colgando a izquierda y derecha de éste el 69 y el 95. A la izquierda y derecha jdel 69, quedará el 68 y 70 tal y como se había predicho en la primera partición.

Note que el particionamiento trabaja *siempre* desde el nodo donde se debiera insertar la nueva llave hacia arriba y concluye, en dos situaciones:

1. Encuentra un nodo padre superior donde almacenar la llave que se sube ó
2. Se parte la raíz y se incrementa el nivel del árbol.

Note que cada vez es más complicado provocar un particionamiento de la raíz, debido a que a medida que se incrementa en uno el nivel, se generan más y más hojas dentro del mismo, lo que incrementa la probabilidad de que se almacene un valor previo a incrementar el nivel.

Borrado Lógico en Árboles B

Como se mencionó anteriormente, este tipo de estructuras son empleadas para indexar grandes volúmenes de datos y minimizar las lecturas desde los dispostivos mecánicos de almacenamiento externo. Este afan de minimizar las lecturas/escrituras de los dispositivos externos, lleva a que se emple frecuentemente un borrado lógico. Éste último, consiste simplemente en marcar una clave o nodo, como eliminado pero sin eliminarlo físicamente.

¿Por qué no es conveniente la eliminación física? Imagine que un árbol B se basa en la premisa del equilibrio, por lo que si usted eliminase un nodo, debería en ese mismo momento re equilibrar el árbol completo, lo que adicionaría un tiempo de lectura/escritura sumamente elevado en términos de recursos, que llevaría a hacer caer la eficiencia del sistema.

El borrado lógico permite que periódicamente, se regenere o "Se efectúe Mantenimiento" de la estructura de indexación o búsqueda, para mejorar su rendimiento. Éste implica, que marco una llave o nodo como eliminado, y que al momento de la carga del nodo desde el disco en la memoria, se los trae junto con las llaves activas, pero gracias a la marca de eliminación, es factible discernir entre una y otra. Esto en efecto, provoca que se incremente el tiempo de búsqueda en la estructura, pero seguramente es muy inferior al tiempo que consumiría el re equilibrio del árbol por cada eliminación física. El mantenimiento de la estructura, consistiría en eliminar el índice y volverlo a crear.

Para comprender porqué se incrementa el tiempo de respuesta si se emplea el borrado lógico, aunque sea más eficiente que el borrado físico, vea el siguiente gráfico:

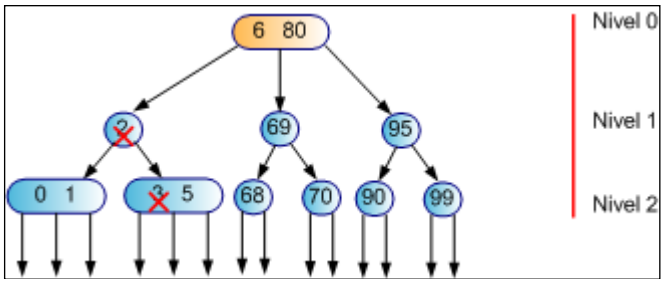


Figura 136: "Análisis del Borrado Lógico"

Suponga que del árbol de la figura anterior, usted tiene las llaves 2 y 3 borradas lógicamente y, un usuario solicita analizar si existe la llave 3, la secuencia sería:

- ¿3 está en el nodo que contiene 6 y 80? No, descienda por el subárbol 0.

- ¿3 está en el nodo que contiene 2? No, descienda por el subárbol 1. Pero note que leyó el valor 2 para saber como continuar el camino, lo que implicó leer un bloque de disco del nodo que contiene a dicha llave.
- ¿El 3 está en el nodo que contiene 3 y 5? Sí, pero está eliminado lógicamente, esto implica que "no está".

Vea que si bien pudo responderse la consulta del usuario, se debió leer el nodo que contiene al 2 desde el disco, lo que implica un retardo en la respuesta, luego se debió leer el nodo que contiene el 3 y 5, y encontrado el valor, debemos responder que no existe debido a que está borrado lógicamente.

En caso de que luego se inserte el valor que actualmente está borrado lógicamente, tiene dos alternativas:

1. Pisa el registro de datos que existía originalmente en el árbol y coloca la marca de borrado en "Falso", ó
2. Solo Coloca la marca de borrado en "Falso",

Depende de la naturaleza del dato indexdo, si es frecuente que el dato dentro o fuera del nodo del cual disponemos la llave, se haya o no modificado. Obviamente la diferencia entre uno y otro, amén de la modificación propiamente dicha, pasa por el tiempo necesario para concluir cada una de las operaciones asociadas a los mismos.

A continuación, se lista el programa "ArbolB.cpp", que implementa en forma estructurada un árbol B en memoria:

Archivo: **"ArbolB.cpp"**

```
1       //Implementacion de un Arbol Multidireccional
2       //Antesala de Arboles B
3
4       #include <iostream.h>
5       #include <conio.h>
6
7       const short int VACIO=-1;
8       const short int true=1;
9       const short int false=0;
10
11      void inicializaNodo(int,void**,int*,int*);
12
13      struct nodo{
14                  int *key;
15                  nodo **sons;
16                  int qArbolOcupados;
17                  int qKeysOcupadas;
18                  int qKeysMaximas;
19                  int *borrado;
20
21                  nodo(const int orden,int keyInicial)
22                  {
```

```
23                  qKeysMaximas=(orden-1);
24                  qKeysOcupadas=0;
25                  qArbolOcupados=2;
26
27                  key=new int[(orden-1)];
28                  borrado=new int[(orden-1)];
29
30                  sons=new nodo*[orden];
31                  inicializaNodo(orden,(void**)sons,key,borrado);
32                  key[0]=keyInicial;//en la posici¢n a la izquierda, coloco
33                                    //el primer elemento
34                  qKeysOcupadas++;//incremento el n£mero de llaves en el nodo
35                 };
36            };
37
38     int nroArbolesEn(nodo*);
39     int keyEn(nodo*,int);
40     int buscarClaveEn(nodo*,int);
41     nodo* buscar(nodo*,int);
42     nodo* hijoDe(nodo*,int);
43     void recorrido(nodo*);
44     nodo* verLugarInsert(nodo*,int,int&,int&);
45     void insertarRegistro(nodo*,int,int);
46     nodo* crearNodo(int,int);
47     void insertarNodo(nodo*&, int,int);
48     void insertarHoja(nodo*,int,int);
49     void borrarNodo(nodo*&,int);
50     nodo* buscarPadre(int,int&,nodo*&);
51     void rotacion(nodo*&);
52     nodo* buscoClaveRotar(int,int&,nodo*,nodo*&,int&);
53     void copy(nodo*,int,int,nodo*,int);
54     void insnodeB(nodo*, int, int, nodo*);
55     void partir(nodo*,int,int,nodo*,nodo*&,int&);
56     void borradoLogico(nodo*,int);
57     void freeWilly(nodo *&);
58
59     //Variable Global con Orden/2 for Arbol B
60     int ndiv2=0;
61
62     void main(void)
63     {
64      clrscr();
65      int ordenArbol=3;//El Inicialzador de Arbol
66      ndiv2=ordenArbol/2;
67
68      nodo *raiz=NULL;
69      insertarNodo(raiz,ordenArbol,2);
70      insertarNodo(raiz,ordenArbol,5);
71      insertarNodo(raiz,ordenArbol,0);
72      insertarNodo(raiz,ordenArbol,1);
```

```
73      insertarNodo(raiz,ordenArbol,6);
74      insertarNodo(raiz,ordenArbol,70);
75      insertarNodo(raiz,ordenArbol,3);
76      insertarNodo(raiz,ordenArbol,80);
77      insertarNodo(raiz,ordenArbol,90);
78      insertarNodo(raiz,ordenArbol,95);
79      insertarNodo(raiz,ordenArbol,99);
80      insertarNodo(raiz,ordenArbol,69);
81      insertarNodo(raiz,ordenArbol,68);
82
83      cout<<"Raiz:"<< raiz->key[0]<<"-"<<raiz->key[1]<<"\n\n";
84      recorrido(raiz);
85      cout<<"-----------------\n";
86      borradoLogico(raiz,5);
87      borradoLogico(raiz,2);
88      borradoLogico(raiz,0);
89      recorrido(raiz);
90
91      int x;cin>>x;
92      insertarNodo(raiz,ordenArbol,5);
93      recorrido(raiz);
94
95      freeWilly(raiz);
96
97      if(raiz==NULL)cout<<"\n\nMemoria Liberada";
98     }
99
100    //Liberen a Willy=> libera la memoria utilizada por el Arbol B
101    void freeWilly(nodo*& p)
102    {
103     if(p==NULL) return;
104
105     for(int i=0;i<p->qArbolOcupados;i++)
106       {
107        if(p->sons[i]!=NULL)
108         freeWilly(p->sons[i]);
109       }
110
111      delete p;
112      p=NULL;
113    }
114
115    void inicializaNodo(int orden,void** voidsons, int* key,int* bor)
116    {
117     nodo** sons=(nodo**)voidsons;
118
119     for(int i=0;i<orden;i++)
120     {
121       sons[i]=NULL;//anulo todos los punteros a hijo
```

```
122        if(i<(orden-1))
123        {
124         key[i]=VACIO;
125         bor[i]=false;
126        }
127      }
128     }
129
130     int nroArbolesEn(nodo* p)
131     {
132      if(p==NULL) return 0;
133
134      return p->qArbolOcupados;
135     }
136
137     //Retorna la clave del nodo p en la posici¢n j
138     int keyEn(nodo* p, int j)
139     {
140      if(p==NULL) return -1;
141      if(j<0) return -1;
142      if(j>p->qKeysOcupadas) return -1;
143
144      return p->key[j];
145     }
146
147     //Retorna el entero j mas peque¤o, tal que key<= k(p,j), donde
148     //k(p,j) retorna la clave en la posicion j para el nodo p o
149     int buscarClaveEn(nodo* elnodo, int key)
150     {
151      if(elnodo==NULL) return (-1);
152
153      if(elnodo->qKeysOcupadas==0) return (-1);
154      if(elnodo->key==NULL) return (-1);
155
156      //Se hace una B£squeda Secuencial de la posicion "j", que
157      //contiene una Clave >= a key (o, key<=clave en pos. j)
158
159      for(int i=0;i<elnodo->qKeysOcupadas;i++)
160      {
161       if(key<=elnodo->key[i]) return i;
162      }
163
164      return elnodo->qKeysOcupadas;//es lo mismo que (numArbol-1)
165     }
166
167     nodo* buscar(nodo* arbol,int key)
168     {
169      nodo* p=arbol;
170
171      while(p!=NULL)
```

```
172     {
173      int i=buscarClaveEn(p,key);
174
175      if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
176      {
177       return p;
178      }
179
180      p=hijoDe(p,i);
181     }
182
183     return NULL;
184    }
185
186    nodo* hijoDe(nodo* p,int pos)
187    {
188     if(p==NULL)return NULL;
189     if(p->sons==NULL) return NULL;
190     if(pos<0 || pos>(p->qKeysMaximas+1)) return NULL;
191     if(p->sons[pos]==NULL) return NULL;
192
193     return p->sons[pos];
194    }
195
196    void recorrido(nodo* p)
197    {
198     if(p!=NULL)
199      {
200       int nroArboles=p->qArbolOcupados;
201
202       for(int i=0;i<(nroArboles-1);i++)
203       {
204        recorrido(hijoDe(p,i));
205
206        if(p->borrado[i]==false)
207               cout<<i<<":-"<<keyEn(p,i)<<"-\n";
208       }
209
210       recorrido(hijoDe(p,(nroArboles-1)));
211      }
212    }
213
214    nodo* verLugarInsert(nodo* arbol,int key,int& posicion, int& encontrado)
215    {
216      nodo* q=NULL;
217      nodo* p=arbol;
218
219      int i=0;
220      while(p!=NULL)
```

```
221      {
222       i=buscarClaveEn(p,key);
223       q=p;
224
225       if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
226       {
227        encontrado=true;
228        posicion=i;
229        return p;
230       }
231
232       p=hijoDe(p,i);
233      }
234
235      encontrado=false;
236      posicion=i;
237      return q;
238    }
239
240    void insertarRegistro(nodo* s,int posicion,int key)
241    {
242     s->key[posicion]=key;
243     s->qKeysOcupadas++;//se ocupo una posicion del nodo
244     s->qArbolOcupados++;//se agrega un arbol mas
245    }
246
247    nodo* crearNodo(int orden,int key)
248    {
249     nodo* nuevonodo=new nodo(orden,key);
250
251     return nuevonodo;
252    }
253
254    void insertarNodo(nodo*& arbol, int orden,int key)
255    {
256     if(arbol==NULL)
257     {
258      arbol=crearNodo(orden,key);//crea el arbol
259      return;
260     }
261
262     int posicion=0;
263     int encontrado=false;
264     nodo* s=verLugarInsert(arbol,key,posicion,encontrado);
265
266     if(encontrado==true)
267     {//Pongo borrado en false, para pasarlo de borrado l¢gicamente a Activo
268      s->borrado[posicion]=false;
269      return;//ya existe
270     }
```

```
271
272     if(nroArbolesEn(s)<orden)
273     {
274      insertarHoja(s,posicion,key);
275      return;
276     }
277     //Aqu¡ es donde trabaja el µrbol B
278     //si llega aqui es porque cant. arboles en s es igual al orden
279     int posEnPadre=0;
280     nodo *padre=buscarPadre(s->key[0],posEnPadre,arbol);
281     nodo* nd2=NULL;
282     nodo* nd=s;
283     nodo* nuevonodo=NULL;
284     int clavemedia=0;
285     int newkey=key;
286
287     while(padre!=NULL && nd->qArbolOcupados==(nd->qKeysMaximas+1))
288     {
289      partir(nd,posicion,newkey,nuevonodo,nd2,clavemedia);
290      nuevonodo=nd2;
291      posicion=posEnPadre;
292      nd=padre;
293      padre=buscarPadre(nd->key[0],posEnPadre,arbol);
294      newkey=clavemedia;
295     }
296
297     if(nd->qArbolOcupados<(nd->qKeysMaximas+1))
298      {
299       insnodeB(nd,posicion,newkey,nuevonodo);
300       return;
301      }
302
303      partir(nd,posicion,newkey,nuevonodo,nd2,clavemedia);
304      arbol=new nodo(nd->qKeysMaximas+1,clavemedia);
305      arbol->sons[0]=nd;
306      arbol->sons[1]=nd2;
307      nd2->sons[0]=nd->sons[nd2->qKeysMaximas];
308
309      arbol->qKeysOcupadas=1;
310      arbol->qArbolOcupados=2;
311
312     //Si llega aqu¡ es porque el padre es NULL y el nro de Arboles es igual
313     //a N. Es una raiz ocupada, hay que dividirla y crear una nueva raiz
314     return;
315    }
316
317    void insertarHoja(nodo *s, int posicion, int key)
318    {
319     for(int i=(s->qArbolOcupados-1);i>posicion;i--)
```

```
320     {
321      s->key[i]=s->key[i-1];
322      s->sons[i+1]=s->sons[i];
323      s->sons[i]=((i-1)>0)?s->sons[i-1]:NULL;
324     }
325
326     s->key[posicion]=key;
327
328     if(posicion==(s->qArbolOcupados-1))
329     {//Porque el valor que ingreso es el extremo derecho
330      s->sons[posicion+1]=NULL;
331     }
332     else
333      s->sons[posicion]=NULL;
334
335     insertarRegistro(s,posicion,key);
336    }
337
338    void borradoLogico(nodo* arbol,int key)
339    {//Se implementa el borrado fisico del nodo
340     nodo* p=arbol;
341
342     nodo* res=buscar(p,key);
343
344     if(res==NULL)
345     {
346       cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
347       return;
348     }
349
350     int posicion=buscarClaveEn(res,key);
351
352     if(keyEn(res,posicion)!=key)
353     {
354       cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
355       return;
356     }
357
358     res->borrado[posicion]=true;
359    }
360
361    void borrarNodo(nodo*& arbol,int key)
362    {//Se implementa el borrado fisico del nodo
363     nodo* p=arbol;
364
365     nodo* res=buscar(p,key);
366
367     if(res==NULL)
368     {
369       cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
```

```
370        return;
371      }
372
373      int posicion=buscarClaveEn(res,key);
374
375      if(keyEn(res,posicion)!=key)
376      {
377        cout<<"Nodo con el valor "<<key<<", no existe en el arbol";
378        return;
379      }
380
381      //Si se esta Aqui, es porque en el nodo direccionado por "res"
382      //en el lugar "posicion" se encuentra el valor a eliminar
383      nodo* hizq=hijoDe(res,posicion);
384
385      if(hizq!=NULL)
386      {//Busco el mayor descendiente derecho del hijo izquierdo
387       int a;
388
389       int   veoPadrePosMayor=posicion;
390       nodo* veoPadre=res;
391       nodo* veo=hizq;
392
393       while(veo!=NULL)
394       {//Encuentro el mayor descendiente izquierdo
395        int posMayor=buscarClaveEn(veo,key);
396
397        if(veo->sons[posMayor]==NULL)
398        {//Se Encontro el mayor descendiente
399           res->key[posicion]=veo->key[(posMayor-1)];
400           veo->qKeysOcupadas--;
401           veo->qArbolOcupados--;
402
403           if(veo->qKeysOcupadas>=1)
404           {//solo anulo la posicion de la antigua clave
405            veo->key[(posMayor-1)]=VACIO;
406            return;
407           }
408           else
409           {//Si nuestro mayor descendiente, era el £nico en el nodo
410            veoPadre->sons[veoPadrePosMayor]=veo->sons[0];
411            delete veo;
412            veo=NULL;
413            return;
414           }
415        }
416
417        veoPadre=veo;
418        veoPadrePosMayor=posMayor;
```

```
419        veo=hijoDe(veo,posMayor);
420       }//fin del while
421      }//fin de si hizq!=NULL
422
423
424      nodo* hder=hijoDe(res,(posicion+1));
425      if(hder!=NULL)
426      {//Busco el mayor descendiente derecho del hijo izquierdo
427       int   veoPadrePosMenor=posicion+1;
428       nodo* veoPadre=res;
429       nodo* veo=hder;
430
431
432       while(veo!=NULL)
433       {//Encuentro el mayor descendiente izquierdo
434        int posMenor=buscarClaveEn(veo,key);
435
436        if(veo->sons[posMenor]==NULL)
437        {//Se Encontro el mayor descendiente
438           res->key[posicion]=veo->key[(posMenor)];
439           veo->qKeysOcupadas--;
440           veo->qArbolOcupados--;
441
442           if(veo->qKeysOcupadas>=1)
443           {//solo anulo la posicion de la antigua clave
444            veo->key[(posMenor)]=VACIO;
445
446            //Efectuo un Corrimiento a Izquierda de las Claves Remanentes
447            int z=0;
448            for(;z<veo->qKeysOcupadas;z++)
449            {
450             veo->key[z]=veo->key[z+1];
451             veo->sons[z]=veo->sons[z+1];
452            }
453            veo->key[z]=VACIO;//El antiguo extremo derecho que se traslado
454            veo->sons[z]=veo->sons[z+1];//Corrimiento del Extremo Derecho
455
456            return;
457           }
458           else
459           {//Si nuestro mayor descendiente, era el £nico en el nodo
460            veoPadre->sons[veoPadrePosMenor]=veo->sons[0];
461            delete veo;
462            veo=NULL;
463            return;
464           }
465        }
466
467        veoPadre=veo;
468        veoPadrePosMenor=posMenor;
```

```
469        veo=hijoDe(veo,posMenor);
470       }//fin del while
471      }//fin de si hder!=NULL
472
473      //Si se llega a esta instancia => que hder=hizq=NULL
474      //Efectuo el Corrimiento a Izquierda Necesario
475      int posicionResEnPadre=0;
476      nodo* padreDeRes=buscarPadre(key,posicionResEnPadre,arbol);
477
478      int pz=posicion;
479      for(;pz<res->qKeysOcupadas;pz++)
480       {
481        res->key[pz]=res->key[pz+1];
482        res->sons[pz]=res->sons[pz+1];
483       }
484
485      res->key[pz]=VACIO;//El antiguo extremo derecho que se traslado
486      res->qKeysOcupadas--;
487      res->qArbolOcupados=(res->qKeysOcupadas==0)?0:res->qArbolOcupados-1;
488      res->sons[pz]=res->sons[pz+1];//Corrimiento del Extremo Derecho
489
490
491      if(res->qKeysOcupadas>0)
492       {//Aqu¡ es donde debo subir una clave de izq o derecha
493        rotacion(res);
494        return;
495       }
496       else
497       {//Debo eliminar el nodo y anular el puntero del padre al mismo
498        if(padreDeRes!=NULL)
499         {//Si fuese NULL, res es la raiz del arbol
500          if(posicionResEnPadre!=-1)
501             padreDeRes->sons[posicionResEnPadre]=NULL;
502          else
503             arbol=NULL;//el padre era el puntero cabeza del arbol(main)
504         }
505
506        delete res;
507        res=NULL;
508       }
509
510     }
511
512     nodo* buscarPadre(int key,int& posicionPadre,nodo*& arbol)
513     {
514      nodo* p=arbol;
515      nodo* padre=NULL;
516      posicionPadre=buscarClaveEn(padre,key);
517
```

```
518     int i=0;
519     while(p!=NULL)
520     {
521      i=buscarClaveEn(p,key);
522
523      if(i<(nroArbolesEn(p)-1) && key==keyEn(p,i))
524      {
525       //padre; es el actualmente direccionado
526       return padre;
527      }
528
529      posicionPadre=i;
530      padre=p;
531      p=hijoDe(p,i);
532     }
533
534     padre=NULL;
535     return padre;
536    }
537
538    //Funcion prototipo de rotacion de claves, no probada aun, verificar
539    //faltaria la de rotacion izquierda tambien
540    void rotacion(nodo*& p)
541    {
542    //Localizo la Rama del Sub rbol distinta de NULL desde derecha a Izq
543     int posicion;
544     int ready=false;
545
546     for(posicion=(p->qArbolOcupados-1);posicion>=0 && !ready;posicion--)
547            if(p->sons[posicion]!=NULL)
548            {
549             ready=true;
550             posicion++;//incremento 1 porque luego viene posicion--, previo a
       evaluar la condicion
551            }
552
553     if(posicion<0) return;//es una hoja
554
555     nodo* padreARotar=p;//originalmente es el mismo nodo donde se elimina
556     nodo* arotar=NULL;
557     int claveARotar=-1;
558     int pospadre=posicion;//originalmente es por donde desciendo
559
560     if((posicion+1)==1)//es el subarbol inferior, implica corrimiento a la
       derecha
561      {
562       arotar=buscoClaveRotar(p->key[posicion],claveARotar,p-
       >sons[posicion],padreARotar,pospadre);
563       claveARotar--;//ya que retorna la pos del subarbol superior nulo
564      }
```

```
565
566     if((posicion+1)>p->qKeysOcupadas)
567     {
568      arotar=buscoClaveRotar(p->key[(p->qKeysOcupadas-1)],claveARotar,p-
>sons[posicion],padreARotar,pospadre);
569      if(arotar==NULL) return;//no hay que rotar
570
571      p->qKeysOcupadas++;
572      p->qArbolOcupados++;
573
574      //Se agrega la clave al final del nodo
575      p->key[(p->qKeysOcupadas-1)]=arotar->key[claveARotar];
576
577      if(arotar->qKeysOcupadas==1)//pregunto antes de decrementar porque
puede ser necesario alg£n corrimiento
578      {
579       arotar->qKeysOcupadas--;
580       arotar->qArbolOcupados--;
581       delete arotar;
582       arotar=NULL;
583       if(padreARotar!=NULL) padreARotar->sons[pospadre]=NULL;
584
585       p->sons[(p->qArbolOcupados-1)]=p->sons[(p->qArbolOcupados-2)];
586       p->sons[(p->qArbolOcupados-2)]=NULL;
587       return;//la clave rotada era el unico valor del nodo origen, fin.
588      }
589      //Si estoy aqu¡, hace falta un corrimiento a la izq. del nodo origen
590      //de la clave rotada
591
592      int corr=0;
593
594      for(;corr<p->qArbolOcupados;corr++)
595      {
596       arotar->sons[corr]=arotar->sons[corr+1];
597       if((corr+1)<p->qKeysOcupadas) arotar->key[corr]=arotar->key[corr+1];
598      }
599
600      arotar->sons[(p->qArbolOcupados-1)]=NULL;
601      arotar->key[(p->qKeysOcupadas-1)]=VACIO;
602      arotar->qArbolOcupados--;
603      arotar->qKeysOcupadas--;
604
605      rotacion(arotar);//el nodo en que se efectuo el corrimiento debe ser
completado
606      return;
607     }//fin  del if que nos decia que estabamos parados en la puerta del
subarbol superior
608     else
609     {//es un subarbol intermedio, sera necesario un corrimiento a la derecha
```

```
610      arotar=buscoClaveRotar(p->key[posicion],claveARotar,p-
       >sons[posicion],padreARotar,pospadre);
611      claveARotar--;
612     }
613
614     if(arotar==NULL) return;//no hay que rotar
615
616     p->qKeysOcupadas++;
617     p->qArbolOcupados++;
618
619     int corr=p->qArbolOcupados-1;
620
621     for(;corr>posicion;corr--)
622     {
623      if((corr-1)!=posicion)
624      {
625       p->sons[corr]=p->sons[corr-1];
626       p->key[corr-1]=p->key[corr-2];
627      }
628     }
629
630     p->sons[posicion+1]=NULL;
631     p->key[posicion]=arotar->key[claveARotar];
632     arotar->qKeysOcupadas--;
633     arotar->qArbolOcupados--;
634
635     if(arotar->qKeysOcupadas==0)
636     {
637      delete arotar;
638      arotar=NULL;
639      if(padreARotar!=NULL) padreARotar->sons[pospadre]=NULL;
640      return;
641     }
642
643     rotacion(arotar);//completo el nodo origen de la clave rotada
644    }
645
646    nodo* buscoClaveRotar(int key, int& posicion, nodo* bolsa, nodo*&
       padreARotar, int& pospadre)
647    {
648     nodo* p=bolsa;
649     nodo* aux=padreARotar;
650
651     while(p!=NULL)
652     {
653      int bajo=buscarClaveEn(p,key);
654
655      if(p->sons[bajo]==NULL)
656      {//encontre
657       padreARotar=aux;
```

```
658        posicion=bajo;
659        return p;
660       }
661
662       pospadre=bajo;
663       aux=p;
664       p=p->sons[bajo];
665      }
666
667      return p;// aqui p es NULL
668     }
669
670     void copy(nodo* ptrOrigen, int obegin, int oend, nodo* ptrDestino, int
        dbegin)
671     {
672      int move=obegin;
673
674      for(;move<ptrOrigen->qKeysOcupadas && move<=oend;move++)
675      {
676       ptrDestino->key[dbegin]=ptrOrigen->key[move];
677       ptrDestino->sons[dbegin]=ptrOrigen->sons[move];
678       dbegin++;
679
680       if((move+1)!=ptrOrigen->qKeysOcupadas && move!=oend)
681       {
682        ptrDestino->qKeysOcupadas++;
683        ptrDestino->qArbolOcupados++;
684       }
685      }
686
687      if(obegin<=oend)
688             ptrDestino->sons[dbegin]=ptrOrigen->sons[move];
689     }
690
691     void insnodeB(nodo* nd, int pos, int key, nodo* newnode)
692     {
693      for(int i=(nd->qArbolOcupados-1);i>=(pos+1);i--)
694      {
695       nd->sons[i+1]=nd->sons[i];
696       nd->key[i]=nd->key[i-1];
697      }
698
699      nd->qArbolOcupados++;
700      nd->qKeysOcupadas++;
701      nd->sons[pos+1]=newnode;
702      nd->key[pos]=key;
703     }
704
```

```
705    void partir(nodo* nd, int pos, int key,nodo* newnode,nodo*& nd2, int&
       clavemedia)
706    {
707     nd2=new nodo((nd->qKeysMaximas+1),0);
708
709     if(pos>ndiv2)
710     {//la nueva clave pertenece al nodo derecho -nd2-
711      copy(nd,ndiv2+1,(nd->qKeysMaximas-1),nd2,0);
712      insnodeB(nd2,pos-ndiv2-1,key,newnode);
713      nd->qArbolOcupados=ndiv2+1;
714      nd->qKeysOcupadas=nd->qArbolOcupados-1;
715      clavemedia=nd->key[ndiv2];
716
717      if(ndiv2==1 && nd2->qKeysMaximas==2)
718      {
719       nd2->key[1]=VACIO;
720       nd2->sons[2]=NULL;
721       nd2->qArbolOcupados--;
722       nd2->qKeysOcupadas--;
723      }
724
725      return;
726     }
727
728     if(pos==ndiv2)
729     {//la nueva es la media
730      copy(nd,ndiv2,(nd->qKeysMaximas-1),nd2,0);
731      nd->qArbolOcupados=ndiv2+1;
732      nd->qKeysOcupadas=nd->qArbolOcupados-1;
733      nd2->sons[0]=newnode;
734      clavemedia=key;
735
736      return;
737     }
738
739     if(pos<ndiv2)
740     {//la nueva clave pertenece al nodo izquierdo -nd-
741      copy(nd,ndiv2,(nd->qKeysMaximas-1),nd2,0);
742
743      if(ndiv2>1)
744      {
745       nd->qArbolOcupados=ndiv2;
746      }
747      else
748      {
749       nd->qArbolOcupados=2;
750       nd->sons[ndiv2]=NULL;//ya que fue copiado a la hoja derecha nd2
751      }
752
753      nd->qKeysOcupadas=nd->qArbolOcupados-1;
```

```
754      insnodeB(nd,pos,key,newnode);
755
756      clavemedia=nd->key[ndiv2];//cambiado
757
758      if(ndiv2==1 && nd->qKeysMaximas==2)
759      {
760       nd->key[ndiv2]=VACIO;
761       nd->sons[ndiv2+1]=NULL;
762       nd->qArbolOcupados--;
763       nd->qKeysOcupadas--;
764      }
765      return;
766     }
767    }
```

Liberación de la Memoria del Árbol B

Dentro del programa "ArbolB.cpp" existe una función denominada "freeWilly", que recibe como parámetro un puntero referencia a un nodo del árbol. El objeto de esta función, es recorrer recursivamente el arbol B, desde abajo hacia la raíz, liberando en el camino la memoria utilizada por cada nodo del árbol.

```
100    //Liberen a Willy=> libera la memoria utilizada por el Arbol B
101    void freeWilly(nodo*& p)
102    {
103     if(p==NULL) return;
104
105     for(int i=0;i<p->qArbolOcupados;i++)
106       {
107        if(p->sons[i]!=NULL)
108         freeWilly(p->sons[i]);
109       }
110
111      delete p;
112      p=NULL;
113    }
```

Vea como se ubican os hijos de cada nodo de izquierda a derecha, y solo inicia el retorno cuandoel puntero es nulo. Tenga en cuenta que solo se borra el nodo si se confirma que es una hoja, lo que implica que no tiene asociados subárboles o bien ya fueron eliminados previamente. La Analogía de la función con "Willy", se vincula al volumen de memoria que potencialmente puede llegar a consumir esta estructura.

Normalmente no se carga el árbol B en memoria en su completitud, sino que se carga solo algunos pocos nodos y, a medida que se requieren, se cargan nuevos nodos y se descargan los pocos utilizados, al mejor estilo de "Memoria Caché".

Árbol B+

Una de las principales desventajas de un Árbol B es el recorrido secuencial de sus registros, ya que para ello debe hacer un "sube y baja" dentro de la estructura del

árbol. Recuerde que en este punto, para recorrer y mostrar la estructura 'B' en orden, debíamos emplear la función recursiva *"recorrido"* en la implementación anterior de árboles B. En consecuencia, por cada descenso en el árbol para consultar el siguiente registro, incurríamos en el orden *O(log(n))* del árbol B.

Una variante del árbol B, que da solución a la problemática enunciada, son los árboles B^+, éstos permiten un orden *O(log(n))* en la búsqueda de un registro y, un *O(1)* en la búsqueda del siguiente y/o anterior registro.

Los árboles B^+, conservan **todas** las llaves en sus hojas, y se duplican las llaves en nodos no hojas para definir trayectorias de búsqueda, con el propósito de localizar *regstros individuales*. Las hojas se encuentran doblemente enlazadas, para facilitar el recorrido entre sus hermanos, lo que le permite obtener un *O(1)* en la operación de obtención de alguno de sus hermanos.

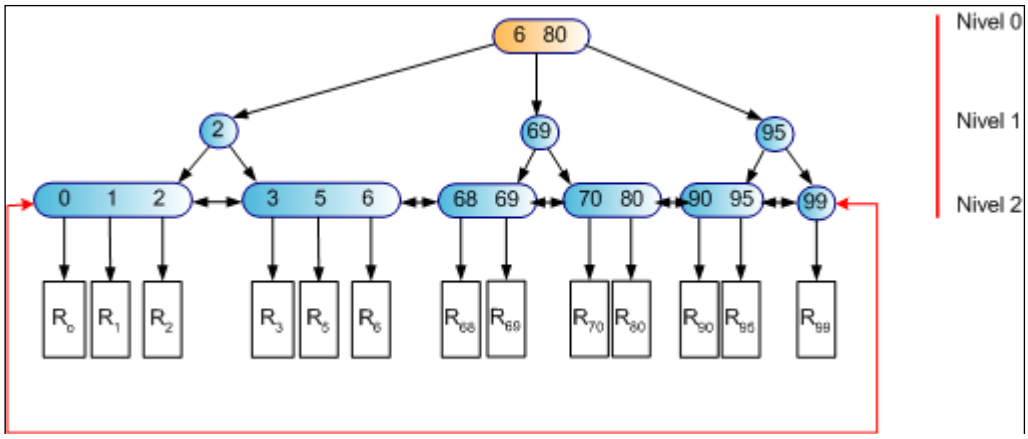


Figura 137: "Concepto de Árbol B^+"

Una variante del árbol B^+, es que sus hojas formen una lista circular doblemente enlazada, lo que al tener sus nodos ordenados, le permite decidir el camino más corto a izquierda o derecha, para obtener el registro buscado.

Note que las claves se duplican en la hoja superior más profunda de su primer hijo izquierdo. En el caso del 6, la llave se duplica a la derecha en el nodo con 3 y 5, y *no contiene hijo izquierdo y/o derecho como los nodos intermedios, sino que las llaves en las hojas contienen directamente el puntero al registro. Además, una fuerte ventaja de los árboles B^+, es que los nodos intermedios no necesitan almacenar apuntadores a registros, ya que de esto se encargan las hojas, lo que permite aumentar el orden potencial del árbol. Los únicos nodos que pueden contener más claves que las indicadas por su orden (n-1), son las hojas, ya que recuerde que son ellas quienes almancenan las llaves duplicadas y apuntadores de los nodos superiores, además de los propios.*

Note que la búsqueda, a diferencia de un árbol B, no se detiene cuando encuentra la llave, sino que se detiene cuando encuentra la hoja que contiene la llave, con lo cual debe seguir hasta el último nivel. El árbol B^+ solo almacena el apuntador al registro en la hoja, mientas que en un árbol B, al encontrar la llave se obteniene también el apuntador al registro.

La lista de hojas vinculadas, se las denomina *conjunto secuencia*, ya que representa un anillo o lita doblemente enlazada, depende la implementación que desee, cuyos nodos están ordenados, permitiendo un recorrido en serie a través del vínculo inter hoja.

En general cada nodo del conjunto secuencia, la hoja, no contiene todas las llaves del árbol sino que contiene un índice hacia un área de datos grande, en donde se conservan una gran cantidad de registros. En base a esto último, la búsqueda implica recorrer la trayectoria del árbol B^+, leer el bloque de datos asociado con la hoja que se accede finalmente, y después buscar en el bloque en forma secuencial el registro requerido. En base a esto último, es que se considera al árbol B^+, una extensión natural de un archivo secuencial indizado.

La Inserción en este tipo de árboles es similar a la del árbol B, excepto que cuando se divide un nodo (la función "partir" de la implementación árboles B del apartado anterior), la llave intermedia se conserva en la mitad izquierda, como si se promoviera hacia el padre. Cuando se elimina una llave de una hoja, es posible conservarla en los nodos no hojas, dado que es un separador o selector válido entre las llaves de los nodos inferiores; esto último no es ni más ni menos que un borrado lógico.

La implementación de esta estructura de datos para búsqueda, se deja como práctica al lector. Vea que la implementación de un árbol B^+, implican reformas sutiles pero precisas en la anterior implementación de árbol B, por lo que no debieran existir mayores complicaciones.

Hashing

Introducción

El Hashing, también conocido como dispersión, es una técnica que intenta obtener un dato en un orden 'k'. Cuando mencionamos "Orden K", si recordamos el capítulo correspondiente a "Análisis de Algoitmos", se convierte en algo sumamente tentador de implementar.

El Hashing se basa en una función de Dispersión, que trasnforma una llave en una posición en la tabla de datos donde se almacenará el registro; esto permite luego, ingresando la misma clave, conocer directamente la posición donde se encuentra almacenado el dato y recuperarlo.

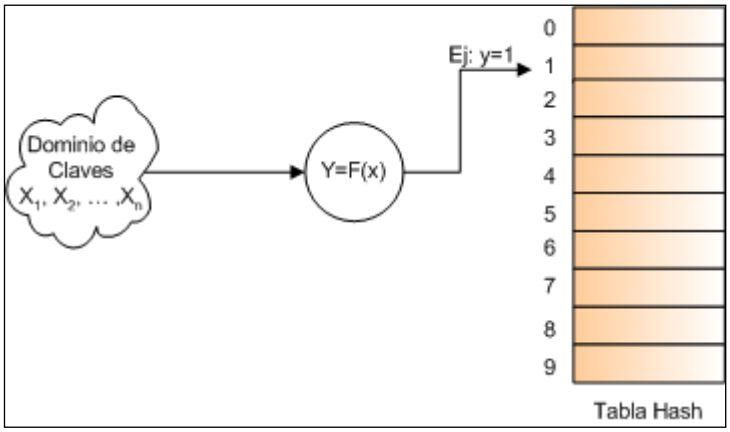


Figura 138: "Concepto de Hash"

Se entiende por Dominio de Claves, a la naturaleza de la clave, como así también las magnitudes en las cuales ésta puede definirse. Ejemplo: Las claves pueden ser enteras (Naturaleza), y asumir valores entre 0 y 9999 (Rango de Magnitudes).

La imagen de la función de dispersión, a partir del dominio de claves, será un conjunto natural, cuyo rango estará delimitado por el tamaño de la tabla hash. En efecto, suponga que su tabla tiene diez posiciones para almacenar datos como el de la figura anterior, si la función F(x) devolviese un '11' ¿sería correcto? No, porque dicha posición no existe realmente en la tabla.

Puede ocurrir que dos valores diferentes del Dominio de Claves, den como resultado un mismo valor en la imagen, a esta situación se la conoce como *"Colisión"*.

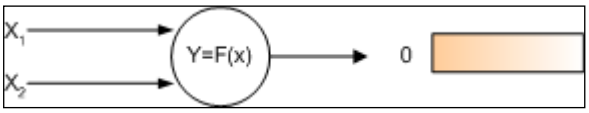


Figura 139: "Concepto de Colisión"

Se volverá sobre la temática de las colisiones en apartados posteriores.

Se puede decir que una F(X) es una buena función de dispersión relativa, si y solo si es la que dispone del menor número de colisiones para un dominio de claves dados y, una cantidad 'n' de funciones analizadas y comparadas.

Implementación de Hash Estático

El Hash Estático consiste en un tamaño de tabla fijo definido en una instancia inicial, el cual no será cambiado en tiempo de ejecución. Ésta técnica supone una fundón de dispersión que, si ocurriese una colisión, no efectúa acción alguna e informa que la llave que se está queriendo ingresar no será almacenada. Por esto último, es necesario un muy buen análisis de la naturaleza y magnitud de las llaves, a los fines de aproximar una buena función de dispersión.

Este mecanismo es útil en datos estáticos, que no cambian durante el tiempo de ejecución y a los cuales es necesario consultarlos en forma frecuente, como por ejemplo las localidades de una provincia o los códigos postales asociados a ellas.

En caso de que la naturaleza del dato a almacenar, no sea estática sino que presenta cierto grado de dinamismo, ésta variante del hasing no sería adecuada.

El comportamiento del algoritmo de inserción de hash estático, es relativamente sencillo:

1. Toma la clave del registro y la dispersa, con lo que obtiene la posición donde se ubicaría el registro dentro de la tabla,
2. Verifica si dicha posción dentro de la tabla se encuentra disponible. Si está disponible almacena la clave y el registro de datos, en caso contrario, avisa que se efectuó una colisión.

En caso de desear borrar un registro de datos, se aplica una operatoria similar a la anterior:

1. Toma la clave del registro y la dispersa, con lo que obtiene la posición donde se ubicaría el registro dentro de la tabla,
2. Verifica si en dicha posción dentro de la tabla, existe el registro con la clave dispersada. Si está disponible pisa el registro, en caso contrario, avisa que el registro no existe.

La consulta, es sumamente sencilla, solo dispersa la clave que desea consultar y analiza si existe en la posción de la tabla que obtuvo como resultado, si es afirmativo retorna el registro de dato, sino informa que la clave no existe.

El programa a continuación, es un ejemplo sencillo que implementa el hash estátco:

Archivo: **“hashs.cpp”**

```
1       #include <iostream.h>
2       #include <conio.h>
3       //Hash B sico, sin tratamiento de Colisiones
4       const int tth=10;//Tama¤o de la tabla Hash
5       const int VACIO=-1;
6
7       //Estructura del Nodo de la Tabla Hash
8       struct nodo{
9                int clave;
10               int dato;
11               nodo()
```

```
12              {
13               dato=VACIO;
14               clave=VACIO;
15              }
16            };
17
18     int dispersa(int);
19     void put(nodo*,int,int);
20     void remove(nodo*,int);
21     int get(nodo*,int);
22     void mostrarAll(nodo*);
23     void freeTabla(nodo*&);
24
25     void main(void)
26     {
27      nodo* tabla=new nodo[tth];
28
29      //Agrego 1,5,9,8
30      put(tabla,1000,1000);
31      put(tabla,1,1);
32      put(tabla,5,5);
33      put(tabla,9,9);
34      put(tabla,8,8);
35      put(tabla,13,13);
36
37      mostrarAll(tabla);
38      put(tabla,18,18);
39      put(tabla,29,56);
40
41      get(tabla,13);
42
43      freeTabla(tabla);
44     }
45
46     void mostrarAll(nodo* tabla)
47     {
48      clrscr();
49
50      for(int i=0;i<tth;i++)
51      {
52       if(tabla[i].clave==VACIO)
53          cout<<"-"<<i<<"-: (X;X)\n";
54         else
55          cout<<"-"<<i<<"-: ("<<tabla[i].clave<<";"<<tabla[i].dato<<")\n";
56      }
57     }
58     void freeTabla(nodo*& tabla)
59     {
60      delete tabla;
61      tabla=NULL;
```

```
62      }
63
64      int dispersa(int key)
65      {
66       return key%tth;
67      }
68
69      void put(nodo* tabla,int clave, int dato)
70      {
71       if(get(tabla,clave))
72       {
73        cout<<"Esta intentando insertar una clave que existe: "<<clave;
74        return;
75       }
76
77       int posicionAInsertar=dispersa(clave);
78
79       if(tabla[posicionAInsertar].dato!=VACIO)
80        {
81         cout<<"\nColisi¢n Detectada entre "<<clave<<" y
        "<<tabla[posicionAInsertar].clave;
82         return;
83        }
84
85        tabla[posicionAInsertar].clave=clave;
86        tabla[posicionAInsertar].dato=dato;
87      }
88
89      int get(nodo* tabla, int clave)
90      {
91       int posicion=dispersa(clave);
92
93       if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
94        {
95         cout<<"\nLa Clave "<<clave<<" No se encuentra";
96         return 0;
97        }
98
99        cout<<"\nEl Dato de la Clave "<<clave<<" es "<<tabla[posicion].dato;
100       return 1;
101     }
102
103     void remove(nodo* tabla,int clave)
104     {
105      int posicion=dispersa(clave);
106
107      if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
108       {
109        cout<<"La Clave "<<clave<<" No se encuentra";
```

```
110         return;
111       }
112
113       tabla[posicion].clave=VACIO;
114       tabla[posicion].dato=VACIO;
115     }
```

Note que la función de dispersión es:

F(x)= (Clave)%(Tamaño de la Tabla)

Si recuerda el capítulo introductoria de C/C++, el '%' era un operador módulo que retornaba el resto de la división entre dos valores, lo que intuitivcamente y tan solo a los fines introductorias de la técnica de dispersión, es útil.

Vea el comportamiento si disponemos de una tabla de tamaño 10 y de valores entre 120 y 129:

Clave	Y=Clave%10
120	0
121	1
122	2
123	3
124	4
125	5
126	6
127	7
128	8
129	9

Vea que la función de dispersión, para la naturaleza y magnitud de datos analizados, nos es de utilidad, al menos a los fines de introducción.

Tratamiento de las Colisiones

Existen varias técnicas a analizar en caso de que ocurriese una colisión, para poder salvar dicha situación y almacenar un valor. En este apartado, las técnicas que analizaremos serán:

- Re dispersión,
- Barrido Secuencial,
- Lista Extensible o Encadenamiento Externo.

La Redispersión consiste en que, una vez detectada la colisión, se aplica una segunda función de dispersión a la clave, para analizar una nueva posición. Esta alternativa supondría que en lugar de obtener un orden '1' en la consulta, deba obtenerse en algunos casos, como máximo, un orden '2' en la búsqueda, debido a que es factible que existan valores redispersados almacenados en la tabla.

El Barrido Secuencial, en lugar de redispersar la llave una vez detectada la colisión, inicia una lectura secuencial a partir de la posición de la tabla donde se

reconoció la colisión, hasta encontrar un lugar vacío donde almacenar la nueva llave. Esta técnica, supone que en la peor situación de búsqueda s obtenga un orden (n-1), donde 'n' en este caso sería el tamaño de la tabala hash.

Una aclaración muy importante, es que tanto la redispersión como el Barrido Secuencial, no generan nuevas posiciones donde almacenar registros de datos, por lo que en caso de llenarse la tabla, no sera factible almacenar un nuevo dato.

La Lista Extensible, o Encadenamiento Externo, supone que ante una colisión, se almacenará el registro de dato en una lista enlazada cuya raíz reside en la posicón de la tabla dispersada por la función. Esto posibilita otorgarle cierto dinamismo a la técnica a costas de incrementar el orden de la búsqueda. Se incrementa el orden ya que, salvo el caso en donde la lista tenga un nodo, deberá consultarse cada nodo de la lista para saber si está el valor o no, durante una operación de búsqueda.

¿Mecanismos para salvar esta dificultad y/o disminuir el orden en la búsqueda? Claro que sí, hasta el momento se vieron:

- Árboles Binarios,
- Árboles AVL,
- Árboles Multidireccionales de Arriba hacia Abajo,
- Árboles B,
- Búsqueda Binaria (si dispone de una lista ordenada),
- Podría usarse una nueva tabla hash con encadenamiento externo,
- O la variante que prefiera.

La raíz de los árboles o el puntero a la nueva tabla hash, residirían en la posición inicial donde dispersó la función de hash.

Gráficamente podría interpretarse comosigue:

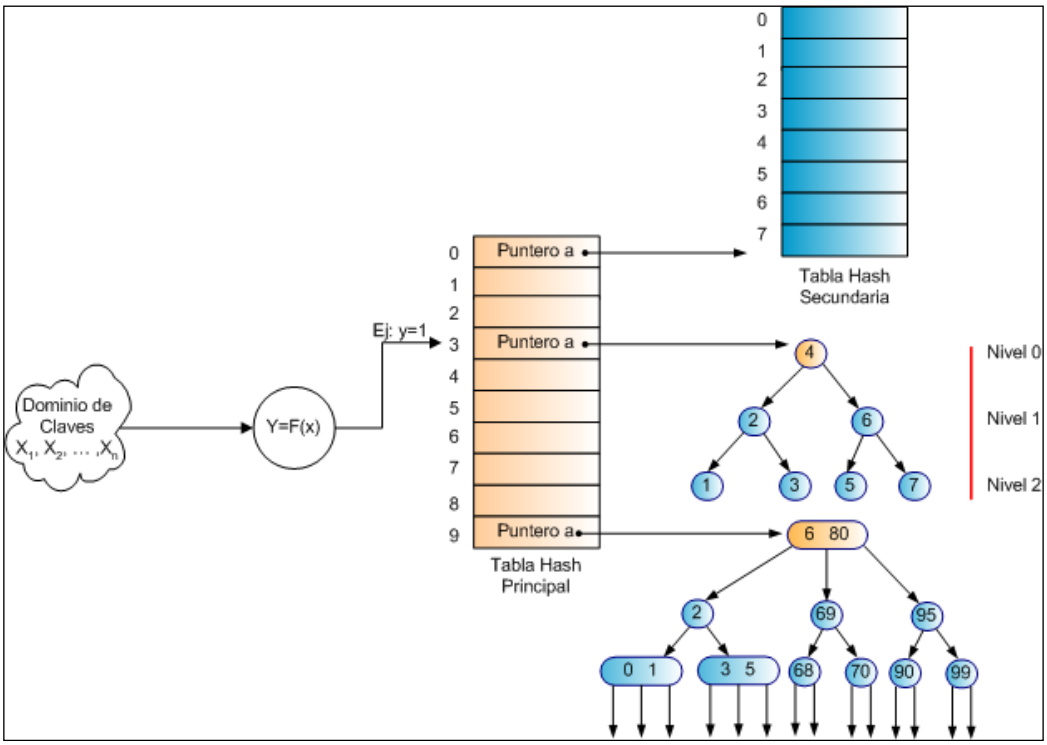


Figura 140: "Concepto de Hash Extendido"

Redispersión

El programa a continuación implementa la alternativa de redispersión, para salvar la dificultad de la colisión. El mismo trabaja de la siguiente manera:

1. Dispersa la clave, si no detecta colisión almacena el valor y termina,
2. Si detectó colisión, lo redispersa. Si detecta colisión nuevamente al redispersar, avisa que no es posible almacenar el registro, pero si al redispersar encuentra lugar, almacena el dato.

Note que la redispersión efectúa, en nuestra implementación, un solo intento de redispersión, pero usted podría indicar que se efectúe 'x' cantidad de intentos de redispersión hasta encontrar un lugar vacío o bien, darse por vencido con una condición dada (Ej: más de cinco redispersiones fallidas -que las posiciones estaban ocupadas en todos los intentos-, implica que el dato no se almacenará en la tabla).

La función de redispersión utilizada en la implementación del apartado, es la siguiente:

F(x)=(clave-2)/(tamaño de la tabla)

Obviamente la intención de la anterior función, es clarificar al lector cual es la idea de la redispersión y, no ingresar en un detalle pormenorizado de las características matemáticas de una función de redispersión.

Supongamos que insertamos el 10 en una tabla de tamaño 10, la posición asignada sería la 0 (10%10=0). Suponga Ahora que desea insertar el 20, esto produciría una colisión, ya que la posición inicial a insertarse sería la cero. Al detectarse la colisión, se utiliza la función de redispersión (20-2)%10 lo que nos deuelve la posición 8 allí, suponiendo que esté libre, se insertará el dato.

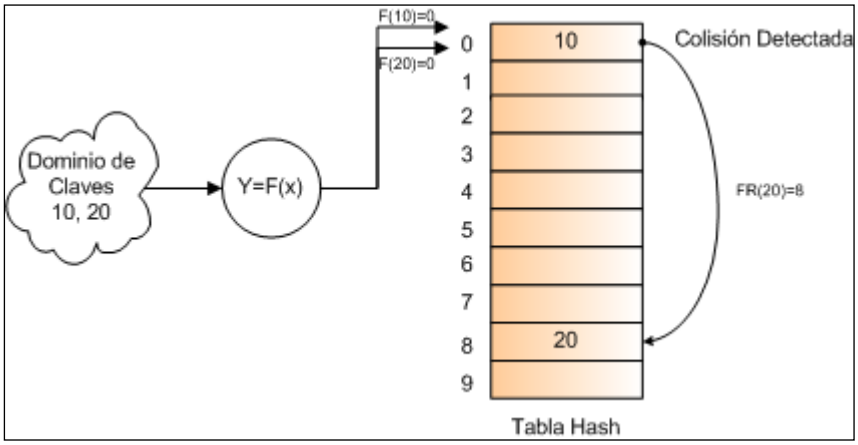


Figura 141: "Concepto de Hash con Redispersión"

En el gráfico anterior FR, representa la utilización de la función de redispersión en 20, lo que sería (20-2)%10=8. Mientras que en forma análoga, F(20) representaría la utilización de la función de dispersión con la llave 20, (20)%2=0.

Archivo: **“rehashs.cpp”**

```
1       #include <iostream.h>
2       #include <conio.h>
3       //Hash B sico, sin tratamiento de Colisiones
4       const int tth=10;//Tama¤o de la tabla Hash
5       const int VACIO=-1;
6       const int true=1;
7       const int false=0;
8
9       //Estructura del Nodo de la Tabla Hash
10      struct nodo{
11                int clave;
12                int dato;
13                nodo()
14                {
15                 dato=VACIO;
16                 clave=VACIO;
17                }
18              };
19
20      int dispersa(int);
21      int redispersa(int);
22      void put(nodo*,int,int);
23      void remove(nodo*,int);
24      int get(nodo*,int);
25      void mostrarAll(nodo*);
26      void freeTabla(nodo*&);
27
28      void main(void)
29      {
30       nodo* tabla=new nodo[tth];
31       clrscr();
32       //Agrego 1,5,9,8 y 13
33       put(tabla,1000,1000);
34       put(tabla,1,1);
35       put(tabla,5,5);
36       put(tabla,9,9);
37       put(tabla,8,8);
38       put(tabla,13,13);
39
40       put(tabla,18,18);
41       put(tabla,29,56);
42       mostrarAll(tabla);
43
44       put(tabla,39,56);
45
46       freeTabla(tabla);
47
```

```
48      }
49
50      void mostrarAll(nodo* tabla)
51      {
52       cout<<"\n\n";
53
54       for(int i=0;i<tth;i++)
55       {
56        if(tabla[i].clave==VACIO)
57            cout<<"-"<<i<<"-: (X;X)\n";
58           else
59            cout<<"-"<<i<<"-: ("<<tabla[i].clave<<";"<<tabla[i].dato<<")\n";
60       }
61      }
62      void freeTabla(nodo*& tabla)
63      {
64       delete tabla;
65       tabla=NULL;
66      }
67
68      int dispersa(int key)
69      {
70       return key%tth;
71      }
72
73      int redispersa(int key)
74      {
75       return ((key-2)%tth);
76      }
77
78      void put(nodo* tabla,int clave, int dato)
79      {
80       if(get(tabla,clave))
81       {
82        cout<<"Esta intentando insertar un valor que existe: "<<clave;
83        return;
84       }
85
86       int posicionAInsertar=dispersa(clave);
87
88       if(tabla[posicionAInsertar].dato!=VACIO)
89        {
90          posicionAInsertar=redispersa(clave);
91
92         if(tabla[posicionAInsertar].dato!=VACIO)
93         {
94          cout<<"\nColisi¢n Detectada entre "<<clave<<" y
        "<<tabla[posicionAInsertar].clave<<" A pesar de la Redispersi¢n";
95          return;
96         }
```

```
97        }
98
99        tabla[posicionAInsertar].clave=clave;
100       tabla[posicionAInsertar].dato=dato;
101     }
102
103     int get(nodo* tabla, int clave)
104     {
105      int posicion=dispersa(clave);
106
107      if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
108       {
109        posicion=redispersa(clave);
110
111        if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
112         {
113          cout<<"La Clave "<<clave<<" No se encuentra";
114          return 0;
115         }
116       }
117
118       cout<<"El Dato de la Clave "<<clave<<" es "<<tabla[posicion].dato;
119       return 1;
120     }
121
122     void remove(nodo* tabla,int clave)
123     {
124      int posicion=dispersa(clave);
125
126      if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
127       {
128        posicion=redispersa(clave);
129
130        if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
131         {
132          cout<<"La Clave "<<clave<<" No se encuentra";
133          return;
134         }
135       }
136
137       tabla[posicion].clave=VACIO;
138       tabla[posicion].dato=VACIO;
139     }
```

Barrido Secuencial

La Técnica del barrido secuencial es una alternativa más al momento de solucionar el inconveniente de las colisiones. Esta técnica tiene la particularidad de asegurar la completa utilización de la tabla, pero tal particularidad y la foema en la que se implementa, hacer perder la eficiencia en la búsqueda, debido a que en caso de que la función de dispersión no encuentre el valor en la posición retornada, se inicia una lectura seriada de la tabla completa, lo que incrementa el tiempo de respuesta.

La Técnica trabaja de la siguiente manera en el caso de una inserción:

1. Dispersa la llave, si la posición en la tabla correspondiente a la posición dispersada está vacía, se guarda el dato y termina el algoritmo.
2. Si se produjo una colisión al dispersar, se comienza una búsqueda seriada desde la posición n+1, siendo n=F(x) inicial.
3. Si llego desde (n+1) al final de la tabla hash y no encontro espacio libre, se consulta si 'n' era cero originalmente (lo que implicaría que no existe espacio libre en todala tabla), de serlo se termina el algoritmo y no existe más espacio.
4. Si 'n' era distinto de cero, inicio un nuevo recorrido desde cero hasta n-1 (la sección de la tabla que falto recorrer), si en ese intervalo encuentro espacio, guardo el dato, de lo contrario y ahora sí *definitivamente*, no existe más espacio.

Grñaficamente, el comportamiento de esta alternativaes como sigue:

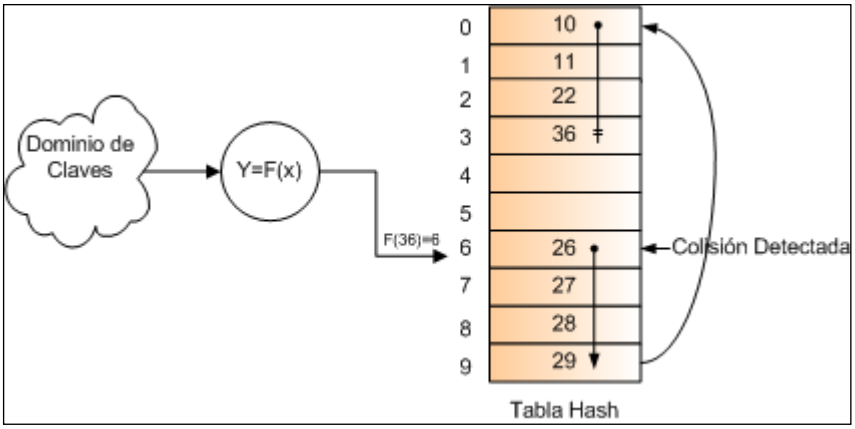


Figura 142: "Concepto de Hash con Búsqueda Secuencial"

Note que cuando se intenta insertar el '36', su posición está ocupada y se produce la colisión. A partir de allí inicia la búsqueda de espacio lbre, cuando llega al final de la tabla pregunta ¿la búsqueda la inicié desde cero? Como se sabe que se inició desde 6, entonces se inicia una nueva búsqueda desde 0 hasta 5 inclusive, pero la misma se detiene en 3 ya que encuentra allí un espacio donde almacenar el dato.

El siguiente programa, denominado hashbs.cpp, implementa la lógica de hashing con búsqueda secuencial.

Archivo: **"hashbs.cpp"**

```
1       #include <iostream.h>
2       #include <conio.h>
3       //Hash B sico, sin tratamiento de Colisiones
4       const int tth=10;//Tama¤o de la tabla Hash
5       const int VACIO=-1;
6       const int true=1;
7       const int false=0;
8
9       //Estructura del Nodo de la Tabla Hash
10      struct nodo{
11                int clave;
12                int dato;
13                nodo()
14                {
15                 dato=VACIO;
16                 clave=VACIO;
17                }
18              };
19
20      int dispersa(int);
21      int redispersa(int);
22      void put(nodo*,int,int);
23      void remove(nodo*,int);
24      int get(nodo*,int);
25      void mostrarAll(nodo*);
26      void freeTabla(nodo*&);
27      int busquedaSecuencial(nodo*,int,int);
28      int getFreeSpaceSecuencial(nodo*,int);
29
30      void main(void)
31      {
32       nodo* tabla=new nodo[tth];
33       clrscr();
34       //Agrego 1,5,9,8 y 13
35       put(tabla,1000,1000);
36       put(tabla,1,1);
37       put(tabla,5,5);
38       put(tabla,9,9);
39       put(tabla,8,8);
40       put(tabla,13,13);
41
42       put(tabla,18,18);
43       put(tabla,29,56);
44       mostrarAll(tabla);
45
46       put(tabla,39,56);
47       get(tabla,18);//note que el 18 est  en 6, por lo que inicialmente busca
```

```
48                    //desde 8 a 9 y luego, desde 0 a 7, lo encuenrtra
finalmente
49                    //en i=6, por lo que nunca llega a 7.
50
51      get(tabla,100);//Avisa que la clave no existe
52
53      freeTabla(tabla);
54
55     }
56
57     void mostrarAll(nodo* tabla)
58     {
59      cout<<"\n\n";
60
61      for(int i=0;i<tth;i++)
62      {
63       if(tabla[i].clave==VACIO)
64          cout<<"-"<<i<<"-: (X;X)\n";
65         else
66          cout<<"-"<<i<<"-: ("<<tabla[i].clave<<";"<<tabla[i].dato<<")\n";
67      }
68     }
69     void freeTabla(nodo*& tabla)
70     {
71      delete tabla;
72      tabla=NULL;
73     }
74
75     int dispersa(int key)
76     {
77      return key%tth;
78     }
79
80     int redispersa(int key)
81     {
82      return ((key-2)%tth);
83     }
84
85     void put(nodo* tabla,int clave, int dato)
86     {
87      if(get(tabla,clave))
88       {
89        cout<<"\nEsta intentando insertar una clave que existe: "<<clave;
90        return;
91       }
92
93      int posicionAInsertar=dispersa(clave);
94
95      if(tabla[posicionAInsertar].dato!=VACIO)
96       {
```

```
97          posicionAInsertar=redispersa(clave);
98
99         if(tabla[posicionAInsertar].dato!=VACIO)
100        {
101         posicionAInsertar=getFreeSpaceSecuencial(tabla,posicionAInsertar);
102
103         if(posicionAInsertar==-1)
104         {
105          cout<<"No Existe m s espacio en la tabla donde insertar el "<<clave;
106          return;
107         }
108        }
109       }
110
111       tabla[posicionAInsertar].clave=clave;
112       tabla[posicionAInsertar].dato=dato;
113     }
114
115     int get(nodo* tabla, int clave)
116     {
117      int posicion=dispersa(clave);
118
119      if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
120       {
121        posicion=redispersa(clave);
122
123        if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
124         {
125          posicion=busquedaSecuencial(tabla,posicion,clave);
126
127          if(posicion==-1)
128          {
129           cout<<"\nLa Clave "<<clave<<" No se encuentra";
130           return 0;
131          }
132         }
133       }
134
135       cout<<"El Dato de la Clave "<<clave<<" es "<<tabla[posicion].dato;
136       return 1;
137     }
138
139     void remove(nodo* tabla,int clave)
140     {
141      int posicion=dispersa(clave);
142
143      if(tabla[posicion].dato==VACIO || tabla[posicion].clave!=clave)
144       {
145        posicion=busquedaSecuencial(tabla,posicion,clave);
```

```
146
147       if(posicion==-1)
148       {
149        cout<<"La Clave "<<clave<<" No se encuentra";
150        return;
151       }
152      }
153
154      tabla[posicion].clave=VACIO;
155      tabla[posicion].dato=VACIO;
156    }
157
158    int busquedaSecuencial(nodo* tabla,int posInicial,int claveABuscar)
159    {
160     if(tabla==NULL) return -1;
161
162     int tamtabla=tth;
163     int again=0;
164     do{
165        for(int i=posInicial;i<tamtabla;i++)
166        {
167         if(tabla[i].clave!=VACIO && tabla[i].dato==claveABuscar)
168            return i;
169         }
170
171        if(tamtabla==tth)
172        {
173         if(posInicial==0) return -1;//no se encuentra la llave
174          else
175          {
176           tamtabla=posInicial;
177           posInicial=0;//barro desde el inicio de la tabla hasta posinicial
178          }
179        }
180        else again=0;//fin del segundo barrido
181
182      }while(again!=0);
183
184      return -1;//retorno fantasma
185    }
186
187    int getFreeSpaceSecuencial(nodo* tabla,int posInicial)
188    {
189     if(tabla==NULL) return -1;
190
191     int tamtabla=tth;
192     int again=0;
193     do{
194        for(int i=posInicial+1;i<tamtabla;i++)
195        {
```

```
196         if(tabla[i].clave==VACIO)
197            return i;
198         }
199
200        if(tamtabla==tth)//implica que es el primer barrido de tabla
201        {
202         if(posInicial==0) return -1;//no se encuentra la llave
203          else
204          {
205           tamtabla=posInicial;
206           posInicial=0;//barro desde el inicio de la tabla hasta posinicial
207           again=1;
208          }
209        }
210        else again=0;//fin de barrido
211
212      }while(again!=0);
213
214      return -1;//retorno fantasma
215    }
```

Encadenamiento Externo

A diferencia de los anteriores mecanismos de salvaguarda de colisiones, el hashing extendido dinámicamente, emplea alguna estructura dinámica para extender la capacidad de almacenamiento de datos de la posición direccionada por la tabla hash.

En los casos anteriores, ante una colisión, se trataba o bien de redispersar o de buscar otra posición en la misma tabla, aquí simplemente se encadena el nuevo valor de alguna forma dentro de la posición de la tabla hash, dispersada por la función hash.

En esta implementación, se utilizó una lista simplemente vinculada con inserción primaria (al inicio), pero podría haberse utilizado un árbol binario, un árbol AVL o bien un árbol mulidireccional para almacenar el nuevo valor pero, por cuestiones introductorias no viene al caso hacerlo, pero no debiera implicar mayores inconvnientes, teniendo en cuenta los temas analizados hastael momento.

Algritmo de Inserción

El algoritmo de inserción dispersa la llave, mediante el empleo de la función hash o de dispersión, y localiza la posición a insertar el nuevo valor.

Detectada la posición, existen dos posibilidades inicialmente:

1. *Que la posición esté vacía*: con lo que simplemente creamos un nodo en forma dinámica, almacenamos los datos en el mismo y vinculamos el puntero de la tabla hash en la posición dispersada a éste.
2. *Que la posición esté ocupada*: Esta situación implica que existe una lista direccionada por el puntero de la tabla hash en la posición dispersada. En forma similar al punto uno, se crea dinámicamente el nodo y se ingresan los valores en los campos respectivos. Al puntero "*next*" del nodo recientemente creado, se le asigna la dirección del primer nodo de la lista apuntado por la tabla hash, y luego se le asigna al puntero de la tabla hash en la posición

dispersada la dirección del nodo creado recientemente. De esta forma, la tabla queda apuntando al nuevo nodo, y el puntero "next" del nuevo nodo, queda apuntando al antiguo primer nodo de la lista (ahora ya segundo).

Suponga la siguiente secuencia de inserción 1000, 1, 5, 9, 8, 13, 18, 28, 29 y 39.

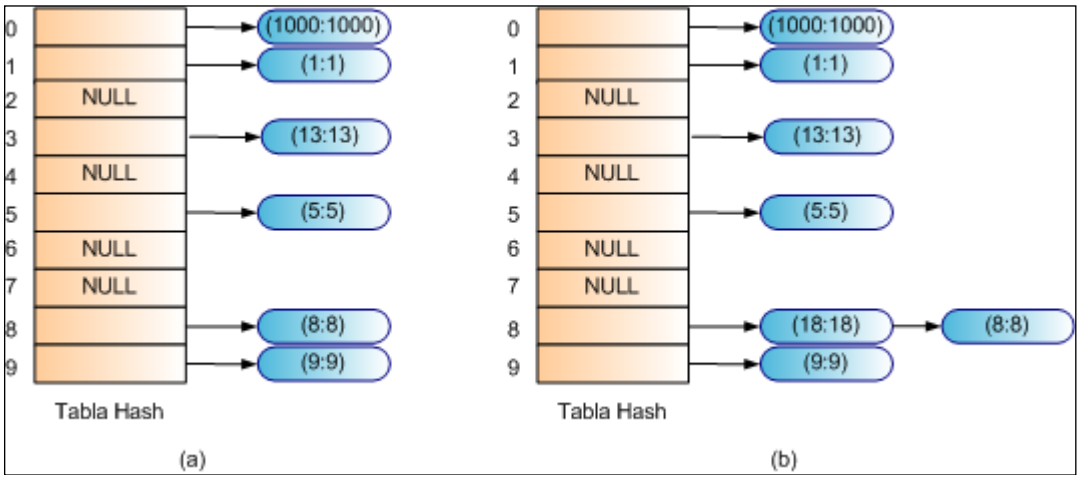


Figura 143: "Concepto de Hash con Extensión Dinámica, Situación 'a' y 'b'"

Vea que en el caso (a), se insertaron las llaves 1000, 1, 5, 9, 8 y 13 sin mayores inconvenientes, ya que no se presento colisión alguna. Note una diferencia importante con los anteriores métodos de dispersión, los valores están encadenados a la tabla hash y almacenados externamente en sus respectivos nodos vinculados, y ya no se almacenan mas dentro de la misma, como se efectuaba anteriormente. Esta ventaja, permite que los valores correspondientes a una misma posición de la tabla hash, puedan ser almacenados y gestionados en cualquier tipo de estructura de datos que decida el desarrollador, ya que el puntero de la tabla hash podría apuntar a algún árbol AVL, binario, Digital, Multidireccional, otra tabla hash, entre otras estructuras; que contendrán las claves de dicha posición.

El caso (b), plantea la situación que se genera al insertar la llave 18, lo que provoca una colisión. Ésta técnica no se preocupa mucho de esta situación, ya que simplemente encadena el nuevo valor al inicio de la tabla y, los nodos pre existentes, se asocian al campo "next" del nuevo nodo (ahora desde el 18 se apunta al nodo con llave 8). Tenga en cuenta que es un ejemplo introductorio, pero no debiera, en términos heurísticos, existir más de tres nodos en la lista por posición de la tabla hash. En caso de que existan más de tres nodos por posición de la tabla hash, se desvitruaría "el orden k" de búsqueda del algoritmo de hashing, debido al recorrido secuencial en la lista de la posición de la tabla, para localizar un valor determinado. Puede modificar el algoritmo propuesto, el cual implementa la lógica de inserción anteriormente mencionada, en el programa "hashext.cpp" listado posteriormente, para hacerlo tan sofisticado como desee.

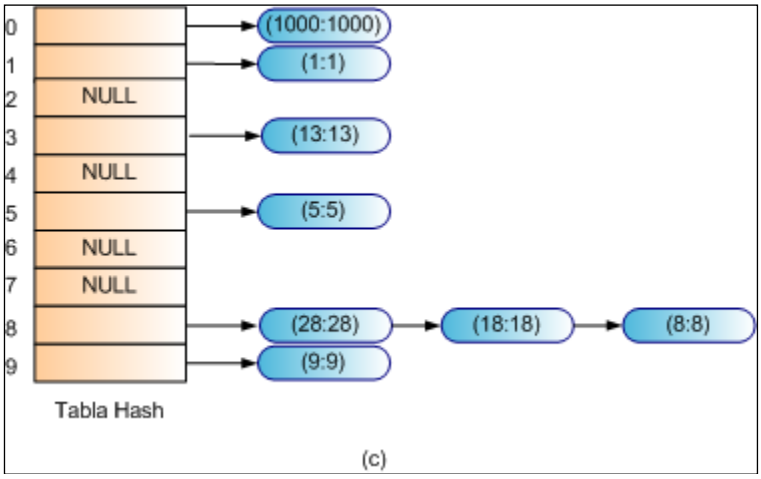


Figura 144: "Concepto de Hash con Extensión Dinámica, Situación 'c'"

Vea que al insertar la llave 28, ésta se encadena directamente a la tabla hash en la posición 8 (devuelta por la función de dispersión), y los nodos que componían la antigua lista, se encadenan al puntero "next" del nodo con la llave 28.

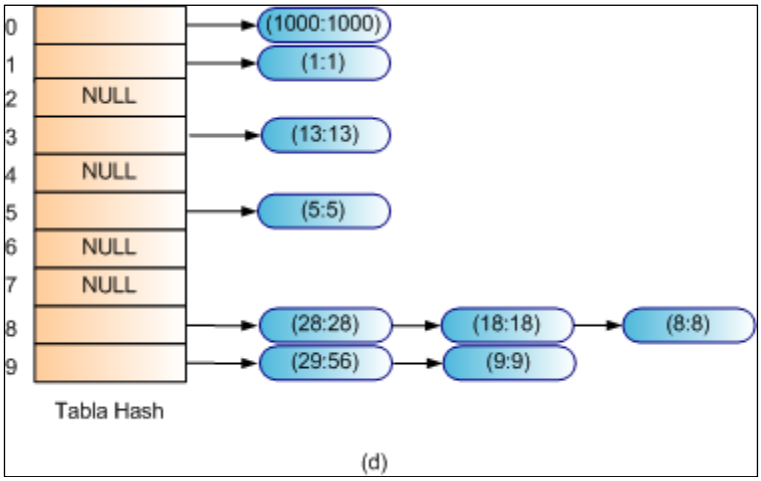


Figura 145: "Concepto de Hash con Extensión Dinámica, Situación 'd'"

En forma análoga a la inserción del 28, la inserción del 29, provoca un encadenamiento del nodo que lo contiene directamente con la tabla y, un encadenamiento de la lista pre existente al nodo "next" del nodo con la llave 29.

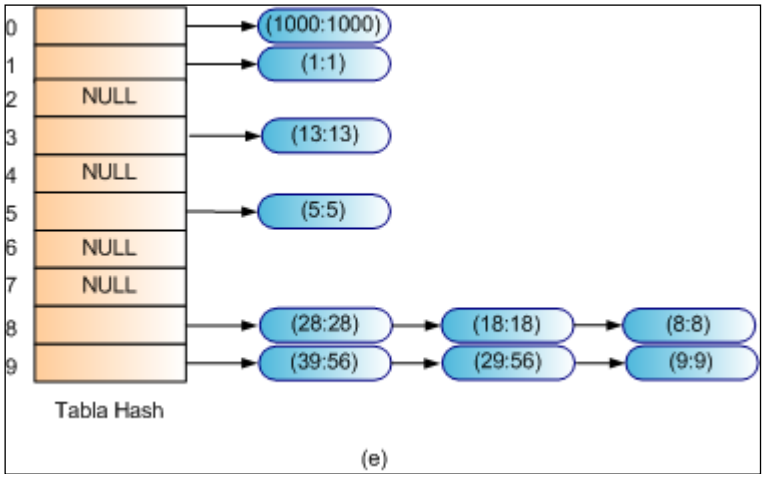


Figura 145: "Concepto de Hash con Extensión Dinámica, Situación 'd'"

Finalmente, el último valor de la secuencia de inserción es el 39, quien al insertarse, produce un desplazamiento de los nodos de la lista antiguamente direccionada por el puntero de la tabla hash en la posición 9. Ahora el nodo con la llave 39 se encadena directamente con la tabla hash en la posición 9, mientras que su puntero "next" apunta al antiguo primer nodo de la lista, el 29.

Algoritmo de Borrado

El algoritmo de borrado localizará el nodo a eliminar en dos pasos:

1. *El primer paso,* es localizar la posición donde se encuentra la lista que tentativamente, debería contener el valor a eliminar. Esto último se logra a través de la función de dispersión.
2. *El segundo paso*, consiste en recorrer la lista simplemente enlazada en forma secuencial, para eliminar el nodo creado dinámicamente, manteniendo la consistencia de los apuntadores de la lista. La consitencia en los apuntadores en la lista, implica que los punteros queden correctamente direccionando a los nodos posteriores y previos, sin que el borrado de un nodo implique la rotura de un enlace en otro.

Al localizar el nodo a eliminar, existen básicamente dos situaciones:

1. *Que el nodo a eliminar se encuentre al inicio de la lista:* en esta situación se encadena la tabla en la posición dispersada, con el segundo nodo de la lista. Una vez actualizado el encadenamiento, se libera la memoria (elimina) ocupada por el nodo objeto de eliminación.

   Suponga que, una vez insertado los valores mencionados en el algoritmo de inserción, se desea eliminar el nodo con la llave 39, vea gráficamente, en base a lo dicho en el párrafo anterior, lo que sucedería con los nodos:

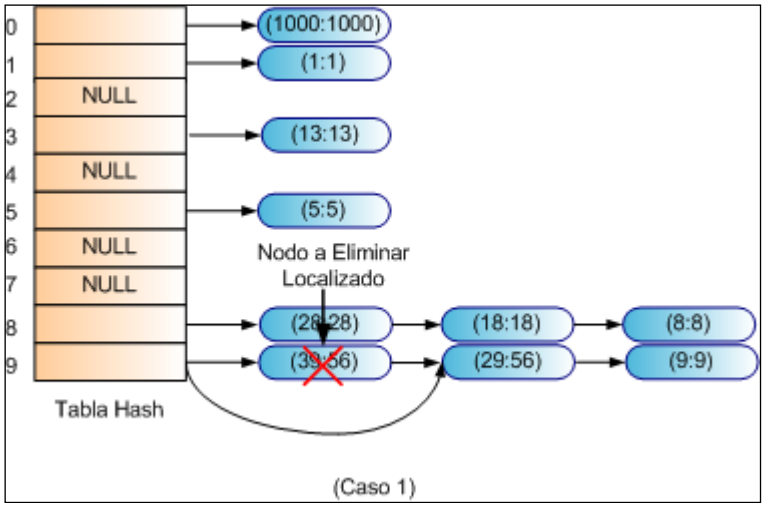


Figura 146: "Concepto de Hash con Extensión Dinámica, Eliminación del primer nodo de la lista"

2. Que el nodo a eliminar se encuentre al final de la lista o bien, se encuentre en algún lugar intermedio de la lista. En esta situación, se debe colocar el puntero "next" del anterior nodo a aquel a eliminar, apuntando al nodo que apunta el puntero "next" del nodo a eliminar. En el caso particular de que se trate del último nodo de la lista, cuando el puntero "next" del nodo previo al que se eliminará, apunte al mismo nodo que el puntero "next" del nodo a eliminar, en realidad estará apuntando a NULL, con lo que se indica el fin de la lista.

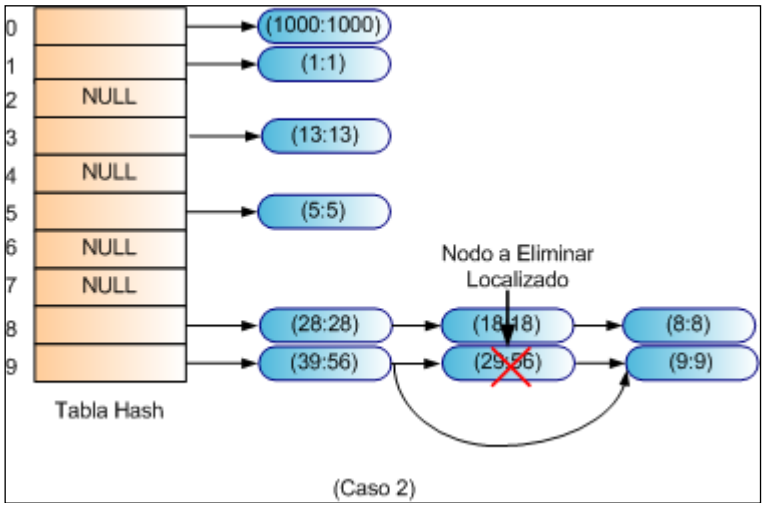


Figura 147: "Concepto de Hash con Extensión Dinámica, Eliminación de un nodo intermedio de la lista"

En la figura anterior, se ve claramente el reacomodamiento del puntero previo al nodo a eliminar, y como pasa éste a direccionar la posición del nodo con la llave 9 (el nodo posterior al que se elimina).

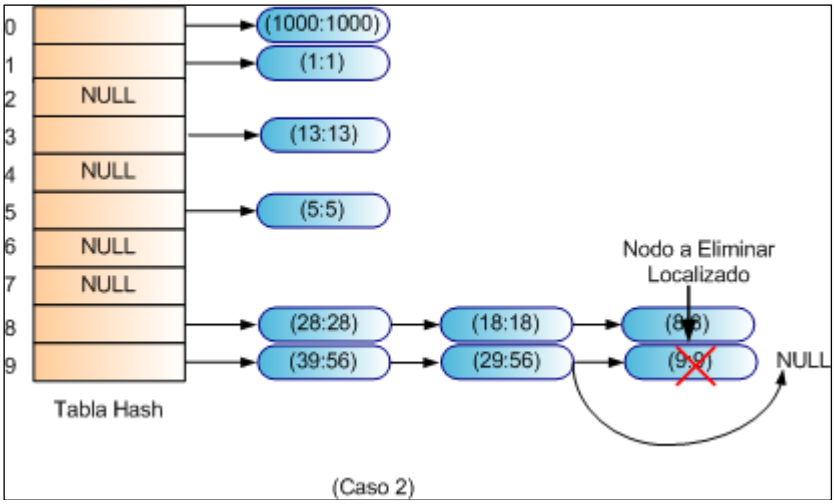


Figura 148: "Concepto de Hash con Extensión Dinámica, Eliminación del último nodo de la lista"

Cuando se intenta eliminar el último nodo de la lista, el nodo previo (29), direccionará al nodo posterior del que se eliminará, pero como 9 no tiene nodo posterior sino que su puntero "next" es NULL, el puntero "next" del nodo con la llave 29 (el previo al nodo a eliminar) apuntará a NULL, señalando que ahora es él quien señalará el final de la lista. Como verá, este es un caso particular de la eliminación de un nodo intermedio, pero es importante tenerlo en cuenta, ya que implica un cambio en el nodo "bandera"del final de lista.

Implementación del Encadenamiento Externo

El programa "hashext.cpp", implementa una tabla hash con encadenamiento externo. Para el encadenamiento externo el programa utiliza una lista simplemente vinculada, donde los nuevos nodos son insertados al inicio, encadenando los nodos preexistentes al nuevo nodo.

Archivo: **"hashext.cpp"**

```
1       #include <iostream.h>
2       #include <conio.h>
3       //Hash B sico, sin tratamiento de Colisiones
4       const int tth=10;//Tama¤o de la tabla Hash
5       const int VACIO=-1;
6       const int true=1;
7       const int false=0;
8
9       //Estructura del Nodo de la Tabla Hash
10      struct nodo{
11              int clave;
12              int dato;
13              nodo* next;
14
15              nodo()
16              {
17               dato=VACIO;
18               clave=VACIO;
```

```
19                next=NULL;
20               }
21             };
22
23     int dispersa(int);
24     void put(nodo**,int,int);
25     void remove(nodo**,int);
26     int get(nodo**,int);
27     void mostrarAll(nodo**);
28     void freeTabla(nodo**&);
29
30     void main(void)
31     {
32
33      nodo** tabla=new nodo*[tth];
34      //Inicializo la tabla hash
35      for(int i=0;i<tth;i++)
36       tabla[i]=NULL;
37
38      //Limpio pantalla
39      clrscr();
40
41      //Agrego 1,5,9,8 y 13
42      put(tabla,1000,1000);
43      put(tabla,1,1);
44      put(tabla,5,5);
45      put(tabla,9,9);
46      put(tabla,8,8);
47      put(tabla,13,13);
48
49      put(tabla,18,18);
50      put(tabla,28,28);
51      put(tabla,29,56);
52      put(tabla,39,56);
53      remove(tabla,39);
54      mostrarAll(tabla);
55      get(tabla,18);//note que el 18 est  en 6, por lo que inicialmente busca
56                    //desde 8 a 9 y luego, desde 0 a 7, lo encuenrtra
finalmente
57                    //en i=6, por lo que nunca llega a 7.
58
59      get(tabla,100);//Avisa que la clave no existe
60
61      freeTabla(tabla);
62
63     }
64
65     void mostrarAll(nodo** tabla)
66     {
```

```
67      cout<<"\n\n";
68
69      for(int i=0;i<tth;i++)
70      {
71       if(tabla[i]!=NULL)
72       {
73        nodo* p=tabla[i];
74
75        int j=0;
76        while(p!=NULL)
77        {
78          cout<<"("<<i<<"-"<<j<<"): ("<<p->clave<<";"<<p->dato<<")\n";
79          p=p->next;
80          j++;
81        }
82       }
83       else
84       {
85        cout<<"<VACIO "<<i<<">\n";
86       }
87      }
88     }
89
90     void freeTabla(nodo**& tabla)
91     {
92      for(int i=0;i<tth;i++)
93      {
94       nodo* p=tabla[i];
95
96       nodo* q=p;
97       while(p!=NULL)
98       {
99        q=p;
100       p=p->next;
101       delete q;
102       q=NULL;
103      }
104     }
105
106     delete tabla;
107     tabla=NULL;
108    }
109
110    int dispersa(int key)
111    {
112     return key%tth;
113    }
114
115    void put(nodo** tabla,int clave, int dato)
116    {//Inserta el nuevo nodo al inicio de la lista
```

```
117     if(get(tabla,clave))
118     {
119      cout<<"\nEst  intentando insertar un valor que ya existe";
120      return;
121     }
122
123     int posicionAInsertar=dispersa(clave);
124
125     nodo* nuevo=new nodo();
126     nuevo->clave=clave;
127     nuevo->dato=dato;
128     nuevo->next=tabla[posicionAInsertar];
129     tabla[posicionAInsertar]=nuevo;
130    }
131
132    int get(nodo** tabla, int clave)
133    {
134     int posicion=dispersa(clave);
135
136     if(tabla[posicion]==NULL)
137      {
138       cout<<"\nLa llave "<<clave<<" no existe en la tabla";
139       return 0;
140      }
141
142      nodo* p=tabla[posicion];
143      while(p!=NULL)
144      {
145       if(p->clave==clave)
146         {
147          cout<<"El Dato de la Clave "<<clave<<" es "<<p->dato;
148          return 1;
149         }
150
151       p=p->next;
152      }
153
154      cout<<"\nLa llave "<<clave<<" no existe en la tabla";
155      return 0;
156    }
157
158    void remove(nodo** tabla,int clave)
159    {
160     int posicion=dispersa(clave);
161
162     if(tabla[posicion]==NULL)
163      {
164       cout<<"\nLa Llave "<<clave<<" no existe en la tabla";
165       return;
```

```
166      }
167
168      nodo* p=tabla[posicion];
169      nodo *q=NULL;
170
171      while(p!=NULL)
172      {
173       if(p->clave==clave)
174       {
175         if(q==NULL)//'p' es el primer nodo de la lista extendida
176          {
177            tabla[posicion]=p->next;
178            delete p;
179            p=NULL;
180          }
181         else
182         {//'p' es intermedio o final en la lista extendida
183           q->next=p->next;
184           delete p;
185           p=NULL;
186         }
187         return;//retorno luego de eliminar
188       }
189
190       q=p;
191       p=p->next;
192      }
193
194      cout<<"\nLa Clave "<<clave<<" No se encuentra";
195    }
196
```

Referencias

- "Estructuras de Datos con C y C++"; Langsam, Aungenstein y Tenenbaum; Editorial Prentice Hall; México, 1997.

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995.

- "The Art of Computer Programming", Volumen 3: Sorting and Searching, Segunda Edición; D.E. Knuth, Editorial Addison-Wesley, 1975.

- "Árboles"; Prof. Dr. Eric Jeltsch F; Material Asociado a la Currícula 'Diseño y Análisis de Algoritmos con Java', Ingeniería en Computación, Universidad de la Serena; Chile, 2004.

- "Hash"; Prof. Dr. Eric Jeltsch F; Material Asociado a la Currícula 'Diseño y Análisis de Algoritmos con Java', Ingeniería en Computación, Universidad de la Serena; Chile, Mayo de 2004.

- "The Evolution of Effective B-Tree: Page Organization and Techniques: a personal account"; David Lomet; ACM SIGMOD Record, Volumen 30, Edición 3; Septiembre de 2001.

7

Búsqueda de Cadenas y Reconocimientos de Patrones

Introducción a la Búsqueda de Cadenas y Recnonocimientos de Patrones

En muchas ocasiones cuando se lee un texto, es normal que se efectúe una lectura secuencial del mismo y, a medida que se avanza, se efectuan anotaciones y/o se generan índices que permitan localizar rápidamente un tema.

Pero este capítulo no trata sobre la generación de índices, para ello refiérase al capítulo anterior, este capítulo lo introducirá en la búsqueda de cadena y el reconocimiento de patrones.

Suponga esta situación: "Está a punto de terminar de leer est párrafo y recuerda que en esta hoja algo se dijo respecto a 'secuencial'", lógicamente tratará de buscar desde el comienzo la palabra 'secuencial' y, cuando la localice habrá encontrado el resto del párafo de su interés. Esta última operatoria, enunciada en forma sencilla, es una búsqueda de cadena y un reconocimiento de patrón, donde el texto que contenía las palabras era la presnete página, el patrón que usted quería identificar era la palabra "Secuencial", por lo que hay algunos aspectos interesantes a destacar de esta situación de búsqueda y reconocimiento:

- Se dispone de un Texto que contiene una secuencia de caracteres Ej: la presente página.
- El Texto tendrá una longitud definida y calculable, Ej: La cantidad de Caracteres en el Texto,
- Se dispone de un patrón a localizar en el tezto E: 'Secuencial',
- El patrón está bien definido y su longitud es calculable, Ej: 'Secuencial', con una longitud de 10 caracteres (sin contar '\n'),
- La búsqueda tiene un objetvo concreto "Localizar la posición inicial donde se reconoce el patrón"

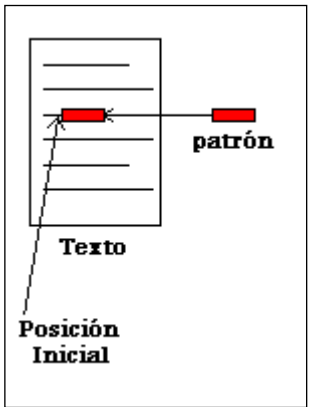


Figura 149: "Concepto de Búsqueda y Reconocimiento de Patrones"

Se puede definir formalmente al Reconocimiento de Patrones, como el *"proceso mediante el cual se intenta ubicar las posiciones de ocurrencia del testigo dentro de su universo de búsqueda, partiendo de que el testigo y el universo de búsqueda se generaron en base al mismo alfabeto"*. El tesitgo no es ni más ni menos que el patrón

que deseamos localizar, y el universo de búsqueda sería la página donde se efectúa la búsqueda.

Algoritmo de Fuerza Bruta

Un algoritmo muy utilizado, sobre todo en los procesadores de textos, es el algoritmo de fuerza bruta. Este algoritmo recorre el texto, desde la posición inicial, en forma secuencial hasta reconocer el patrón de búsqueda, al hacerlo retorna la posición del primer carácter del patrón dentro del texto.

A continuación se resume una implementación en C/C++ del algoritmo:

Archivo: **"bp_fbrut.cpp"**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4
5       enum bool{false,true};
6
7       int busquedafb(char*,char*);
8
9       void main(void)
10      {
11       char *texto="Con Esfueuerzo";
12       char *patron="uer";
13
14       clrscr();
15       cout<<"El patron \""<<patron<<"\" aparece en
        "<<busquedafb(texto,patron);
16      }
17
18      int busquedafb(char* texto,char* patron)
19      {
20       bool partida=true;
21       int first=-1;//almacena la primera ocurrencia
22       int i,j;
23       int TT=strlen(texto);
24       int TP=strlen(patron);
25
26       for(i=0,j=0;j<TP && i<TT;i++)
27        if(texto[i]==patron[j])
28        {
29         first=(partida)?i:first;
30         partida=false;
31         j++;
32        }
33        else      {
34         if(j>0) i=first;//Se venia inspeccionando parcialmente el patron, al
35                 // debo riniciar entonces la inspección a partir de (first+1)
36         j=0;
37         first=-1;
```

```
38          partida=true;
39         }
40
41       if(j==TP) return first;
42
43       return -1;//no encontro
44      }
```

En la implementación del algoritmo de fuerza bruta del programa "bp_fbrut.cpp", la función *busquedafb*, recibe dos parámetros:

- El parámetro "texto": es un puntero al texto que representa el universo de búsqueda,
- El parámetro "patron": es un puntero a la cadena de caracteres que representa el patrón a localizar en texto.

A partir de los dos parámetros enunciados, a través de la función strlen ("Cuenta la cantidad de caracteres de una cadena, excepto el '\n'"), se calcula:

- TT: Tamaño o Longitud total del Universo de Búsqueda,
- TP: Tamaño o Longitud del Patrón,

Estos valores, permitirán controlar el ciclo for como sigue:

- Se avanzará en la cadena direccionada por "texto" hasta encontrar la primera ocurrencia del patrón,
- Encontrada la primera ocurrencia, en las sucesivas iteraciones irá evaluando si los caracteres que continúan en el texto luego de la primera ocurrencia coinciden con los caracteres que continúan en el texto del patrón.
    - Si coinciden todos ellos, y se reconoce el patrón, llegará un momento en que "j" será igual a TP, lo que provoca la salida del ciclo de reconocimiento por encontrar la primera ocurrencia,
    - Si alguno de los valores posteriores a la primera ocurrencia difiere,
        - Se mantiene el puntero al universo de búsqueda en el carcater (first+1),
        - Se reinicializa 'j' (contador de coincidencias consecutivas), para comenzar a partir de la posición (first+1) en el texto, nuevamente la búsqueda.

La implementación del algoritmo, emplea dos variables auxiliares:

- **Partida:** del tipo enumeración, emula un dato booleano. Si es "true" y se produce una coincidencia, se sabe que es la primera y se almacena en la variable auxiliar "first" el valor actual de "i",
- **First:** del tipo entero, almacena la posición, en el texto, del primer carácter que coincide con el patrón.

						0	1	2												
						u	e	r												
						0	1	2	3	4	5	6	7	8	9	10	11	12	13	
partida	**first**	**TT**	**TP**	**i**	**j**	C	o	n		E	s	f	u	e	u	e	r	z	o	
true	-1	14	3	0	0	C														
false	7			1			o													
				2				n												
				3																
				4						E										
				5							s									
				6								f								
				7	1								u							
				8	2									e						
true	-1			7	0															
				8																
false	9			9	1										u					
false	9			10	2											e				
false	9	14	**3**	11	**3**												r			

Figura 150: “Análisis del Algoritmo de Fuerza Bruta”

En el ejemplo de la figura anterior, nuestro universo de búsqueda es la cadena “Con Esfueuerzo” y, el patrón de a reconocer será “uer”.

Al iniciar, la variable “i”, que conduce la lectura en el universo de búsqueda, avanza hasta la posición 7 donde encuentra la coincidencia del carácter del texto con el del patrón, como “partida” es trae, almacena 7 en first y asigna ‘false’ a partida. El carácter 8 coincide con el patrón por lo que j se incrementa y vale 2, cuando se analiza la posición 9 del universo de búsqueda, se percata de que difiere con el patrón y como j<TP, implica que aún no se reconoció el patrón, por lo que se lleva a ‘i’ a valer “first” (para que cuando de el final de vuelta del for, se lo deje en la posición (first+1) nuevamente), se coloca first en -1 (indicando que aún no se reconoción patrón alguno) y se enciende la bandera de “partida”.

Se Reinicia la búsqueda a partir de (first+1), para evitar el solapamiento de patrones, Analice el siguiente caso:

Texto	**1010**100111
Patrón	**1010**0111

Si prosiguiera analizando a partir de la posición donde se produjo la diferencia (el quinto 1 de izquierda a derecha en ‘Texto’), simplemente no se encontraría el patrón ¿por qué? Porque al reiniciar la comparación a partir del punto de diferencia se compararía:

Texto	1010**100**111
Patrón	**10**100111

Vea que el contenido anterior del texto se despreció al reiniciarse la comparación desde el punto de diferencia, lo cual es un error porque se desprecio un subpatrón coincidente con la cadena (posición 3 de la cadena “Texto”) que era la llave para reconocerla ¿Dónde?

Texto	1010100111
Patrón	10100111
Texto	1010100111
Patrón	10100111

En el recuadro anterior, se ve claramente que el patrón coincide con la cadena de texto a partir de la posición 3, de allí que ante una diferencia en la comparación, se reinicia desde first+1 la nueva comparación y no desde la posición donde se produjo la diferencia.

El porque se utilizan a TT y TP como reguladores del ciclo pasa por no exceder los límites de los arreglos del universo de búsqueda y del patrón de comparación respectivamente.

El Algoritmo de Fuerza Bruta, en el peor de los casos, efectuará TP*TT comparaciones. ¿Cuál sería el peor de los casos para el ejemplo anterior? Realice una prueba de escritorio buscando el patrón "uer", dentro de un texto que contiene todas "u", calcule al final de la ejecución del algoritmoel total de comparaciones y verá que se aproxima a TP*TT comparaciones.

Algoritmo de Knuth-Morris-Pratt

En 1976 se publica el Algoritmo de Knuth, Morris y Pratt que intenta solucionar un inconveniente fundamental del algoritmo de fuerza bruta en el reconocimiento de patrones. "El retroceso obligatorio hasta la posición first+1".

Como se analizó en el apartado anterior, el algoritmo de fuerza bruta retrocede a 'i' hasta la posición *first+1*, en caso de encontrar una diferencia entre el patrón y la cadena que venía analizando. Se mencionó que no era posible continuar el análisis desde el punto de la diferencia porque se podía correr el riesgo de saltear la parte inicial del patrón oculta entre la posición first y la posición que se produjo la diferencia.

Hnuth, Morris y Pratt, percatados de este inconveniente, le dieron solución a través de una tabla de proximidad y autoemparejamiento del patrón, por ende dicha tabla depende exclusivamente del patrón. Veamos cuales son las reglas que los autores impusieron para la creación de la misma:

- Haga una Copia del Patrón de los 'j' primeros caracteres en forma imaginaria (donde j es la posición del patrón donde se produce la diferencia),
- Coloque el primer carácter de la copia sobre el segundo carácter del patrón,
- Deslice una copia de izquierda a derecha hasta que:
  1. coincidan todos los caracteres del patrón ó
  2. no haya ninguno

Supongamos que queremos construir la tabla de proximidad y autoemparejamiento para el patrón "uer",

j	
1	uer
	uer

El texto con fondo gris haga de cuenta que no existe, se colocó a los fines de que vea puntualmente el desplazamiento imaginario que se mencionó, pero tenga en cuenta que usted solo visualiza los 'j' primeros caracteres del patrón, como j es 1, solo vería la 'u', al iniciar el desplazamiento imaginario desde la izquierda en la posición 2, ya no se superpone carácter alguno (de los que poseen fondo blanco), entonces significa que si la diferencia entre el patrón y el texto donde se busca éste, es en la posición 1 del patrón (por ende no existe emparejamiento), deberá retrasarse 'j' a 0, en consecuencia se continúa analizando el patrón desde la posición '0' para el resto del texto.

Vea ahora qué ocurre si la diferencia entre el texto y el patrón se produce en la posición 1 (En C/C++ 1 es la posición 2 del arreglo), j es 2:

J		
2	uer **u**er	¿Los caracteres solapados coinciden totalmente? No, los únicos caracteres solapados son 'e' y 'u', al ser distintos, sigue el desplazamiento imaginario a la derecha
	uer uer	¿Los Caracteres solapados coinciden totalmente? No existen más caracteres solapados, por ende, no hay coincidencias y no existió emparejamiento del patrón.

¿Qué significa que no exista emparejamiento en la posición 1 (j=2) del arreglo que contiene el patrón? Que no existe riesgo de que, al continuar desde la posición en que se produjo la diferencia con la búsqueda del patrón, se saltee alguna parte inicial del mismo contenido entre la posición que se denominaba first (Algoritmo de Fuerza Bruta) y aquella donde efectivamente se produjo la diferencia. ¿Cómo se sabe que no existe riesgo? Porque al producirse la diferencia en la posición 1 (j=2) del arreglo, y como se vio en el ejemplo anterior, no emparejaba consigo mismo el patrón en los sucesivos desplazamientos imaginarios, por ende se continua analizando el patrón desde la posición 0 para el resto del texto.

Y si la posición donde se produce la diferencia es en la 2 (j=3), vea como se analizan los desplazamientos imaginarios:

j		
3	uer uer	¿Los caracteres solapados coinciden totalmente? No, en la posición dos, 'e' es distinta de 'u'. Esto último es condición suficiente para seguir con el corrimiento hacia la derecha.
	uer uer	¿Los Caracteres solapados coinciden totalmente? No, 'r' es distinto de 'u'. Esto es condición suficiente para continuar el corrimiento.
	uer uer	¿Los Caracteres solapados coinciden totalmente? No existen más caracteres solapados, por ende, no hay coincidencias y no existió emparejamiento del patrón.

Esto implica, que en caso de producirse la diferencia entre el texto y el patrón en la tercera posición del patrón, se continuará analizando desde la posición '0' del patrón al texto remanente, debido a la falta de emparejamiento del patrón para j igual a 3.

De esta forma el arreglo auxiliar, denominado "prox" en el programa "bc_kmp.cpp", tendrá:

- Una longitud de tres enteros, ya que el patrón contiene tres caracteres,
- En base a lo calculado, tendrá los siguientes valores en cada posición:

j		
0	**1**	**2**
0	0	0

¿Cómo se lee el contenido del arreglo para el patrón "uer"?

- Si en el proceso de comparación entre el texto y el patrón, se produce una diferencia en la primera posición (0 en C/C++, y j=1 en la tabla anteriormente analizada), se continuará analizando la cadena de caracteres remanente en "texto", desde la posición actual 'i' (sin retrasos, a diferencia del algoritmo de fuerza bruta), pero comparándola desde la posición **'0'** del patrón.
- Si en el proceso de comparación entre el texto y el patrón, se produce una diferencia en la segunda posición (1 en C/C++, y j=2 en la tabla anteriormente analizada), se continuará analizando la cadena de caracteres remanente en "texto", desde la posición actual 'i' (sin retrasos, a diferencia del algoritmo de fuerza bruta), pero comparándola desde la posición **'0'** del patrón.
- Si en el proceso de comparación entre el texto y el patrón, se produce una diferencia en la tercera posición (2 en C/C++, y j=3 en la tabla anteriormente analizada), se continuará analizando la cadena de caracteres remanente en "texto", desde la posición actual 'i' (sin retrasos, a diferencia del algoritmo de fuerza bruta), pero comparándola desde la posición **'0'** del patrón.

Vea la implementación en C/C++ para el algoritmo de Knuth-Morris-Pratt, mediante el programa denominado "bc_kmp":

Archivo: "**bc_kmp.cpp"**

```
1       #include <iostream.h>
2       #include <conio.h>
3       #include <string.h>
4
5       enum bool{false,true};
6
7       int busquedakmp(char*,char*,int*);
8       void inicprox(char*,int*);//inicializa el arreglo de corrimientos
9
10      void main(void)
11      {
12       char *texto="Con Esfueuerzo";
13       char *patron="uer";
14       int *prox=new int[strlen(patron)];
15
16       clrscr();
17       cout<<"El patron \""<<patron<<"\" aparece en
        "<<busquedakmp(texto,patron,prox);
```

```
18      }
19
20      int busquedakmp(char* texto,char* patron,int* prox)
21      {
22       int i,j;
23       int TT=strlen(texto);
24       int TP=strlen(patron);
25
26       inicprox(patron,prox);
27       for(i=0,j=0;j<TP && i<TT;i++,j++)
28             while((j>=0) && (texto[i]!=patron[j])) j=prox[j];
29
30       if(j==TP) return i-TP;
31
32       return -1;//no encontro
33      }
34
35      void inicprox(char* p,int* prox)
36      {
37       int i,j;
38       int TP=strlen(p);
39
40       prox[0]=-1;
41       for(i=0,j=-1;i<TP; i++, j++, prox[i]=j)
42         while((j>=0) && (p[i]!=p[j])) j=prox[j];
43      }
```

Como recomendación útil para comprender a fondo la generación del arreglo de proximidad y emparejamiento, elija una cantidad ‘n’ de patrones y efectúe una prueba de escritorio con la función *“inicprox”*, que implementa el mecanismo de inicialización de dicho arreglo, para cada uno de ellos.

Si quiere analizar un caso en que es importante el algoritmo y donde, sí existe solapamiento de la clave suponga que el texto donde buscará es:

“10101111011” y el patrón a buscar es “1011”

“El Algoritmo de Búsqueda de Cadenas de Knuth-Morris-Pratt nunca efectúa más de TT+TP comparaciones de caracteres”

Esto último se deduce de la lectura del código,’i’ nunca se Decrementa, j se incrementa o se reinicializa a partir del valor en la tabla ‘prox’, una vez por cada ‘i’.

Algoritmo de Boyer-Moore

El Algoritmo de Boyer-Moore, de orden TP+TT (donde TP es el tamaño del patrón y TT es el tamaño del Texto) en el peor de los casos, se basa en una "tabla de corrimientos" al igual que el de Knuth-Morris-Pratt. La diferencia consiste en que la comparación del patrón se efectúa de *derecha a izquierda y, ante una diferencia en el carácter del texto y del patrón se efectúa lo siguiente:*

- El carácter del texto que difiere, suponga en una posición 'i' del texto, ¿Se encuentra en el patrón?
- Si sí, se busca la primera ocurrencia de derecha a izquierda y se retorna la magnitud del salto (a continuación se analizará cómo se determina dicha magnitud),
- Obtenida la magnitud del salto, la que denominamos 's', se desplaza el patrón en 's' posiciones a la derecha y se compara la posición extrema derecha del patrón con el de la posición 'i+s', reiniciándose el ciclo en caso de diferencias entre los caracteres, solo que ahora el nuevo 'i' es igual a 'i+s'.

Cálculo del Desplazamiento a Derecha del Patrón

El Desplazamiento a Derecha del patrón depende del patrón mismo y considerando que el análisis del patrón con el texto se hace de derecha a izquierda, las magnitudes de desplazamiento se obtienen como sigue:

Suponga un patrón 'uer':

		***(e)**	
u	e	r	
	u	e	r

Suponga que la difiere el primer carácter de derecha a izquierda (j=1), y en lugar de una 'r' el texto tiene una 'e'. En forma análoga al algoritmo del apartado anterior, se coloca una copia del patrón coincidiendo el penúltimo carácter de la copia con el último carácter del patrón, y se inicia el desplazamiento de izquierda a derecha hasta que coincidan todos, o ninguno, de los caracteres solapados. En este caso antes de iniciar el desplazamiento coinciden todos los caracteres solapados, ya que al ser j igual a uno (se considera un carácter) y, 'e' del texto es igual a 'e' de la copia desplazada, la pregunta es ¿Cuánto se desplazó la copia? 1 (uno), entonces implica que la magnitud del desplazamiento del patrón, si encontrase una 'e' en el texto, será de 1.

¿Y si en lugar de una 'e' hubiese aparecido una 'u'? Vea que sucede:

			***(u)**		
	u	e	r		
Primer Desplazamiento		u	e	r	
Segundo Desplazamiento			**u**	e	r

Si hubiese una 'u' en el texto que difiere con 'r' en una posición 'i', efectuó el mismo procedimiento:

- Realizo una copia del patrón, y hago coincidir la penúltima posición de la copia con la última posición del patrón,
- Desplazo a la derecha la copia del patrón, hasta que me coincida la totalidad de los caracteres de 'j' longitud (contando desde la derecha), donde 'j' es la posición del patrón, donde se produjo la diferencia entre el texto y éste.
- Si empareja totalmente los 'j' caracteres entre el texto y el patrón, la cantidad de desplazamientos de la copia será la magnitud del desplazamiento del patrón en el texto, mientras que si no existieron coincidencias, se desplaza en 3 (magnitud del patrón).

Si el carácter que difiere con 'r' no es ni 'u' ni 'e', es aún más simple, se desplaza directamente el patrón a la derecha en TP unidades, donde TP es el tamaño del patrón. Este último desplazamiento no implica riesgo de pérdida de subpatrones, ya que si la letra que difiere con el útlimo carácter a la derecha del patrón, no es siquiera una de las contenidas en el patrón, no existe posibilidad de solapamiento de patrones porque simplemente no coincide con ninguna subparte del patrón.

Todas estas situaciones analizadas, es asumiendo que la diferencia se produce en el último carácter por lo que j es igual a 1, pero veamos el siguiente caso donde el patrón es:

10110101

¿Cómo calcular el desplazamiento si j>=2?, es decir si la diferencia no se produce en el primer carácter de derecha a izquierda del patrón. La respuesta implica el empleo del mismo mecanismo visto con anterioridad, pero considerando los 'j' caracteres de derecha a izquierda (anteriormente se consideraba 1 porque la diferencia se produjo en el último carácter del patrón, esto implicaba j=1):

Suponga que la diferencia se produjo en el penúltimo carácter del patrón, **j=2**:

						***(1)**								***(1)**
1	0	1	1	0	1	0	1							
	1	0	1	1	0	1	0	1						
		1	0	1	1	0	1	0	1					
			1	0	1	1	0	1	0	1				
				1	0	**1**	**1**	0	1	0	1			

Si la diferencia se produce entonces en el penúltimo carácter, y el valor en el texto es 1 (como se señala en el encabezado de la tabla), se desplazará a la derecha el patrón tantas veces como haga falta hasta que coincidan los 'j' últimos caracteres (2 en este caso) o hasta que no coincida ninguno. Si se encuentra una coincidencia, la magnitud del desplazamiento del patrón, está dado por la cantidad de desplazamientos efectuados en la copia del patrón, que en este último caso fue 4, si no se encontrase coincidencias, se desplaza a la derecha el patrón en TP (Recuerde que TP era el tamaño del patrón)

Veamos que ocurre si la diferencia entre el patrón y el texto se produce en el tercer carácter desde la derecha del patrón, es decir j=3:

					*(0)									*(1)
1	0	1	1	0	1	0	1							
	1	0	1	1	0	1	0	1						
		1	0	1	1	0	1	0	1					
			1	0	1	1	0	1	0	1				
				1	0	1	1	0	1	0	1			
					1	0	1	1	0	1	0	1		
						1	0	1	1	0	1	0	1	
							1	0	1	1	0	1	0	1

Note que en este caso la coincidencia se encontro al final, luego de 7 movimientos, por lo que se determina de que en caso de que la diferencia se produzca en el tercer carácter del patrón desde la derecha, se desplazará el patrón en 7 unidades. Si usted piensa ¿Si coincidió un elemento únicamente? Ocurre que el ejemplo fue intencionado, los elementos que se comparan son 3 pero en el último caso se solapa uno en la copia, que coincide con su análogo en el patrón, por ende "Todos los elmentos solapados coinciden". Note que en el anterior caso se solapaban dos elementos, pero seguían sin coincidir, era '01' vs. '10', recién en el último caso existió coincidencia.

Si la diferencia se produce en el cuarto carácter del patron desde la derecha, j=4:

				*(1)										*(1)
1	0	1	1	0	1	0	1							
	1	0	1	1	0	1	0	1						
		1	0	**1**	**1**	**0**	**1**	0	1					

Vea que en esta situación, bastaron dos movimientos para que coincidan los 'j-ésimos últimos caracteres del patrón y con los solapados de la copia desplazada', de esta forma si se produce la diferencia en el cuarto carater desde la derecha del patrón, éste deberá desplazarse en dos unidades.

Y de esta misma forma para 'j' mayor a cuatro y menor a 9.

Análisis del Funcionamiento Lógico del Algoritmo

Suponiendo el texto "Con Esfueuerzo" y el patrón 'uer', vea como trabaja el algoritmo de Moyer-Moore:

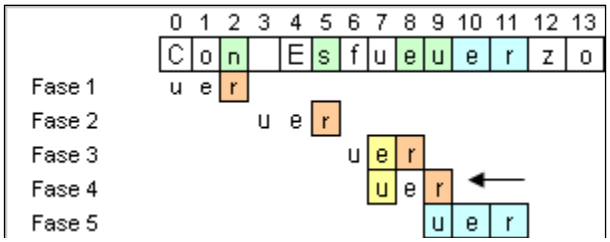


Figura 151: "Análisis del Algoritmo de Boyer-Moore"

Iicialmente se compara la 'r' con la 'n', como difieren y la 'n' no se encuentra contenida en el patrón, se desplaza en la magnitud del patrón, en 3. Luego se compara la posición i+TP (como i era 2 y TP era 3, i+TP es 5), con la última posición del patrón, al

diferir la ‘s’ con la ‘r’ y la primera no está contenida en el patrón, se desplaza en en la magnitud del patrón. La nueva posición a comparar, i+TP, es ahora 5 más 3, 8.

En 8, ‘e’ difiere de ‘r’ pero ‘e’ está contenida en el patrón, la tabla de salto nos mencionaba que de encontrarla el patrón debería desplazarse en una unidad y volver a realizar la comparación nuevamente, esto lleva a que la nueva posición a comparar sea i+1, 9.

En 9, ‘u’ difiere de ‘r’ pero está contenida en el patrón, la tabla de desplazamientos del patrón nos indicaba que en esta situación deberíamos de desplazar el patrón en 2 unidades, por lo que la nueva posición ‘i’ a comparar será el ‘i’ actual mas dos, 11.

En 11, al coincidir ‘r’, se analizan los caracteres a la izquierda para:

- Confirmar que coinciden totalment, en consecuencia se encontro patrón ó
- Hubo diferencias en alguno, por lo que debe de analizarse el desplazamiento,

En este último caso se encontro el patrón, por lo que culmina su búsqueda.

Implementación del Algoritmo de Boyer-Moore en C/C++

En base a lo mencionado, y acplicado el algoritmo a cadenas de caracteres, se expone a continuación una implementación funcional en C/C++ del algoritmo de Boyer-Moore, a través del programa “bc_bm.cpp”:

Archivo: **“bc_bm.cpp”**

```
1       //Algoritmo de Boyer-Moore para B£squeda de Cadenas
2
3       #include <iostream.h>
4       #include <conio.h>
5       #include <string.h>
6
7       enum bool{false,true};
8
9       int busquedabm(char*,char*,int*);
10      void inicsaltos(char*,int*);//inicializa el arreglo de corrimientos
11      int fsaltar(int*,char*,char);
12
13      void main(void)
14      {
15       char *texto="Con esfueuerzo";
16       char *patron="uer";
17       int *saltar=new int[strlen(patron)];
18
19       clrscr();
20       cout<<"El patron \""<<patron<<"\" aparece en
        "<<busquedabm(texto,patron,saltar);
21      }
22
23      int busquedabm(char* texto,char* patron,int* saltar)
24      {
25       int i,j,t;
```

```
26      int TT=strlen(texto);
27      int TP=strlen(patron);
28
29      inicsaltos(patron,saltar);
30      for(i=TP-1,j=TP-1;j>0;i--,j--)
31             while(texto[i]!=patron[j])
32             {
33               t=fsaltar(saltar,patron,texto[i]);
34               i+=((TP-j) >t)? TP-j:t;
35
36               if(i>=TT) return -1;//no esta el patron
37               j=TP-1;
38             }
39
40      return i;//no encontro
41     }
42
43     void inicsaltos(char* p,int* saltos)
44     {
45       int TP=strlen(p);
46
47       for(int i=0;i<TP;i++)
48          saltos[i]=TP-i-1;//inicializo los corrimientos a la derecha
49     }
50
51     int fsaltar(int *saltos,char* patron,char val)
52     {
53       //Busca de derecha a izquierda (de menor a mayor corrimiento)
54       //porque puede darse que exista en el patron una letra repetida,
55       //por ende si los anteriores valores coincidieron debe realizar
56       //el menor desplazamiento posible a la derecha para evitar la perdida
57       //de un subpatron
58
59       int TP=strlen(patron);
60
61       if(TP==0) return 0;
62
63       for(int j=(TP-1);j>=0;j--)
64          if(patron[j]==val) return saltos[j];
65
66      return TP;
67     }
```

Introducción al Reconocimiento Teórico de Patrones

El Reconocimiento de Patrones va más alla de una búsqueda de patrones, consiste en la interpretación de una expresión regular y su consiguiente localización en un universo de búsqueda. Entiéndase por una expresión regular a un conjunto de

símbolos del dominio de valores, que conforma el universo de búsqueda, combinados con operaciones fundamentales.

Dentro de los elementos que constituyen el dominio de valores para el universo de búsqueda correspondiente, se pueden realizar tres operaciones fundamentales:

- **Concatenación:** Dado dos cadenas y/o expresiones 'a' y 'b', se dice que

  a.b o ab

  Es igual a una nueva cadena formada en primer lugar por los caracteres o componenetes de la expresión 'a', seguidos de los caracteres o componenetes de la expresión 'b'.
  La Concatenación cumple, como cualquier operación fundamental, algunas propiedades:

  1. *Conmutatividad:* No es lo mismo 'a.b' que 'b.a'. Se prueba lo dicho a través del siguiente contra ejemplo, suponga que:
     a='Hola'
     b='Alumnos'

     Entonces a.b="HolaAlumnos" y b.a="AlumnosHola"
  2. *Asociatividad:* La Concatenación si es asociativa, debido a que la asociatividad se refiere a la vinculación entre elementos adyacentes **manteniendo su orden**. Suponga que:
     a='Hola'
     b='Alumnos'
     c='Informáticos'

     Entonces vea que:

     a.b.c= "HolaAlumnosInformáticos",
     (a.b).c → a.b="HolaAlumnos" → a.b.c="HolaAlumnosInformáticos"
     a.(b.c) → b.c="AlumnosInformáticos"→ a.b.c="HolaAlumnosInformáticos"

     Como verá, debido a que la asociatividad mantiene el orden de vinculación entre elementos, la concatenación no sufre alteración alguna.
  3. **Elemento Neutro:** La concatenación admite elemento neutro, esto implica que dentro del dominio de valores de los elementos, existe alguno de ellos que representa la inalterabilidad de un segundo elemento ante la aplicación de una operación fundamental con éste. En una base decimal por ejemplo, el elemento neutro en un producto es el '1', vea que:

     Ax1= A es equivalente 1xA=A

     En el tratamiento de cadenas, el elemento neutro sería '', lo que podría representarse como '*n*', así:

     A*n*=*nA*

Si A="Hola" y n='', entonces:

A*n*="Hola" y *n*A="Hola"

- **Unión (Or):** Esta operación permite especificar alternativas en un patrón determinado. La Unión permite la ocurrencia alternativa de dos expresiones dentro de un patrón determinado. Se suele representar con el símbolo '+' y se representa:

a+b

Donde 'a' y 'b' son expresiones formadas a partir del dominio de valores. En reconocimiento de patrones, la unión se instancia como sigue:

Suponga ques usted debe buscar el patrón "Aznar", "Asnar" y "Atnar" en un texto (El universo de búsqueda), puede correr los algoritmos de búsqueda de cadenas una vez con cada patrón, ó puede realizar un reconocimiento de patrones utilizando la siguiente expresión regular:

A(z+s+t)nar

Lo que significa que el patrón de texto a localizar dentro del universo de búsqueda, deberá cumplir con las reglas impuestas por la expresión regular, a saber:

1. El primer carácter del patrón **debe** ser 'A',
2. El segundo carácter del patrón **debe** ser una de las siguientes alternativas: 'z' o 's' o 't',
3. El patrón restante **debe** ser "nar",

De esta forma, se considerarán válidas las Cadenas "Aznar", "Asnar", "Atnares" y ¿Acnar cumple? No, viola la segunda regla, ya que la 'c' no está dentro de los valores especificados por la 'unión'.

- **Clausura:** Esta operación permite que algunas de las partes del patrón se puedan repetir arbitrariamente. Se suele reconocer al '*' como símbolo de la clausuray su notación sería:

A*

Donde 'A' representa una expresión o valor instanciada a partir del dominio de valores. Suponga que a la expresión regular utilizada en la operación de unión, se le agregase una clausura en sus alternativas, lo que llevaría a:

A(s+z+t)*nar

La clausura provoca que cualquier repetición arbitraria de los caracteres que integran la unión provoque una cadena válida, por lo que las reglas se rescriben como sigue:

1. El Primer Carácter **debe** ser 'A',
2. El Segundo carácter **debe** ser 's' o 'z' o 't' o cualquier repetición arbitraria de los mismos,
3. La Cadena remanente **debe** ser "nar",

De esta forma, cadena como "Asssssnar", "Assnar", "Atttnar", "Azzzzznar", "Assznar", "Azzzsssstttnar" son válidas. Tenga en cuenta que:

"No es lo mismo A(s+z+t)*nar que A(a*+z*+t*)nar"

Ya que en la última expresión regular la clausura trabaja individualmente sobre cada elemento y no sobre el grupo, como es el caso de la primera expresión regular.

Máquinas de Estados para el Reconocimiento de Patrones

Un estado representa un conjunto de variables con valores particulares asumidos en un momento dado y bien definidos. Por ejemplo: al momento de cursar una materia, se encuentra en estado de "Cursando", lo que implica que cumple con las correlativas para cursarla y tiene el porcentaje de asistencia por encima del mínimo exigido (caso contrario estaría 'Libre' y no 'Cursando').

Existen Eventos que provocan la transición o cambio de estados. Si usted se encontraba en estado 'Cursando', al momento de aprobar la cantidad d exámenes mínimos y cumplir con el porcentaje de asistencias, transita al estado 'Regularizado', caso contrario, de no cumplirse la asistencia o no aprobar ninguno de los exámenes parciales transita al estado 'Libre', si desaprobo uno de los examenes parciales, transita al estado 'Recupera Parcial', gráficamente podría verse la combinación de estados y eventos como sigue:

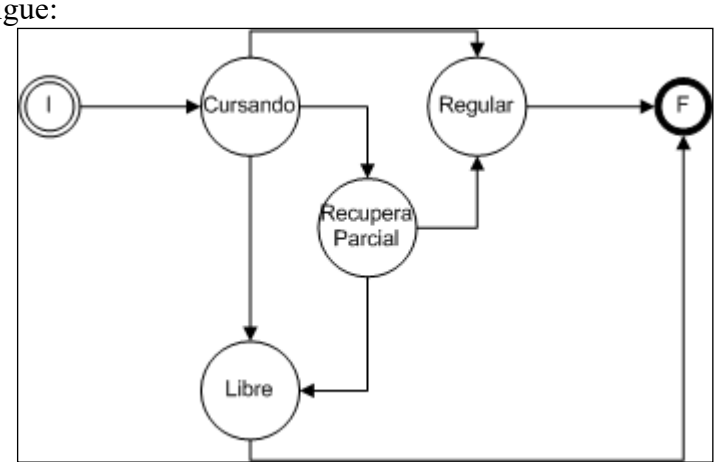


Figura 152: "Diagrama de Estados Básico de la Realización de una Materia"

El Diagrama que permite visualizar los estados posibles y la transición posible entre ellos, se denomina Diagrama de Estados. La Figura anterior muestra el diagrama básico de estados básico para la realización de una materia. El estado inicial se

simboliza por un doble círculo con una "I" en el centro, mientras que el estado final, se simboliza por un círculo, con marco negro con una "F"en el centro.

Vea que al iniciarse, si cumple con las condiciones de correlativas e inscripción, el alumno se encuentra en el estado "Cursando", desde allí puede transitar a tres estados diferentes:

1. Si no cumple el porcentaje mínimo de asistencia o es aplazado en más de la cantidad permitida de pariciales, transita al estado "Libre",
2. Si es aplazado en una cantidad permitida de parciales, transita al estado "Recupera Parcial",
3. Si aprobo los parciales, transita al estado "Regular".

Desde Libre, se transita al esado final cuando se pasan las planillas de regularidad al Departamento de alumnos al finalizar el dictado de la materia (se formaliza el estado). Del estado "Recupera Parcial" puede transitar a "Libre" si es aplazado en el recuperatorio o si no cumple el porcentaje mínimo de asistencia, o bien, transita al estado regular ya que aprobo el recuperatorio. Del estado Regular, se transita al estado final, cual el docente entrega la planilla de alumnos regulares a Departamento de Alumnos, donde se formaliza la regularidad.

De lo analizado, se pudo observar que:

- Existe un conjunto *finito* de estados, no vacío (Q),
- Se dispone de *un* estado inicial (q_o).
- Se disponde de *al menos un* estado final ($F \subset Q$).
- Existe un Vocabulario Definidos para los símbolos de entrada (T_e) (Ej: en el reconocimiento de patrones, utilizamos un conjunto de valores pertenecientes un dominio de valores, ese dominio de valores es el Vocabulario de Entrada),
- Existe un Vocabulario Definido para los símbolos de salida (T_s),
- Existe una función de transición (f), donde para cada valor de la entrada considerando el estado actual, existe un valor de salida,
- Existe una función de salida (g),

Se dice que un Autómata finito es una máquina de estados secuencial, definido por el sistema:

$$\mathbf{AF = (T_e, T_s, Q, f, g, q_o, F)}$$

Como en el caso de reconocimiento de patrones no es necesaria una salida formal por cada transición de estados antes una entrada determinada sino que la misma transición es la que nos informa en que etapa del reconocimiento se encuentra, se puede decir que $T_s = \lambda$ *(cadena vacía) y en consecuencia 'g' carece de sentido, ésto, se conoce como un Autómata Reconocedor, y queda definido como:*

$$\mathbf{AF = (T_e, Q, f, q_o, F)}$$

La Configuración Inicial en un autómata finito, se expresa como:

$$\mathbf{(q_o, w)}$$

Donde:

q_o: es el estado inicial
w: es la cadena que resta por reconocer

La Configuración final de un autómata finito, se expresa como:

$$(q_i, \lambda)$$

Donde:

q_i: es uno de los estados finales.
λ : Representa que la cadena que resta por reconocer es nula.

La Transición de un estado a otro se suele denominar ***'movimiento'*** y se representa como:

$$(q,aw) \rightarrow (q',w)$$

Donde:

(q,aw): representa el estado actual y la subcadena aún por reconocer y,
(q',w): representa el estado al que se transita y la subcadena por reconocer remanente.

Un Número no determinado de movimientos, se representa como "→*", por lo que se puede definir a un ***autómata finito determinista (AFD), como aquel que a partir de una cadena de entrada en una configuración inicial y, a través de una cantidad finita de movimientos, alcanza una configuración final***:

$$(q_o,w) \rightarrow (q_i, \lambda)$$

Aplicación de AFD al reconocimiento de Patrones

Si bien es posible utilizar un autómata finito no determinista para el reconocimiento de patrones, es un tema que escapa al alcance del presente texto y aquí se planteará introductoriamente, y a través de un ejemplo, el mecanismo teórico de reconocimiento de patrones empleando AFD.

Suponga que uste desea reconocer el patrón representado a través de la siguiente expresión regular:

A(s+t+z)*nar

Antes de construir la máquina de estados debemos evacuar las siguientes cuestiones:

- Cómo se desplaza el patrón en caso de discordancia en 'Asnar',
- Cómo se desplaza el patrón en caso de discordancia en 'Aznar',
- Cómo se desplaza el patrón en caso de discordancia en 'Atnar',

Con lo que se debe construir la tabla de desplazamientos vista para el algoritmo de Knuth-Morris-Pratt:

J = 1	**A**	s	n	a	r					
		A	s	n	a	r				

J = 2	**A**	**s**	n	a	r					
		A	s	n	a	r				
			A	s	n	a	r			

J = 3	**A**	**s**	**n**	a	r					
		A	s	n	a	r				
			A	s	n	a	r			
				A	s	n	a	r		

J = 4	**A**	**s**	**n**	**a**	r					
		A	s	n	a	r				
			A	s	n	a	r			
				A	s	n	a	r		

J = 5	**A**	**s**	**n**	**a**	**r**					
		A	s	n	a	r				
			A	s	n	a	r			
				A	s	n	a	r		
					A	s	n	a	r	
						A	s	n	a	r

De esta forma para el patrón “Asnar”, dependiendo en qué letra se produzca la diferencia, deberá desplazar el patrón de la siguiente forma:

j	Desplazamiento
1	0
2	0
3	0
4	3
5	0

Figura 153: “Tabla de Desplazamientos Izquierdos para Asnar”

Veamos como queda la tabla si el patrón fuese “Aznar”

J = 1	**A**	z	n	a	r					
		A	z	n	a	r				

J = 2	**A**	**z**	n	a	r					
		A	z	n	a	r				
			A	z	n	a	r			

J = 3	**A**	**z**	**n**	a	r					
		A	z	n	a	r				
			A	z	n	a	r			
				A	z	n	a	r		

J = 4	**A**	**z**	**n**	**a**	r					
		A	z	n	a	r				
			A	z	n	a	r			
				A	z	n	a	r		

J = 5	**A**	**z**	**n**	**a**	**r**					
		A	z	n	a	r				
			A	z	n	a	r			
				A	z	n	a	r		
					A	z	n	a	r	
						A	z	n	a	r

Casualmente para al patrón Aznar resultó la misma tabla que para Asnar, vea ahora como resultaría para Atnar:

J = 1	**A**	t	n	a	r					
		A	t	n	a	r				

J = 2	**A**	**t**	n	a	r					
		A	t	n	a	r				
			A	t	n	a	r			

J = 3	**A**	**t**	**n**	a	r					
		A	t	n	a	r				
			A	t	n	a	r			
				A	t	n	a	r		

J = 4	**A**	**t**	**n**	**a**	r					
		A	t	n	a	r				
			A	t	n	a	r			
				A	t	n	a	r		

J = 5	**A**	**t**	**n**	**a**	**r**					
		A	t	n	a	r				
			A	t	n	a	r			
				A	t	n	a	r		
					A	t	n	a	r	
						A	t	n	a	r

Nuevamente, y por casualidad, Atnar retorno la misma tabla. Note que tanto ‘Aznar’ como ‘Atnar’ dieron como resultado la misma tabla que ‘Aznar’, debido a que la ‘z’, la ‘s’ y la ‘t’ no se repiten nuevamente en el patrón, de hacerlo seguramente las tablas serían diferentes entre ellas.

Vea ahora cómo queda, teniendo en cuenta estas tablas, el Automáta Finito Determinista Reconocedor:

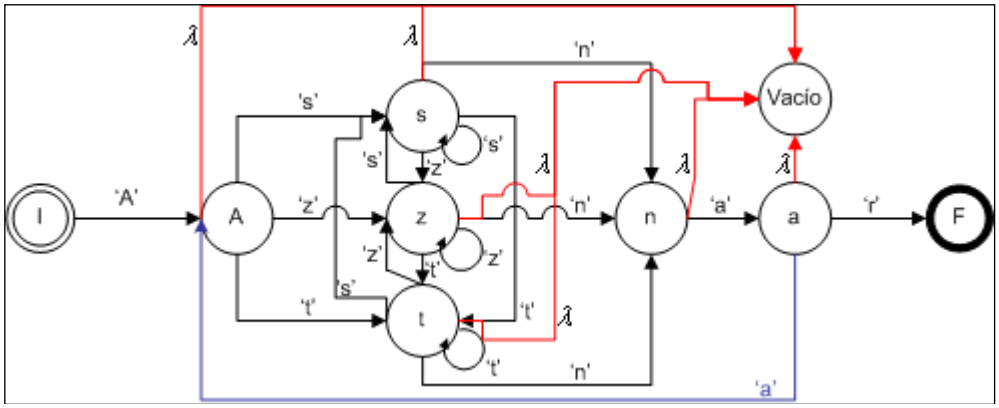


Figura 154: “AFD para la Expresión Regular A(s+z+t)*nar”

Se parte desde ‘I’, el estado inicial, en caso de reconocer una ‘A’, se transita al estado ‘A’, desde allí puede que llege una ‘s’, una ‘z’ o una ‘t’, de llegar, se transitará al estado asociado correspondiente. Si en el estado s, donde se sabe se reconoció mínimamente ‘As’, llega una nueva ‘s’ me mantengo en el mismo, si llega una ‘z’, transito al estado asociado, si llega una ‘t’ transito al estado de ‘t’. Si se encuentra en ‘s’, ‘z’ o ‘t’ e ingresa ‘n’, note que transita al estado ‘n’, lo que significa que mínimamente se tiene ‘Asn’ ó ‘Azn’ ó ‘Atn’, desde allí ya no se puede volver hacia atrás, por lo que si ingresase una ‘s’, ‘z’ o ‘t’ en este estado se volvería automáticamente a ‘I’, el estado inicial del reconocedor. Desde ‘n’, si ingresa una ‘a’, se transita a ‘a’, donde se espera una ‘r’ para poder transitar al estado final y reconocer la cadena.

Algunas cuestiones importantes a tener en cuenta del AFD de la figura anterior:

- No se colocaron todas las entradas posibles del alfabeto, para mantener la claridad del gráfico, pero si ingresase alguna letra o símbolo distinto de los enunciados en la anterior figura, producirían que se transite al estado inicial.
- Note que si usted se encuentra en el estado ‘a’ e ingresa una ‘a’, se produce un desplazamiento al estado ‘A’, esto es debido a las tablas de corrimiento anteriormente calculadas. Recuerde que si la diferencia se producía en el cuarto carácter de izquierda a derecha, j = 4, se desplazaba el patrón en tres unidades, en AFD la forma de desplazar es volviendo al estado asociado a la letra del patrón ‘A’. Esto último permite continuar con el análisis del texto, asumiendo que ahora el patrón se encuentra en el estado ‘A’, por ende las entradas que producirían un movimiento válido serían: ‘s’, ‘z’ y ‘t’.
- En caso de que la entrada en cualquiera de los estados sea λ, se transita automáticamente al Estado ‘Vacío’, indicando que finalizó la cadena sin reconocer el patrón.

De acuerdo a la definición de AFD Reconocedor, se define formalmente como sigue:

$\mathbf{T_e}$ = {a, b, c, d, e, f, g, h, i, j, k, l, m, n, ñ, o, p, q, r, s, t, u, v, w, x, y, z, λ }
Q = {'A', 'a', 's', 'z', 't', 'n', 'F', 'Vacío', 'I'}
$\mathbf{q_o}$ = {I}
$\mathbf{q_i}$ = {'F', 'Vacío'}

f	**Entradas**							
Estados	a	z	s	T	n	**r**	λ	otro
I	A	I	I	I	I	I	Vacío	I
A	I	z	s	T	I	I	Vacío	I
s	I	z	s	T	n	I	Vacío	I
z	I	z	s	T	n	I	Vacío	I
t	I	z	s	T	n	I	Vacio	I
a	A	I	I	I	I	**F**	Vacio	I
F	--	--	--	--	--	--	--	--
Vacío	--	--	--	--	--	--	--	--

Figura 154: "AFD para la Expresión Regular A(s+z+t)*nar"

Tome como ejemplo el estado y entrada en negrita de la función 'f' anterior, la tabla se lee como sigue: *"Si se encuentra actualmente en el estado '**a**' e ingresa una '**r**', transite al estado '**F**', como éste último se reconoce como estado final, se detiene la máquina".*

De esta forma, el Automata Finito Determinista Reconocedor, queda definido para la expresión regular 'A(s+z+t)*nar' como:

$$\mathbf{AF = (T_e, Q, f, q_o, F)}$$

Referencias

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995.

- "Analysis of Boyer-Moore-type String Searching Algorithms"; Ricardo Baeza-Yates, Gaston Gonnet y Mireille Régnier; Simposio Anual ACM-SIAM de Algoritmos Discretos; Enero de 1990.

- "Computation of the Knuth-Morris-Pratt Skip Tables"; Andy Huang; ACM SIGACT News, Volumen 29, Edición 3; Septiembre de 1998.

- "Apunte de Cátedra: Sintaxis y Semántica del Lenguaje", Tomo 1; Autor: Cátedra de Sintaxis y Semántica del Lenguaje, Universidad Tecnológica Nacional, Facultad Córdoba; 1998.

8 Compresión de Datos

Introducción a la Compresión de Datos

Si bien la potencia de los microprocesadores y los anchos de banda día a día son incrementados gracias a los avances de la tecnología, no es un motivo para desaprobecharlos, sino por el contrario, para optimizarlos.

La compresión surge como una disciplina para el aprovechamiento del espacio que ocupan los datos, sea en memoria o bien en disco. En la actualidad se disponen de discos de gigabytes de almacenamiento y, no sería de extrañar que un usuario hogareño en no más de 3 años disponga de discos de Terabytes lo que, en el año 1992 era impensado, ya que tenga en cuenta que la PC más agil era una PC AT 286 con 140MB de disco rígido.

Es cierto que al incorporar tareas de compresión, necesariamente se emplearán algoritmos que incrementarán el tiempo de procesamiento de una transacción o del mecanismo de acceso a memoria, como en el caso de la memoria virtual (en sistemas Unix/Linux se la conoce como memoria SWAP), pero esta carga de trabajo adicional por otro lado permite hacer un mejor aprovechamiento del espacio físico.

Naturalmente y, teniendo en cuenta la tecnología actual, una persona podría pensar que investigar aún más sobre compresión sería algo poco productivo, pero en absoluto es así, tenga en cuenta que aún existen dispositivos muy limitados en espacio que son muy utilizados en numerosos campos, como ser los Celulares y PDAs, que necesitan de la compresión para navegar por internet, enviar mensajes de texto hasta controlar un motor de bases de datos corporativo desde su PDA.

En el presente capítulo se abordarán dos mecanismos de compresión básicos, el primero consiste en la compresión por longitud de series, mientras que el segundo consiste en la codificación de longitud variable, con lo que se concluirá el capítulo con el método de Huffman.

Compresión por Codificación de Longitud de Series

La Compresión por codificación por longitud de series, consiste en encontrar una repetición sucesiva de un mismo símbolo y reemplazarlo en la cadena comprimida por la cantidad de ocurrencias y el símbolo, en este orden. Esta técnica es relativamente sencilla de implementar, sponga que tiene la siguiente cadena de caracteres:

A	A	A	A	B	B	B	B	C	C	C	J	C	C	J	C

Figura 155: “Cadena de Carcateres a Comprimir”

El método trabaja como sigue:

- Detecto el primer carácter,
- Veo si se repite, de ser así cuento las repeticiones,
- Contadas las repeticiones, en la cadena comprimida solo agrego el número que indica la cantidad de ocurrencias y el símbolo respectivo, en dicho orden,
- Agregado la subcadena comprimida, conitúo con el carácter a su derecha.

Vea como quedaría la cadena comprimida:

4	A	4	B	3	C	J	2	C	J	C

Figura 156: "Cadena de Carcateres Comprimida"

Vea que como el símbolo 'A' se repetía cuatro veces en la cadena original (Ver la cadena a comprimir), en la cadena comprimida se coloca "4A" lo que ahorra dos caracteres, que en términos de almacenamiento representan 2 bytes. De la misma forma que el carácter 'A', se trabajará con las restantes letras o símbolos de la cadena, pero note que para el caso en que no existe repetición, simplemente se incorpora el símbolo en la cadena comprimida, como por ejemplo 'J', esto último es debido a que si colocamos la cantidad de ocurrencias y el símbolo, en realidad estaríamos usando más espacio en lugar de comprimir, porque tendríamos la siguiente situación:

…	J	…

Cadena Original

…	1	J	…

Cadena Comprimida
(Supuesto de que Incorporemos el 1 aunque no exista repetición)

Salta a la vista que en dicha situación, nos conviene simplemente copiar el símbolo y omitir las ocurrencias del mismo, debido a que en realidad es '1' y no existen repeticiones asociadas.-

El proceso de descompresión, consiste en el proceso reverso sobre la cadena comprimida, deberá leer los caracteres numéricos existentes delante de la cadena, los que darán la pauta de cuantas veces debe de repetirse el símbolo a la derecha del número, concluido el primer símbolo se continúa con el resto de la cadena comprimida, así si tenemos:

4	A	4	B	3	C	J	2	C	J	C

Figura 157: "Cadena de Carcateres Comprimida"

Implica que la 'A' debe de repetirse cuatro veces en la cadena expandida, luego la 'B' se repite también cuatro veces y así sucesivamente hasta el final de la cadena. En el caso anterior, se volvería a obtener la cadena original sin comprimir.

Este algoritmo tal vez no le vea mucha utilidad aplicado a cadenas de caracteres, pero es muy útil aplicado a nivel de bits, donde la única simbología son '0' y '1' y, es frecuente la repetición entre ellos.

Implementación del Algoritmo de Compresión por Longitud de Series

A continuación se expone el programa "comp_ls.cpp" que implementa los métodos de compresión/descompresión mediante longitud de series de una cadena de caracteres estática:

Archivo: **"comp_ls.cpp"**

```
1       #include <iostream.h>
2       #include <stdlib.h>
3       #include <string.h>
4       #include <conio.h>
5       #include <ctype.h>
6
7       struct nodo{
8               char data;
9               nodo* next;
10              nodo(char caracter)
11              {
12               data=caracter;
13               next=NULL;
14              }
15             };
16
17      char* comprimir(char*);
18      char* descomprimir(char*);
19
20      //Funciones de Lista de Soporte a la Descompresion
21      int contarNodos(nodo*);
22      void insertLast(nodo*&,char);
23      void deleteAll(nodo*&);
24
25      void main(void)
26      {
27       clrscr();
28       char cadena[]="AAAAAABBBBBBBBCCCCCCEFFFFFFFFJAAAAAAAAGGGGGGGGGG";
29       cout<<"Cadena a Comprimir:\n\t";
30       cout<<cadena<<"\n\n";
31       cout<<"Cadena Comprimida:\n\t";
32       char* cadcomp=comprimir(cadena);
33       cout<<cadcomp<<"\n\n";
34       char* descomp=descomprimir(cadcomp);
35       cout<<"Cadena Descomprimida:\n\t";
36       cout<<descomp;
37
38       delete cadcomp;
39       delete descomp;
```

```
40      cadcomp=NULL;
41      descomp=NULL;
42     }
43
44     char* comprimir(char* acomp)
45     {
46      int longitud=strlen(acomp);
47      char *cadcomp=new char[longitud+1];
48      cadcomp[0]='\0';
49
50      char dummy[2];//debido a que strcat espera puntero a cadena
51
52      if(cadcomp==NULL)
53      {
54       cout<<"\nNo Hay Memoria para Comprimir la Cadena";
55       return NULL;
56      }
57
58      int repite=1;//al menos existe una letra
59      int j=0;//ptr posicion comprimida
60      for(int i=1;i<longitud;i++)
61      {
62       if(acomp[i]==acomp[i-1])
63       {
64        repite++;
65       }
66       else
67       {
68        if(repite>1)
69        {//existieron varias letras continuas
70         char rep[5];
71         gcvt((double)repite,2,rep);//convierto a cadena
72
73         for(int z=0;z<strlen(rep) && j<longitud;z++,j++)
74             cadcomp[j]=rep[z];
75
76         cadcomp[j]=acomp[i-1];
77         j++;
78
79         repite=1;//reinicio cuenta
80        }
81        else
82        {//solo es una letra
83         cadcomp[j]=acomp[i-1];
84         j++;
85        }
86       }
87      }//fin del for
88
89      //Al Salir puede que:
```

```
90        if(repite>1)
91        {//falte por agregar la ultima secuencia
92         char rep[5];
93         gcvt((double)repite,2,rep);//convierto a cadena
94         for(int z=0;z<strlen(rep) && j<longitud;z++,j++)
95              cadcomp[j]=rep[z];
96
97         cadcomp[j]=acomp[i-1];
98         j++;
99
100        repite=1;//reinicio cuenta
101       }
102       else
103       {//solo es una letra
104        cadcomp[j]=acomp[i-1];
105        j++;
106       }
107
108     cadcomp[j]='\0';
109     return cadcomp;
110    }
111
112    char* descomprimir(char* cadcomp)
113    {
114     nodo* raiz=NULL;
115     char numero[5];
116     int flagnro=0;
117
118     for(int i=0;i<=strlen(cadcomp);i++)
119     {
120      switch(cadcomp[i])
121      {
122       case '0':
123       case '1':
124       case '2':
125       case '3':
126       case '4':
127       case '5':
128       case '6':
129       case '7':
130       case '8':
131       case '9':
132              numero[flagnro]=cadcomp[i];
133              flagnro++;
134              break;
135       default://no es numero
136           if(flagnro==0)
137            {//es solo un simbolo
138              insertLast(raiz,cadcomp[i]);
```

```
139            }
140           else
141            {
142             numero[flagnro]='\0';//cierro la carga de numeros
143             int nro=atoi(numero);
144             for(int k=0;k<nro;k++) insertLast(raiz,cadcomp[i]);
145             flagnro=0;//reinicio cadena
146            }
147      }// fin del switch
148     }//fin del for
149
150     int tam=contarNodos(raiz);//obtengo el tamanio de la cadena expandida
151     char* expand=new char[tam+1];//agrego un caracter para la cadena en
        blanco
152
153     nodo* p=raiz;
154     int pos=0;
155
156     while(p!=NULL)
157     {
158      expand[pos]=p->data;
159      pos++;
160      p=p->next;
161     }
162
163     expand[pos]='\0';//fin cadena expandida
164     deleteAll(raiz);
165     return expand;
166    }
167
168    int contarNodos(nodo* raiz)
169    {
170     nodo* p=raiz;
171     int contador=0;
172
173     while(p!=NULL)
174     {
175       contador++;
176       p=p->next;
177     }
178
179     return contador;
180    }
181
182    void insertLast(nodo*& raiz,char data)
183    {
184     if(raiz==NULL)
185      {
186       raiz=new nodo(data);
187       return;
```

```
188       }
189
190      nodo* aux=NULL;
191      nodo* p=raiz;
192      while(p!=NULL)
193      {
194       aux=p;
195       p=p->next;
196      }
197
198      nodo* last=new nodo(data);
199      aux->next=last;
200     }
201
202     void deleteAll(nodo*& raiz)
203     {
204      nodo* aux=NULL;
205      nodo*p=raiz;
206
207      while(p!=NULL)
208      {
209        aux=p;
210        p=p->next;
211        delete aux;
212        aux=NULL;
213      }
214
215      raiz=NULL;
216     }
```

Compresión de Longitud Variable

Como opción a la compresión por longitud de serie, se encuentra el algoritmo de compresión de longitud variable, éste mecanismo de compresión permite ganar cantidad de espacio a nivel de bits debido a que consiste en recodificar la representación de un símbolo desde los de mayor ocurrencia a los de menor ocurrencia, con símbolos que contienen la combinación más pequeña hasta los de combinación más extensa respectivamente.

Dado una cadena o archivo a comprimir, uno podría intentar definir alguna forma de asignar una combinación de unos y ceros que represente cada uno de los símbolos que contiene.

Suponga que se tiene la cadena:

ABAABCCD

En forma arbitraria se decide asignar a:

- A la combinación 1,
- B la combinación 0,

- C la combinación 10,
- D la combinación 11,

Si Comprimimos utilizando esta combinación de series variables, la cadena original quedaría como:

10110101011

Aparentemente muy eficiente, ya que para almacenar una 'A' empleaba 8 bit mientras que ahora solo 1 bit y, de acuerdo a esto la cadena comprimida posee 11 bits mientras que la cadena original posee 8 símbolos*8 bits/símbolo= 64 bits.

Pero el problema aparecerá al intentar recuperar el texto original del texto comprimido, debido a que como no se respeto una norma clara de asignación de códigos, ahora no se puede diferenciar entre un símbolo y otro, vea:

1	0	1	1	0	1	0	1	0	1	1

Figura 158: "Cadena de Bits Comprimida (Caso A)"

¿Será una A?

1	0	1	1	0	1	0	1	0	1	1

Figura 159: "Cadena de Bits Comprimida (Caso B)"

¿Será una C?

Se podría entonces establecer una regla de asignación como ser: "todas las combinaciones de bits para representar un símbolo deben terminar en cero", entonces se podría disponer de:

A	10
B	110
C	1110
D	11110

Figura 160: "Tabla de Codificación para Longitud Variable de Series"

Ahora la cadena original:

ABAABCCD

Quedaría como:

1011010101101110111011110

Note que si bien se emplearon ahora 27 bits en lugar de los 11 bits vistos con anterioridad, esta codificación permite la reversibilidad, en otras palabras, la descompreción, vea:

1	**0**	1	1	**0**	1	0	1	0	1	1	0	1	1	1	0	1	1	1	0	…
0	**1**	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…

Figura 161: "Cadena de Bits según Tabla de Codificación"

Vea que ahora sí se puede diferenciar la 'A', ya que al encontrar el cero, deberemos contar la cantidad de '1' entre el cero encontrado y el inicio o el último cero encontrado y, a partir de ello se determina el código asociado al símbolo. En el caso de la 'B', el cero se encontro en '4', el cero anterior se encuentra en la posición '1', con lo que tenemos dos '1' entre ellos, lo que a su vez implica que deberemos buscar el símbolo para el código '110'.

Es claro que el método continúa siendo eficaz, ya que para una cadena que naturalmente necesitaría 64 bits para ser representada, tan solo se necesitan 27 bits mediante este mecanismo de longitud variable de series.

Algoritmo de Huffman

Este último problema de longitud variable de series, fue abordado por el Dr. Huffman en el año 1952, quien descubrió un mecanismo eficiente de asignación de códigos a símbolos que optimiza el mecanismo de asignación de códigos a los símbolos. Básicamente el Algoritmo se diferencia en tres fases o etapas:

1. **Construcción de una Tabla de Frecuencias:** esta etapa se inspeccionan todas las ocurrencias posibles de símbolos dentro del documento a comprimir, asociando a cada uno de ellos la frecuencia con la que aparecen en el mismo.
2. **Construcción del Árbol de Huffman:** A partir de la tabla de frecuencias, se construye un árbol binario cuyas hojas serán los símbolos del documento y el la cantidad máxima de nodos del árbol será n+(n-1).
3. **Compresión propiamente Dicha:** A partir del árbol de huffman, se determinará para cada símbolo su correspondiente combinación de bits, ésta se coloca en el archivo comprimido en lugar del símbolo y, se continúa con los siguientes símbolos hasta que no existan más en la cadena orignial.

A continuación, se analizarán las tres fases con el ejemplo concreto de la cadena:

ABAABCCD

En la Primera etapa, "Contrucción de una Tabla de Frecuencias", se determina los símbolos del documento y su frecuencia en el mismo, el resultado será:

Símbolo	Frecuencia
A	3
B	2
C	2
D	1

Figura 162: "Tabla de Frecuencias"

La construcción de la tabla es relativamente sencilla desde el punto de vista algorítmico, solo se debe ir símbolo por símbolo analizando la repetición del mismo en el documento y almacenar la combinación (símbolo; frecuencia) en la tabla asociada, tal y como se muestra en la figura anterior.

La Segunda etapa consiste en la construcción de un árbol binario a partir de la tabla de frecuencias, el resultado final es:

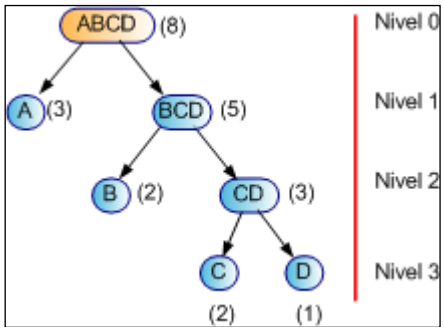


Figura 163: "Árbol de Huffman"

Note que a medida que se sube en el árbol binario, se van encontrando en las hojas los símbolos con mayor ocurrencia en el documento y ello no es casualidad, se pondrá de manifiesto el porque en la construcción de la tabla de códigos.

Para construir el árbol binario, se toman los dos símbolos con menor frecuencia, el hijo derecho ('D') será el de menor frecuencia de los dos, mientras que el hijo izquierdo ('C') el de mayor frecuencia, el padre de éstos será la combinación de los mismos, cuyo nombre es la concatenación del nombre del hijo izquierdo con el del derecho en ese orden ('CD') y, la frecuencia del padre se determina a partir de la suma de las frecuencias de sus hijos (3=1+2). Construido el padre, los dos símbolos empleados se borran de la tabla de frecuencia, con lo que queda:

Símbolo	Frecuencia
A	3
B	2

Figura 163: "Tabla de Frecuencias, luego de crear 'CD'"

En el nivel superior que sigue, se toma el padre construido ('CD') como hijo derecho y, el símbolo de menor frecuencia en la nueva tabla ('B'), como hijo izquierdo as´, se contruye un nuevo padre denominado 'BCD' con una frecuencia de 5=3+2. Construido 'BCD', se elimina la entrada del símbolo 'B' de la tabla de frecuencias quedando:

Símbolo	Frecuencia
A	3

Figura 164: "Tabla de Frecuencias, luego de crear 'BCD'"

Se toma al reciente padre, 'BCD', como hijo derecho y al elemento de menor frecuencia de la tabla, como hijo izquierdo, así se contruye el nuevo padre denominado 'ACBD' con una frecuencia de 8=3+5. Construido el padre 'ACBD', se elimina 'A' de la tabla de frecuencia quedando:

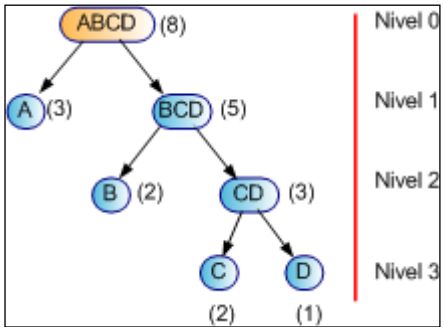
Símbolo	Frecuencia

Figura 165: "Tabla de Frecuencias, luego de crear 'ABCD'"

Como en la tabla de frecuencia no existen más elementos, finaliza la construcción del árbol de huffman, note algunos aspectos interesantes:

- El árbol de huffman emplea N+(N-1) nodos, donde N es la cantidad de símbolos distintos en un documento y se necesitan (N-1) nodos para los padres que se costruyen a partir de éstos,
- Todos los símbolos se encuentran ubicados en las hojas del árbol,
- La Hoja Extrema Derecha es la única que representa un símbolo, ya que las restantes hojas con símbolos son izquierdas, pero no es cualquier símbolo, es el de menor frecuencia en el documento,
- Los Símbolos con mayor ocurrencia en el documento se encuentran cercanos a la raíz,
- La Frecuencia de la raíz debe coincidir con la cantidad de símbolos analizados en el documento, contando repetidos, de lo contrario se salteo alguno. Es un mecanismo de control útil.

Los Códigos de los símbolos se construyen navegando desde la hoja con el símbolo hasta la raíz, de esta forma si se sube por la izquierda se agrega un cero '0' de derecha a izquierda, mientras que si se sube por derecha, se agrega un '1' de derecha izquierda; se finaliza al llegar a la raiz. Note algo muy importante, el llenado de símbolos es de derecha a izquierda, con lo que a partir de árbol de huffman se obtiene:



Símbolo	Código
A	0
B	10
C	110
D	111

Figura 166: "Tabla de Códigos de Huffman"

La 'A' subiendo un nivel (por izquierda) llega a la raíz, esto implica que su código es '0'. La 'B' debe subir una vez por izquierda y una vez por derecha para llegar a la raíz, lo que da el código '10'. La 'C' debe subir una vez por izquierda y dos veces por derecha para llegar a la raíz, lo que devuelve el código '110'. Finalmente, 'D' debe subir tres veces por derecha lo que devuelve '111'.

De esta forma los símbolos con mayor frecuencia son los que disponen del código de menor longitud, y allí está la justificación de porqué se los ubica cercanos a la raíz, mientras que por el contrario los símbolos con menor frecuencia, son los más alejados de la raíz están y, los que tienen una longitud de código más extensa, pero justamente al tener una baja frecuencia su impacto no es tan significativo en el resultado final como lo sería codificar con un código extenso a la 'A'.

La Tercera etapa, comprime el mensaje utilizando estos códigos. Partiendo del mensaje original:

ABAABCCD

Se obtiene la cadena comprimida:

0	1	0	0	0	1	0	1	1	0	1	1	0	1	1	1

Figura 167: "ABAABCCD comprimida según Huffman"

Vea que todos los finales de códigos se representan con un '0', salvo el del símbolo de menor frecuencia que es todos 1 ¿Se podrá identificar? Sí, a través de la longitud máxima de código empleada para generarlos, que no es más que el nivel del árbol binario '3'. En otras palabras, para identificar un símbolo busco:

- Un Cero que indique el fin de serie, siempre que lo haga antes de excederme del tercer carácter,
- Si no se encuentra un cero en tres carateres analizados, se sabeque es el de menor frecuencia 'D', y se reincia la búsqueda en el siguiente carácter.

Entonces, el proceso de descompresión no es más que localizar el código en la tabla y transcribir el símbolo en el archivo o documento "descomprimido", vea:

0	**1**	**0**	0	**0**	1	0	**1**	**1**	**0**	1	1	0	**1**	**1**	**1**
A		B	A	A		B			C			C			D

Figura 168: "ABAABCCD descomprimida según Huffman"

Vea que la 'D' se identificó debido a que se encontraron tres unos consecutivos, sin cero entre medio. Este valor 'tres' surge del nivel del árbol binario, que coincide con la longitud máxima de la serie.

Implementación del Algoritmo de Huffman

A continuación, se expone el código del Algoritmo de Huffman en C/C++, preste especial atención a la construcción del arbol binario donde, teniendo en cuenta que el árbol binario de huffman es completo, se puede utilizar un arreglo para su implementación sin mayores inconvenientes (Remítase al Apartado Árboles Binarios para mayor detalle):

Archivo: "comp_huf.cpp"

```
1       #include <iostream.h>
2       #include <stdlib.h>
3       #include <string.h>
4       #include <conio.h>
5       #include <ctype.h>
6
7       struct nodo{
8               char data;
9               int frecuencia;
10              nodo* next;
```

```
11
12            nodo(char caracter,int frec)
13            {
14             frecuencia=frec;
15             data=caracter;
16             next=NULL;
17            }
18           };
19
20     enum bool{FALSE,TRUE};
21
22     struct nodocodigo{
23            char nombre[30];
24            bool isIzquierdo;
25            char data;
26            int frecuencia;
27           };
28
29     char* comprimir(nodocodigo*,int,char*);
30     char* descomprimir(nodocodigo*,int,char*);
31
32     //Funciones de Lista de Soporte a la Compresion/Descompresion
33     int contarNodos(nodo*);
34     int buscar(nodo*,char);
35     void insertOrdenado(nodo*&,char,int);
36     void insertLast(nodo*&,char);
37     void deleteAll(nodo*&);
38     void creaTablaFrecuencia(nodo*&, char*);
39     void mostrarFrecuencias(nodo*);
40     void borrarPrimero(nodo*&);
41     void arbolHuffman(nodocodigo*&,nodo*&,int&);
42
43     //Funciones auxiliares
44     char* codificar(nodocodigo*,char,int);
45     int maxLongitudSerie(nodocodigo*,int);
46
47     void main(void)
48     {
49      nodocodigo* tabla=NULL;
50      nodo *raiz=NULL;
51      int totalnodos=0;
52
53      clrscr();
54      char cadena[]="AAAAAABBBBBBBBBCCCCCCEFFFFFFFFJAAAAAAAAGGGGGGGGGG";
55      cout<<"Cadena a Comprimir:\n\t";
56      cout<<cadena<<"\n";
57      creaTablaFrecuencia(raiz,cadena);
58      arbolHuffman(tabla,raiz,totalnodos);//construyo el arbol binario
59
```

```
60      cout<<"Arbol de Huffman (Max.Long.:
       "<<maxLongitudSerie(tabla,totalnodos)<<")\n";
61      for(int i=0;i<totalnodos;i++)
62      {
63
       cout<<tabla[i].nombre<<"\t"<<tabla[i].frecuencia<<"\t"<<codificar(tabla,t
       abla[i].data,totalnodos)<<"\n";
64      }
65
66      cout<<"\nCadena Comprimida:\n";
67      char* comprimida=comprimir(tabla,totalnodos,cadena);
68      cout<<comprimida;
69      cout<<"\nCadena Descomprimida:\n";
70      char* descomprimida=descomprimir(tabla,totalnodos,comprimida);
71      cout<<"\t"<<descomprimida;
72
73
74      delete comprimida;
75      delete descomprimida;
76      comprimida=NULL;
77      descomprimida=NULL;
78     }
79
80     char* comprimir(nodocodigo* tabla, int totalnodos,char* orig)
81     {
82      int maxlong=maxLongitudSerie(tabla,totalnodos);
83      char* acomp=new char[strlen(orig)*maxlong];
84      acomp[0]='\0';
85
86      for(int i=0;i<strlen(orig);i++)
87      {
88       strcat(acomp,codificar(tabla,orig[i],totalnodos));
89      }
90      cout<<"Comprimida: "<<strlen(acomp)<<" bits vs. "<<(strlen(orig)*8)<<"
       bits\n";
91      return acomp;
92     }
93
94     char* descomprimir(nodocodigo* tabla, int totalnodos,char* comp)
95     {
96      int maxlong=maxLongitudSerie(tabla,totalnodos);
97
98      nodo* descomp=NULL;
99      int longTmp=1;
100     int poscodigo=0;
101
102     for(int i=0;i<strlen(comp);i++)
103     {
104      if(comp[i]=='1')
105       {
```

```
106         poscodigo+=2;//hijo derecho
107
108         if(longTmp==maxlong)//un codigo no tiene mas de maxlong
109           {
110            insertLast(descomp,tabla[poscodigo].data);
111            poscodigo=0;
112            longTmp=1;
113           }
114          else
115           longTmp++;
116       }
117      else
118       {//0 termina un simbolo
119          poscodigo++;//hijo izquierdo
120          insertLast(descomp,tabla[poscodigo].data);
121          poscodigo=0;
122          longTmp=1;
123       }
124     }
125
126     int qnodos=contarNodos(descomp);
127     char* des=new char[qnodos+1];
128     nodo* p=descomp;
129     i=0;
130     while(p!=NULL)
131      {
132       des[i]=p->data;
133       i++;
134       p=p->next;
135      }
136     des[i]='\0';
137
138     deleteAll(descomp);
139     return des;
140    }
141
142    int contarNodos(nodo* raiz)
143    {
144     nodo* p=raiz;
145     int contador=0;
146
147     while(p!=NULL)
148     {
149       contador++;
150       p=p->next;
151     }
152
153     return contador;
154    }
```

```
155
156    int buscar(nodo* raiz,char data)
157    {
158     if(raiz==NULL) return 0;
159
160     nodo* p=raiz;
161     while(p!=NULL && p->data!=data) p=p->next;
162
163     if(p==NULL) return 0;//no existe
164
165     return 1;//existe
166    }
167
168    void insertOrdenado(nodo*& raiz,char data,int frec)
169    {
170     if(buscar(raiz,data))
171           return;//Retornar, pre existe
172
173     nodo* p=raiz;
174     nodo* q=NULL;
175     nodo* s=NULL;
176
177     while(p!=NULL && frec>p->frecuencia)
178       {
179        q=p;
180        p=p->next;
181       }
182
183      s=new nodo(data,frec);
184      if(s==NULL)
185        {
186         cout<<"Falta de Memoria";
187         return;
188        }
189
190      if(q==NULL)
191       {
192         if(raiz==NULL) raiz=s;
193          else
194           {
195            s->next=raiz;
196            raiz=s;
197           }
198       }
199       else
200       {
201        s->next=q->next;
202        q->next=s;
203       }
204    }
```

```
205
206     void deleteAll(nodo*& raiz)
207     {
208      nodo* aux=NULL;
209      nodo*p=raiz;
210
211      while(p!=NULL)
212      {
213        aux=p;
214        p=p->next;
215        delete aux;
216        aux=NULL;
217      }
218
219      raiz=NULL;
220     }
221
222     void creaTablaFrecuencia(nodo*& raiz, char* acomp)
223     {
224      int frecuencia=0;
225      for(int i=0;i<strlen(acomp);i++)
226      {
227        for(int j=0;j<strlen(acomp);j++)
228            frecuencia=(acomp[i]==acomp[j])?++frecuencia:frecuencia;
229
230        insertOrdenado(raiz,acomp[i],frecuencia);
231        frecuencia=0;
232      }
233     }
234
235     void mostrarFrecuencias(nodo* raiz)
236     {
237       cout<<"\nTabla de Frecuencias:\n\n";
238       nodo* p=raiz;
239
240       if(p==NULL) cout<<"Raiz Nula";
241
242       while(p!=NULL)
243       {
244        cout<<p->data<<"\t"<<p->frecuencia<<"\n";
245
246        p=p->next;
247       }
248     }
249
250     void borrarPrimero(nodo*&raiz)
251     {
252      nodo* p=raiz;
253      raiz=raiz->next;
```

```
254     delete p;
255     p=NULL;
256    }
257
258    void arbolHuffman(nodocodigo*& arbol,nodo*& listaFrecuencias,int
       &totalenarbol)
259    {
260     int nronodos=contarNodos(listaFrecuencias);
261     totalenarbol=nronodos+(nronodos-1);
262     arbol=new nodocodigo[totalenarbol];
263
264     nodo* p1=listaFrecuencias;
265     nodo* p2=listaFrecuencias->next;
266
267     //Hoja Extrema Derecha del Arbol Huffman
268     arbol[totalenarbol-1].nombre[0]=p1->data;
269     arbol[totalenarbol-1].nombre[1]='\0';
270     arbol[totalenarbol-1].isIzquierdo=FALSE;
271     arbol[totalenarbol-1].data=p1->data;
272     arbol[totalenarbol-1].frecuencia=p1->frecuencia;
273
274     //Hoja Extrema Izquierda del Arbol Huffman
275     arbol[totalenarbol-2].nombre[0]=p2->data;
276     arbol[totalenarbol-2].nombre[1]='\0';
277     arbol[totalenarbol-2].isIzquierdo=TRUE;
278     arbol[totalenarbol-2].data=p2->data;
279     arbol[totalenarbol-2].frecuencia=p2->frecuencia;
280
281     //Primer Padre del Arbol Huffman
282     arbol[totalenarbol-3].nombre[0]=p2->data;
283     arbol[totalenarbol-3].nombre[1]=p1->data;
284     arbol[totalenarbol-3].nombre[2]='\0';
285     arbol[totalenarbol-3].isIzquierdo=FALSE;
286     arbol[totalenarbol-3].data='\0';
287     arbol[totalenarbol-3].frecuencia=p1->frecuencia+p2->frecuencia;
288
289     borrarPrimero(listaFrecuencias);//borra p1
290     borrarPrimero(listaFrecuencias);//borra p2
291
292     for(int i=(totalenarbol-3);i>=2;i-=2)
293     {
294       p1=listaFrecuencias;
295       arbol[i-1].nombre[0]=p1->data;
296       arbol[i-1].nombre[1]='\0';
297       arbol[i-1].isIzquierdo=TRUE;
298       arbol[i-1].data=p1->data;
299       arbol[i-1].frecuencia=p1->frecuencia;
300
301       arbol[i-2].nombre[0]=p1->data;
302       arbol[i-2].nombre[1]='\0';
```

```
303       strcat(arbol[i-2].nombre,arbol[i].nombre);//concateno los nombres
304       arbol[i-2].isIzquierdo=FALSE;
305       arbol[i-2].data='\0';
306       arbol[i-2].frecuencia=p1->frecuencia+arbol[i].frecuencia;
307
308       borrarPrimero(listaFrecuencias);
309     }
310    }
311
312    char* codificar(nodocodigo* tabla,char data,int totalnodos)
313    {
314     for(int i=0;i<totalnodos && tabla[i].data!=data;i++);
315
316     if(data=='\0') return "";
317
318     char ultimo='\0';
319     if(tabla[i].isIzquierdo==TRUE)
320        {
321         i--;//solo en uno decremento
322         ultimo='0';
323        }
324       else
325        {
326         i-=2;//decremento en dos para llegar al padre
327         ultimo='1';
328        }
329
330     int contador=0;
331     for(int j=i;j>=0;j-=2,(j>=0)?++contador:contador);
332
333     char* codigo=new char[contador+2];//un espacio para el ultimo caracter y
        el otro
334                                  //para el de fin de cadena
335
336     for(j=0;j<contador;j++)
337        codigo[j]='1';
338
339     codigo[j]=ultimo;
340     codigo[j+1]='\0';
341
342
343     return codigo;
344    }
345
346    int maxLongitudSerie(nodocodigo* tabla, int totalnodos)
347    {
348     int max=0;
349     for(int i=0;i<totalnodos;i++)
350     {
```

```
351      char* tmp=codificar(tabla,tabla[i].data,totalnodos);
352
353      max=(strlen(tmp)>max)?strlen(tmp):max;
354     }
355      return max;
356    }
357
358    void insertLast(nodo*& raiz,char data)
359    {
360     if(raiz==NULL)
361      {
362       raiz=new nodo(data,1);
363       return;
364      }
365
366     nodo* aux=NULL;
367     nodo* p=raiz;
368     while(p!=NULL)
369     {
370      aux=p;
371      p=p->next;
372     }
373
374     nodo* last=new nodo(data,1);
375     aux->next=last;
376    }
```

Referencias

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995.

- "Design and Analysis of Dynamic Huffman Codes"; Jeffrey Scout-Vitter; Journal of ACM, Vlumen 34, Edición 4; Octubre de 1987.

- "Introducción a los Sistemas de Bases de Datos", Volumen 1, Quinta Edición; Dr. C.J. Date; Editorial Addison Wesley Lonman; México 1993.

- "Data Compression"; Debra Lewer y Daniel Hirschberg; ACM Computing Surveys; Volumen 19, Edición 3; Septiembre de 1987.

9

Criptología

Introducción a la Criptología

Se podría decir que desde hace tiempo la gente desea proteger información de extraños que puedan perjudicarlos. Se han tenido en cuenta diversas herramientas para lograr esta última finalidad y, dentro de éstas se encuentra la Criptología. Se podría definir a la Criptología como la "Lógica de los Mecanismos de Ocultamiento Dirigido y Reversible de Información".

El presente apartado, pretende ser una introducción pragmática a los mecanismos de ocultamiento de información, a través del empleo de distintos algoritmos que nos permiten lograr dicho objeto.

Los Algoritmos Analizados en el presente capítulo son en su mayoría del tipo "Boletín de Mensajes", es decir que a partir de un mensaje a cifrar, se aplica la lógica de un algoritmo cifrador y se obtiene el correspondiente mensaje cifrado.

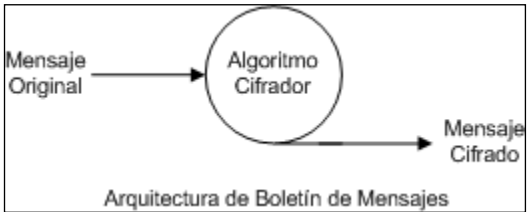


Figura 169: "Arquitectura del Boletín de Mensajes"

Existen otras arquitecturas de Cifrado como la de Clave Privada y Pública, en donde el mensaje a cifrar emplea una clave privada (solo la conoce el emisor) con un algoritmo cifrador y, solo podrán leer el mensaje, aquellas personas que dispongan de una clave pública (que entrega el emisor a aquellas personas que desea habilitar a que lean sus mensajes) y ejecuten el correspondiente algoritmo descifrador con ella.

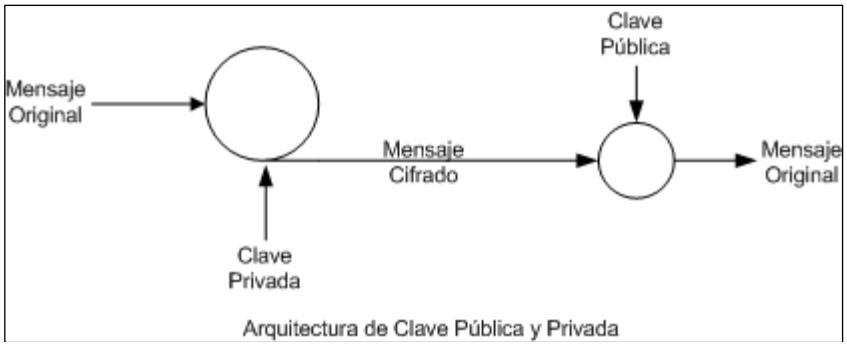


Figura 170: "Arquitectura de Clave Pública y Privada"

Una característica muy importante de esta arquitectura, es que la clave privada no puede ser obtenida a partir de la clave pública.

Hasta aquí es donde se llegará con la introducción teórica al tema de Criptología y, como se mencionó anteriormente, el capítulo tiene por intención introducirlo pragmáticamente desde algoritmos de cifrado simples, incrementando la complejidad gradualmente, hasta que obtenga un panorama general de la temática.

Algoritmo de Encriptación: "Cifra del César"

El Algoritmo de Encriptación del César, también conocido como "Cifra del César", se basa en algunas premisas a saber:

- Supone que los símbolos del mensaje poseen un orden definido e inmutable dentro de su alfabeto. Ej: Si tomamos el abecedario, sabemos que la primera letra es la 'a', la segunda es la 'b', la tercera es la 'c' y así sucesivamente hasta llegar a su última letra, la 'z'.
- Es factible aplicar alguna operación aditiva a cada uno de los símbolos del alfabeto, basándose en la premisa anterior. Ej: Si a 'a' se le suma 1, debiera tenerse la 'b', si a ésta última se le suma un 2, debiera tenerse la 'd'.

Así, el algoritmo original del César, suponía que dado el mensaje que se quería cifrarcomo una combinación sucesivade los símbolos de un alfabeto, el mensaje cifrado consistía e desplazar cada símbolo del alfabeto en un factor 'k' entero. Éste último factor 'k' originalmente era el entero 3, pero el desarrollador puede decidir una magnitud entera arbitraria sin que esto, perjudique la lógica del algoritmo.

Entonces, si tenemos el mensaje "Hola", con un k igual a 3, el mensaje encriptado debiera ser "KROD". Cae de maduro, que el método no es muy seguro hoy día, ya que un criptoanalista tan solo debiera identificar el valor correcto de 'K', para descifrar el mensaje.

El Programa a continuación, denominado "Ecesar.cpp", implementa el algoritmo "Cifra del César" en C/C++:

Archivo: **ecesar.cpp**

```
1       #include <iostream.h>
2       #include <string.h>
3       #include <conio.h>
4
5       const short int factorCesar=3;
6
7       char* encriptarCesar(char*);
8       char* descifrarCesar(char*);
9
10      void main(void)
11      {
12       char cadena[]="Hola Alumnos de Cs. de la Computaci¢n";
13       clrscr();
14       cout<<"Mensaje Original: "<<cadena;
15       char* enc=encriptarCesar(cadena);
16       cout<<"\nMensaje Cifrado: "<<enc;
17       char* desc=descifrarCesar(enc);
18       cout<<"\nMensaje Descifrado: "<<desc;
19       delete desc;
20       delete enc;
21       desc=NULL;
```

```
22      enc=NULL;
23     }
24
25     char* encriptarCesar(char* aenc)
26     {
27      char* enc=new char[strlen(aenc)+1];
28
29      for(int i=0;i<strlen(aenc);i++)
30           enc[i]=(char)((int)aenc[i]+factorCesar);
31
32      enc[i]='\0';
33      return enc;
34     }
35
36     char* descifrarCesar(char* aenc)
37     {
38      char* enc=new char[strlen(aenc)+1];
39
40      for(int i=0;i<strlen(aenc);i++)
41           enc[i]=(char)((int)aenc[i]-factorCesar);
42
43      enc[i]='\0';
44
45      return enc;
46     }
```

Algoritmo de Vigenere

El Algoritmo de la Cifra de Vigenere, es similar al de la cifra del césar, solo que en lugar de utilizar un factor ‘k’ constante, emplea una clave arbitraria,para definir un factor ‘k’ en un intervalo entero.

Este algoritmo, se basa en las siguientes premisas:

- Se defina una clave finita. Ej: “Corredor”, “Jugador”, etc.
- Cada letra **dentro** de la clave tiene una posición definida. Ej: Si tomamos el primer ejemplo de clave, “Corredor”, la ‘C’ es 1, la ‘o’ es 2, la primera ‘r’ es 3, la segunda ‘r’ es cuatro, y así sucesivamente hasta la última letra del mensaje, ‘r’ que es 8.
- A partir de un mensaje finito a cifrar, se emparejará y repetirá en serie la clave seleccionada, tantas veces como haga falta hasta igualar la longitud del mensaje.

A partir de estas premisas se tiene, por ejemplo, el siguiente mensaje:

H	O	L	A	L	A	U	R	A

Se escoge como clave ‘corredor’, por lo que al emparejar en serie con el mensaje se tiene:

H	O	L	A	L	A	U	R	A

C	O	R	R	E	D	O	R	**C**

Note que como el mensaje posee una longitud superior a la de la clave utilizada, debe repetirse el mensaje en la cantidad de símbolos que haga falta para emparejar completamente el mensaje, en este último caso fue necesario tan solo un símbolo, la 'C'.

La Cifra de Vigenere, se basa en desplazar en 'k' al símbolo del mensaje, pero donde la magnitud entera de 'k' es la posición relativa al inicio de la clave, para el símbolo de la clave que empareja con el símbolo del mensaje a cifrar. Por Ejemplo:

Mensaje Original	H	O	L	A	L	A	U	R	A
Clave	C	O	R	R	E	D	O	R	**C**
Posiciones Relativas de 'k'	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**1**
Mensaje Cifrado	**I**	**Q**	**O**	**E**	**Q**	**G**	**B**	**Z**	**B**

Vea que como la 'C', es el símbolo de la clave empareja con la 'H' del mensaje, y 'C' es la primera letra de la clave, el desplazamiento 'k' es 1, por lo que la 'H' se cifra como 'I'. En el segundo símbolo del mensaje, coinciden las 'O' de la clave con el mensaje, como la 'O' de la clave es el segundo símbolo de la misma, el desplazamiento 'k' es dos y así la 'O', se cifra como 'Q' y, de esta forma varía el desplazamiento 'k' en forma sucesiva a lo largo del mensaje. Preste atención a que el último símbolo del mensaje es 'A', pero como se empareja con la 'C' inicial de la clave, la cual ocupa la primera posición de la misma, el desplazamiento 'k' es 1 nuevamente. En caso de existir más letras en el mensaje, la siguiente letra que le continuase a la 'A' se desplazaría en 2, la siguiente en 3, y así hasta llegar a 8, donde luego se volvería a reinicia la cuenta.

Con esto último, se pone de manifiesto uno de los aspectos cruciales del algoritmo que es: "La Longitud de la Clave determina el intervalo de variación del desplazamiento de k", en efecto, en el ejemplo, la clave 'corredor' tiene una longitud de 8, por lo que el desplazamiento k, estará comprendido entre:

$$1 \leq k \leq 8$$

Téngase en cuenta que cuanto más extensa es la clave, menor es el riesgo de que reconozcan el mensaje original.

El programa a continuación, denominado "Evig.cpp", implementa el Algoritmo de la Cifra de Vigenere:

Archivo: **evig.cpp**

```
1       #include <iostream.h>
2       #include <string.h>
3       #include <conio.h>
4       #include <stdlib.h>
5       #include <stdio.h>
6
7       const char clave[]="clave";
8       enum bool{FALSE,TRUE};
9
10      char* encriptarVigenere(char*);
```

```
11      char* descifrarVigenere(char*);
12
13      void main(void)
14      {
15       char cadena[]="Hola Alumnos de Cs. de la Computaci¢n";
16       clrscr();
17       cout<<"Mensaje Original: \n\t"<<cadena;
18
19       char* enc=encriptarVigenere(cadena);
20       cout<<"\nMensaje Cifrado: \n\t"<<enc;
21
22       char* desc=descifrarVigenere(enc);
23       cout<<"\nMensaje Descifrado: \n\t"<<desc;
24       delete desc;
25       delete enc;
26       desc=NULL;
27       enc=NULL;
28      }
29
30      char* encriptarVigenere(char* aenc)
31      {
32       char* enc=new char[strlen(aenc)+1];
33
34       int posclave=1;
35
36       for(int i=0;i<strlen(aenc);i++)
37          {
38            enc[i]=aenc[i]+posclave;
39
40            posclave=(posclave<=(strlen(clave)))?++posclave:1;
41          }
42
43       enc[i]='\0';
44       return enc;
45      }
46
47      char* descifrarVigenere(char* ades)
48      {
49       int longitud=strlen(ades);
50       char* enc=new char[longitud+1];
51       int posclave=1;
52
53       for(int i=0;i<strlen(ades);i++)
54          {
55            enc[i]=(short int)ades[i]-posclave;
56
57            posclave=(posclave<=(strlen(clave)))?++posclave:1;
58          }
59
```

```
60      enc[i]='\0';
61      return enc;
62     }
63
```

Algoritmo de Cifrado XOR

El Algoritmo de Cifado XOR, se basa en aplicar al símbolo del mensaje una operación XOR lógica, el símbolo resultante es el que será incorporado en el mensaje cifrado. A Priori se sabe que la operación XOR es binaria, por lo que requiere de dos elementos para poder efectuarse el cálculo lógico, aquí se tienen dos alternativas:

- Utilizar el mismo símbolo del mensaje,

Mensaje Original	H	O	L	A	L	A	U	R	A
Mensaje a utilizar con XOR	H	O	L	A	L	A	U	R	A
En el Mensaje Cifrado irá el resultado de:	H Xor H	O Xor O	L Xor L	A Xor A	L Xor L	A Xor A	U Xor U	R Xor R	A Xor A

- Utilizar una clave, que en caso de ser de inferior longitud al mensaje, se emparejaría y repetiría en serie en forma Análoga a la Cifra de Vigenere, para poder efectuar una Xor lógica menos riesgosa:

Mensaje Original	H	O	L	A	L	A	U	R	A
Mensaje a utilizar con XOR	C	O	R	R	E	D	O	R	C
En el Mensaje Cifrado irá el resultado de:	H Xor C	O Xor O	L Xor R	A Xor R	L Xor E	A Xor D	U Xor O	R Xor R	A Xor C

Vea que nuevamente, cuanto más extensa y no común fuese la clave, menos riesgo de descubrimiento del mensaje se tiene. El Algoritmo por Xor tiene una particularidad interesante, es Simétrico. Esto último implica que el algoritmo para cifrar y descifrar un mensaje es el mismo, compare los algoritmos de cifrado y descifrado a del programa "Exor" a continuación, y verá que las diferencias se deben a formalidades del lenguaje, pero no a la operatoria de cifrado/descifrado:

Archivo: **exor.cpp**

```
1       #include <iostream.h>
2       #include <string.h>
3       #include <conio.h>
4       #include <stdlib.h>
5       #include <stdio.h>
6
7       const char clave[]="clave";
8       enum bool{FALSE,TRUE};
9
10      short int* encriptarXor(char*,int&);
11      char* descifrarXor(short int*,int);
12
13      void main(void)
```

```
14      {
15       char cadena[]="Hola Alumnos de Cs. de la Computaci¢n";
16       clrscr();
17       cout<<"Mensaje Original: "<<cadena;
18
19       int longitud=0;
20       short int* enc=encriptarXor(cadena,longitud);
21       cout<<"\nMensaje Cifrado: (Los puntos son para la claridad de lectura
        solamente)\n\t";
22       for(int j=0;j<longitud;j++)cout<<enc[j]<<".";
23
24       char* desc=descifrarXor(enc,longitud);
25       cout<<"\nMensaje Descifrado: "<<desc;
26       delete desc;
27       delete enc;
28       desc=NULL;
29       enc=NULL;
30      }
31
32      short int* encriptarXor(char* aenc, int& longitud)
33      {
34       short int* enc=new short int[strlen(aenc)+1];
35       longitud=strlen(aenc);
36
37       int posclave=0;
38
39       for(int i=0;i<strlen(aenc);i++)
40          {
41            enc[i]=(short int)aenc[i]^clave[posclave];
42
43            posclave=(posclave<(strlen(clave)-1))?++posclave:0;
44          }
45
46       enc[i]=(short int)'\0';
47       return enc;
48      }
49
50      char* descifrarXor(short int* ades, int longitud)
51      {
52       char* enc=new char[longitud+1];
53       int posclave=0;
54
55       for(int i=0;i<longitud;i++)
56          {
57            enc[i]=(char)(ades[i]^((int)clave[posclave]));
58
59            posclave=(posclave<(strlen(clave)-1))?++posclave:0;
60          }
61
```

```
62      enc[i]='\0';
63      return enc;
64     }
65
```

Algoritmo de Cifrado por Rotación de Bits

Una forma interesante de cifrar es jugar con la permutación del orden de los bits. Por ejemplo, suponga que la 'H' del mensaje tiene la siguiente combinación de bits que la rerpresentan en ASCII:

Peso de los Bits		**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Primera Letra del Mensaje (Representación Binaria)	H	0	0	0	1	0	1	0	1
Rotación Absoluta de Bits		1	0	1	0	1	0	0	0

Vea que el 0 del bit '7' se traslado a la posición 0 del símbolo cifrado, el 0 del bit 6 a la posición 1, el 0 del bit 5 a la posición 4, el 1 del bit 4 ala posición 3 y así sucesivamente hasta invertir el bit absolutamente. El símbolo ASCII asociado a la combinación de bits resultante se almacena como el símbolo dentro del mensaje cifrado para la 'H'. Esta misma metodología se emplea para con los restantes símbolos del mensaje a cifrar.

Note que si emplea la inversa, al igual que en matemática:

$$f^{-1}(f(A)) = A$$

De esta forma, el algoritmo de Rotación de Bits para el caso particular de rotación absoluta, es simétrico. Téngase en cuenta que es el desarrollador del algoritmo de cifrado, quien decide cómo se permutarán los bits entre el mensaje original y el cifrado, una alternativa a la rotación absoluta podría ser:

- Los Cuatro Bits de Menor Peso se Trasladan a la posición de los cuatro bits de mayor peso y éstos últimos, ocuparán la posición de los de menor peso, o.
- Puede construir una tabla de migración de bits entre el símbolo original y el cifrado como sigue:

Posición del Bit Original	Nueva Posición
7	5
6	2
5	3
4	1
3	0
2	7
1	6
0	4

Aplicando esta tabla de migración al ejemplo original con la 'H', tenemos que:

Peso de los Bits		**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Primera Letra del Mensaje (Representación Binaria)	H	0	0	0	1	0	1	0	1
Rotación Según Tabla de Migración de Bits		1	0	0	1	0	0	1	0

Este mecanismo, de acuerdo como plantee la migración, no necesariamente será simétrico pero posee un menor riesgo de ser descifrado.

El programa a continuación, denominado "Erotbits.cpp", implementa el Algoritmo de Rotación Absoluta de Bits, note que la función para cifrar/descifrar es la misma:

Archivo: **erotbits.cpp**

```
1       #include <iostream.h>
2       #include <string.h>
3       #include <conio.h>
4       #include <math.h>
5
6       char* encriptarRotBits(char*);
7       char invertir(char);
8       void activarBit(char&,int);
9       void desactivarBit(char&,int);
10
11      void main(void)
12      {
13       char cadena[]="Hola Alumnos de Cs. de la Computaci¢n";
14       clrscr();
15       cout<<"Mensaje Original: "<<cadena;
16       char* enc=encriptarRotBits(cadena);
17       cout<<"\nMensaje Cifrado: "<<enc;
18       char* desc=encriptarRotBits(enc);
19       cout<<"\nMensaje Descifrado: "<<desc;
20       delete desc;
21       delete enc;
22       desc=NULL;
23       enc=NULL;
24      }
25
26      char* encriptarRotBits(char* aenc)
27      {
28       char* enc=new char[strlen(aenc)+1];
29
30       for(int i=0;i<strlen(aenc);i++)
31            enc[i]=(invertir(aenc[i]));
32
33       enc[i]='\0';
```

```
34      return enc;
35     }
36
37     char invertir(char data)
38     {
39      //Permite una rapida inicializacion de los bits
40      //Todos en cero
41      short int tmp=0;
42
43      for(int i=7;i>=0;i--)
44      {
45       if(data & (short int)pow(2,i))
46        {//Activar 7-i
47         activarBit((char)tmp,(7-i));
48        }
49       else
50        {//Desactivar 7-i
51         desactivarBit((char)tmp,(7-i));
52        }
53      }
54
55      return (char)tmp;
56     }
57
58     void activarBit(char& byte,int bit)
59     {
60       if(bit<0 || bit>7)
61       {
62        cout<<"\nEncriptaci¢n Defectuosa, Bit fuera de Rango";
63        return;
64       }
65
66       if(byte & ((short int)pow(2,bit)))
67       {//ya est  activado
68        return;
69       }
70       else
71       {//no est  activado
72        byte=byte | ((short int)pow(2,bit));
73        return;
74       }
75     }
76
77     void desactivarBit(char& byte,int bit)
78     {
79       if(bit<0 || bit>7)
80       {
81        cout<<"\nEncriptaci¢n Defectuosa, Bit fuera de Rango";
82        return;
83       }
```

```
84
85        if(byte & ((short int)pow(2,bit)))
86        {//ya est  activado
87         byte=byte^ ((short int)pow(2,bit));//El XOR para 1,1 da 0
88         return;
89        }
90        else
91        {//no est  activado
92         return;
93        }
94      }
```

Algoritmo de Encriptación Por Tablas

El Algoritmo de Encriptación por Tablas, consiste en reemplazar un símbolo del mensaje original por otro, definido previamente en una tabla de intercambio, por ejemplo:

- Si el Símbolo en el mensaje original es una 'A', coloque una 'Z' en su posición dentro el mensaje cifrado,
- si fuese una 'B' en el mensaje original, coloque una 'Y' en el mensaje cifrado,
- Y así sucesivamente hasta especificar todos los cambios de símbolos que se deseen.

La debilidad de este algoritmo pasa por la tabla misma, que puede ser descubierta a través de un análisis estadístico de los símbolos cifrados conociendo el idioma del generador del mensaje.

El Programa a continuación, denominado "ETabla.cpp", implementa una versión en C/C++ de la Encriptación por tablas:

Archivo: **etabla.cpp**

```
1       #include <iostream.h>
2       #include <string.h>
3       #include <conio.h>
4
5       const short int factorCesar=3;
6       enum bool{FALSE,TRUE};
7
8       char* encriptarTabla(char*);
9       char* descifrarTabla(char*);
10      char tabla(char,bool);
11
12      void main(void)
13      {
14       char cadena[]="hola alumnos de cs. de la computaci¢n";
15       clrscr();
16       cout<<"Mensaje Original: "<<cadena;
17       char* enc=encriptarTabla(cadena);
```

```
18      cout<<"\nMensaje Cifrado: "<<enc;
19      char* desc=descifrarTabla(enc);
20      cout<<"\nMensaje Descifrado: "<<desc;
21      delete desc;
22      delete enc;
23      desc=NULL;
24      enc=NULL;
25     }
26
27     char* encriptarTabla(char* aenc)
28     {
29      char* enc=new char[strlen(aenc)+1];
30
31      for(int i=0;i<strlen(aenc);i++)
32           enc[i]=tabla(aenc[i],TRUE);
33
34      enc[i]='\0';
35      return enc;
36     }
37
38     char* descifrarTabla(char* aenc)
39     {
40      char* enc=new char[strlen(aenc)+1];
41
42      for(int i=0;i<strlen(aenc);i++)
43           enc[i]=tabla(aenc[i],FALSE);
44
45      enc[i]='\0';
46
47      return enc;
48     }
49
50     char tabla(char viene,bool encripta)
51     {
52      if(encripta==TRUE)
53      {
54       switch(viene)
55       {
56        case 'a': return 'z';
57        case 'b': return 'y';
58        case 'c': return 'x';
59        case 'd': return 'w';
60        case 'e': return 'v';
61        case 'f': return 'u';
62        case 'g': return 't';
63        case 'h': return 's';
64        case 'i': return 'r';
65        case 'j': return 'q';
66        case 'k': return 'p';
67        case 'l': return 'o';
```

```
68          case 'm': return '¤';
69          case 'n': return 'n';
70          case 'ñ': return 'm';
71          case 'o': return 'l';
72          case 'p': return 'k';
73          case 'q': return 'j';
74          case 'r': return 'i';
75          case 's': return 'h';
76          case 't': return 'g';
77          case 'u': return 'f';
78          case 'v': return 'e';
79          case 'w': return 'd';
80          case 'x': return 'c';
81          case 'y': return 'b';
82          case 'z': return 'a';
83          default: return viene;///es algun s¡mbolo de puntuacion o especial
84         }
85        }
86        else
87        {
88         switch(viene)
89         {
90          case 'z': return 'a';
91          case 'y': return 'b';
92          case 'x': return 'c';
93          case 'w': return 'd';
94          case 'v': return 'e';
95          case 'u': return 'f';
96          case 't': return 'g';
97          case 's': return 'h';
98          case 'r': return 'i';
99          case 'q': return 'j';
100         case 'p': return 'k';
101         case 'o': return 'l';
102         case '¤': return 'm';
103         case 'n': return 'n';
104         case 'm': return 'ñ';
105         case 'l': return 'o';
106         case 'k': return 'p';
107         case 'j': return 'q';
108         case 'i': return 'r';
109         case 'h': return 's';
110         case 'g': return 't';
111         case 'f': return 'u';
112         case 'e': return 'v';
113         case 'd': return 'w';
114         case 'c': return 'x';
115         case 'b': return 'y';
116         case 'a': return 'z';
```

```
117        default: return viene;///es algun s¡mbolo de puntuacion o especial
118       }
119
120      }
121     }
```

Algoritmo de Encriptación Base64

El Algoritmo de Encriptación Base64 suele ser utilizado en bastantes sistemas operativos para la autenticación en áreas de confianza de los usuarios. Es un Algoritmo más potente que los anteriormente vistos, que basa su filosofía de trabajo en el Algoritmo de Rotación de Bits y en el de Encriptación por Tablas.

Consiste de las siguientes fases:

- Toma de a tres bytes de la cadena que contiene el mensaje original,
- Esos Tres bytes se convierten en 4 bytes en el mensaje cifrado como sigue:
  - Los Primeros 6 bits de mayor peso, del primer byte del grupo de tres, se colocan desde la posición 5 a la 0, en ese orden, del primer byte del grupo de cuatro y en la posición 7 y 6 se colocan '0',
  - Los dos bits remanentes del primer byte del grupo de tres, y los primeros 4 bits de mayor peso del segundo byte del grupo de tres, se colocan desde la posición 5 a la 0, en ese orden, del segundo byte del grupo de 4, y los bits 7 y 6 se ponen en cero.
  - Los 4 bits remanente del segundo byte mas los dos bits de mayor peso del tercer byte del grupo de tres, se colocan desde la posición 5 a la 0, en ese orden, del tercer byte del grupo de cuatro, y los bits 7 y 6 se ponen en cero.
  - Los Seis bits remanentes del tercer byte del grupo de tres, se colocan desde la posición 5 a la 0, en ese orden, del cuarto byte del grupo de cuatro, y se ponen los bits 7 y 6 en cero.

Una Vez transformados los tres bytes en cuatro bytes, cada byte del grupo de cuatro solo puede representar 64 símbolos ya que, recuerde que los bits 7 y 6 siempre están en cero, con lo que el último y máximo valor de una tabla de intercambio, podría ser:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	
0	**0**	*1*	*1*	*1*	*1*	*1*	*1*	
		2^5	2^4	2^3	2^2	2^1	2^0	$\sum_{i=0}^{5} 2^i = 64$

De cada uno de los cuatro bytes, se toma su representación entera, un valor entre 0 y 64, que será reemplazado por uno de los 64 símbolos una la tabla de intercambio, y éste último símbolo, es el que será almacenado en el mensaje cifrado. Por consiguiente, en el mensaje cifrado, solo existirán símbolos de la tabla de intercambio.

De la particularidad del tamaño de la tabla, surge el nombre del Algoritmo, "Base64", ya que cualquiera sea la información del mensaje, a través de la migración de bits y el intercambio de símbolos con la tabla, el mensaje cifrado siempre contendrá alguna variante de los 64 símbolos de la tabla.

Una vez que reemplazo los primeros tres bytes del mensaje, por los cuatro símbolos de la tabla en el mensaje cifrado, se toman los siguientes tres bytes del mensaje original y se efectúa la misma operatoria; de eta forma se posigue hasta el final.

Pero en el final, salvo que la longitud del mensaje sea múltiplo de tres, puede ocurrir que solo tenga que convertir uno o dos bytes solamente, entonces:

- Cuando se codifican dos bytes, los bytes forman una secuencia de 16 bits, entonces para completar los 18 bits, se agregan al final de los primeros 16, dos bits en cero. Estos 18 bits, incluyendo los dos bits ceros agregados al final, forman tres símbolos únicamente:
    - El primer byte: Bit 7 y 6 en Cero, y los bits desde 5 a 0, en ese orden, se llenan con los primeros 6 bits de los 18,
    - El Segundo Byte: Bit 7 y 6 en Cero, y los bits desde 5 a cero, en ese orden, se llenan con los segundos 6 bits de los 18,
    - El Tercer Byte: Bit 7 y 6 en Cero, y los bits desde 5 a cero, en ese orden, se llenan con los terceros 6 bits de los 18,
    - El Cuarto Byte se rellena con **un** carácter especial, que no este dentro de los 64 símbolos de la tabla, Ejemplo: "<", que permitirá detectar esta situación al descifrar el mensaje.
- Cuando se codifica un byte, este forma una secuencia de 8 bits, por lo que se deberá agregar al final 4 bits en cero para completar los 12 y representar dos símbolos de la tabla, como sigue:
    - El Primer Byte: Bit 7 y 6 en Cero, y los bits desde 5 a 0, en ese orden, se llenan con los primeros 6 bits de los 12,
    - El Segundo Byte: Bit 7 y 6 en cero, y los bits desde 5 a 0, en ese orden, se llenan con los segundos 6 bits de los 12,
    - El Tercer y Cuarto Byte se rellena con **dos** carácteres especiales, que no esten dentro de los 64 símbolos de la tabla, Ejemplo: "<", que permitirá detectar esta situación al descifrar el mensaje.

El programa a continuación, denominado "Ebase64.cpp", implementa el algoritmo de Encriptación Base64 como sigue:

Archivo: **ebase64.cpp**

```
1       #include <iostream.h>
2       #include <string.h>
3       #include <conio.h>
4       #include <stdlib.h>
5       #include <stdio.h>
6       #include <math.h>
7
8       enum bool{FALSE,TRUE};
9
10      char* encriptarBase64(char*);
11      char* descifrarBase64(char*);
12      char tablaCifra(short int);
13      short int tablaDescifra(char);
```

```
14      void activarBit(char&,int);
15      void desactivarBit(char&,int);
16      char* queda1Byte(char);
17      char* queda2Bytes(char*);
18      char* extraer1Byte(char*);
19      char* extraer2Bytes(char*);
20
21      void main(void)
22      {
23       char cadena[]="Hola Alumnos de Cs. de la Computacion";
24       clrscr();
25       cout<<"Mensaje Original: "<<cadena;
26
27       char* enc=encriptarBase64(cadena);
28       cout<<"\nMensaje Cifrado:\n\t"<<enc;
29
30       char* desenc=descifrarBase64(enc);
31       cout<<"\nMensaje Descifrado: \n\t"<<desenc;
32
33
34       delete desenc;
35       delete enc;
36       desenc=NULL;
37       enc=NULL;
38      }
39
40
41      char tablaCifra(short int valor)
42      {
43       switch(valor)
44       {//Tabla Base 64 Explicita
45        case 0: return 'A';
46        case 1: return 'B';
47        case 2: return 'C';
48        case 3: return 'D';
49        case 4: return 'E';
50        case 5: return 'F';
51        case 6: return 'G';
52        case 7: return 'H';
53        case 8: return 'I';
54        case 9: return 'J';
55        case 10: return 'K';
56        case 11: return 'L';
57        case 12: return 'M';
58        case 13: return 'N';
59        case 14: return 'O';
60        case 15: return 'P';
61        case 16: return 'Q';
62        case 17: return 'R';
63        case 18: return 'S';
```

```
64        case 19: return 'T';
65        case 20: return 'U';
66        case 21: return 'V';
67        case 22: return 'W';
68        case 23: return 'X';
69        case 24: return 'Y';
70        case 25: return 'Z';
71        case 26: return 'a';
72        case 27: return 'b';
73        case 28: return 'c';
74        case 29: return 'd';
75        case 30: return 'e';
76        case 31: return 'f';
77        case 32: return 'g';
78        case 33: return 'h';
79        case 34: return 'i';
80        case 35: return 'j';
81        case 36: return 'k';
82        case 37: return 'l';
83        case 38: return 'm';
84        case 39: return 'n';
85        case 40: return 'o';
86        case 41: return 'p';
87        case 42: return 'q';
88        case 43: return 'r';
89        case 44: return 's';
90        case 45: return 't';
91        case 46: return 'u';
92        case 47: return 'v';
93        case 48: return 'w';
94        case 49: return 'x';
95        case 50: return 'y';
96        case 51: return 'z';
97        case 52: return '0';
98        case 53: return '1';
99        case 54: return '2';
100       case 55: return '3';
101       case 56: return '4';
102       case 57: return '5';
103       case 58: return '6';
104       case 59: return '7';
105       case 60: return '8';
106       case 61: return '9';
107       case 62: return '+';
108       case 63: return '/';
109      }
110
111      cout<<"\nError de Procesamiento, Valor no encontrado ("<<valor<<")";
112      exit(0);
```

```
113     return 63;//dummy
114    }
115
116    short int tablaDescifra(char valor)
117    {
118     switch(valor)
119     {//Tabla Base 64 Explicita
120      case 'A': return 0;
121      case 'B': return 1;
122      case 'C': return 2;
123      case 'D': return 3;
124      case 'E': return 4;
125      case 'F': return 5;
126      case 'G': return 6;
127      case 'H': return 7;
128      case 'I': return 8;
129      case 'J': return 9;
130      case 'K': return 10;
131      case 'L': return 11;
132      case 'M': return 12;
133      case 'N': return 13;
134      case 'O': return 14;
135      case 'P': return 15;
136      case 'Q': return 16;
137      case 'R': return 17;
138      case 'S': return 18;
139      case 'T': return 19;
140      case 'U': return 20;
141      case 'V': return 21;
142      case 'W': return 22;
143      case 'X': return 23;
144      case 'Y': return 24;
145      case 'Z': return 25;
146      case 'a': return 26;
147      case 'b': return 27;
148      case 'c': return 28;
149      case 'd': return 29;
150      case 'e': return 30;
151      case 'f': return 31;
152      case 'g': return 32;
153      case 'h': return 33;
154      case 'i': return 34;
155      case 'j': return 35;
156      case 'k': return 36;
157      case 'l': return 37;
158      case 'm': return 38;
159      case 'n': return 39;
160      case 'o': return 40;
161      case 'p': return 41;
162      case 'q': return 42;
```

```
163      case 'r': return 43;
164      case 's': return 44;
165      case 't': return 45;
166      case 'u': return 46;
167      case 'v': return 47;
168      case 'w': return 48;
169      case 'x': return 49;
170      case 'y': return 50;
171      case 'z': return 51;
172      case '0': return 52;
173      case '1': return 53;
174      case '2': return 54;
175      case '3': return 55;
176      case '4': return 56;
177      case '5': return 57;
178      case '6': return 58;
179      case '7': return 59;
180      case '8': return 60;
181      case '9': return 61;
182      case '+': return 62;
183      case '/': return 63;
184     }
185
186     cout<<"\nError de Procesamiento, Valor no encontrado ("<<valor<<")";
187     exit(0);
188     return 63;//dummy
189    }
190
191    void activarBit(char& byte,int bit)
192    {
193      if(bit<0 || bit>7)
194      {
195       cout<<"\nEncriptaci¢n Defectuosa, Bit fuera de Rango";
196       return;
197      }
198
199      if(byte & ((short int)pow(2,bit)))
200      {//ya est  activado
201       return;
202      }
203      else
204      {//no est  activado
205       byte=byte | ((short int)pow(2,bit));
206       return;
207      }
208    }
209
210    void desactivarBit(char& byte,int bit)
211    {
```

```
212      if(bit<0 || bit>7)
213      {
214       cout<<"\nEncriptaci¢n Defectuosa, Bit fuera de Rango";
215       return;
216      }
217
218      if(byte & ((short int)pow(2,bit)))
219      {//ya est  activado
220       byte=byte^ ((short int)pow(2,bit));//El XOR para 1,1 da 0
221       return;
222      }
223      else
224      {//no est  activado
225       return;
226      }
227    }
228
229    char* procesaTriplo(char* tmp3bytes)
230    {
231     if(strlen(tmp3bytes)!=3)
232     {
233      cout<<"\nError en la Encriptaci¢n, (triplo distinto de 3):
       "<<strlen(tmp3bytes);
234      cout<<"\n cad:"<<tmp3bytes;
235      exit(0);
236     }
237
238     char* enc=new char[5];
239     int bitorigen=7;
240     int bitdest=5;
241
242     int ready=1;
243     for(int Aorigen=0,Adest=0;Adest<4;Aorigen++)
244      {
245         desactivarBit(enc[Adest],7);
246         desactivarBit(enc[Adest],6);
247
248         for(;bitdest>=0 && ready;bitdest--)
249         {
250          if(tmp3bytes[Aorigen]&((short int)pow(2,bitorigen)))
251          {//Bit de Origen Activado
252             activarBit(enc[Adest],bitdest);
253             bitorigen--;
254          }
255          else
256          {//Bit de Origen Desactivado
257             desactivarBit(enc[Adest],bitdest);
258             bitorigen--;
259          }
260
```

```
261           if(bitorigen<0)
262             {
263              ready=0;
264              bitorigen=7;
265             }
266         }//fin del for interno
267
268         if(bitdest<0)
269         {
270          Adest++;
271          bitdest=5;
272
273          if(bitorigen>0)//No termino de examinar el byte de origen
274              Aorigen--;//trampa para que permanezca en el mismo origen
275         }
276
277         if(bitorigen==7) ready=1;
278      }
279
280      enc[4]='\0';
281      return enc;
282    }
283
284    char* encriptarBase64(char* data)
285    {
286     int longitud=strlen(data);
287
288     if(data==NULL || longitud==0)
289     {
290      clrscr();
291      cout<<"\nDato a Encriptar Invalido";
292      exit(0);
293     }
294
295     bool isMultiplo3=((longitud%3)==0)?TRUE:FALSE;
296
297     char* encriptada=new char[longitud+((int)(longitud/3))+3];
298
299     if(encriptada==NULL)
300      {
301       cout<<"\nNo Hay Memoria Suficiente para Encriptar";
302       exit(0);
303      }
304
305     char triplo[4];
306     triplo[3]='\0';//Inicializo bits a cero
307
308     char *enctriplo=NULL;
309
```

```
310     for(int i=0,j=0;(i+3)<=longitud;i+=3,j+=4)//Avanza de a Tres Caracteres
311      {
312       triplo[0]=data[i];
313       triplo[1]=data[i+1];
314       triplo[2]=data[i+2];
315       triplo[3]='\0';
316
317       enctriplo=procesaTriplo((char*)triplo);
318
319       if(enctriplo==NULL)
320        {
321         cout<<"\nError de Procesamiento de Triplo";
322         exit(0);
323        }
324
325       encriptada[j]=tablaCifra((short int)enctriplo[0]);
326       encriptada[j+1]=tablaCifra((short int)enctriplo[1]);
327       encriptada[j+2]=tablaCifra((short int)enctriplo[2]);
328       encriptada[j+3]=tablaCifra((short int)enctriplo[3]);
329
330       delete enctriplo;
331       enctriplo=NULL;//libero memoria del enctriplo provisional
332      }
333
334     if(!isMultiplo3)
335     {
336      switch(longitud-(i))
337      {
338       case 1:
339             enctriplo=queda1Byte(data[i]);
340
341             if(enctriplo==NULL)
342               {
343                cout<<"\nError de Procesamiento del ultimo Byte";
344                exit(0);
345               }
346
347              encriptada[j]=tablaCifra((short int)enctriplo[0]);
348              encriptada[j+1]=tablaCifra((short int)enctriplo[1]);
349              encriptada[j+2]=enctriplo[2];//caracter especial
350              encriptada[j+3]=enctriplo[3];//caracter especial
351
352              delete enctriplo;
353              enctriplo=NULL;//libero memoria del enctriplo provisional
354             break;
355       case 2:
356             char dos[2];
357             dos[0]=data[i];
358             dos[1]=data[i+1];
359             enctriplo=queda2Bytes(dos);
```

```
360
361             if(enctriplo==NULL)
362               {
363                cout<<"\nError de Procesamiento de Dos ultimos Bytes";
364                exit(0);
365               }
366
367              encriptada[j]=tablaCifra((short int)enctriplo[0]);
368              encriptada[j+1]=tablaCifra((short int)enctriplo[1]);
369              encriptada[j+2]=tablaCifra((short int)enctriplo[2]);
370              encriptada[j+3]=enctriplo[3];//caracter especial
371
372              delete enctriplo;
373              enctriplo=NULL;//libero memoria del enctriplo provisional
374             break;
375       default:
376              cout<<"\nError de Calculo:'Desfasaje longitud-(i-1)':
      "<<(longitud-(i-1));
377              cout<<"\nLongitud: "<<longitud;
378              cout<<"\nPos. Actual: "<<i;
379              exit(0);
380              break;
381      }
382     }
383
384     encriptada[j+4]='\0';
385     return encriptada;
386    }
387
388    char* queda1Byte(char caracter)
389    {
390     char* enc=new char[5];
391     short int bitorigen=7;
392     short int bitdest=5;
393     short int Adest=0;
394
395     for(;bitorigen>=0;bitdest--)
396     {
397         desactivarBit(enc[Adest],7);
398         desactivarBit(enc[Adest],6);
399
400          if(caracter&((short int)pow(2,bitorigen)))
401          {//Bit de Origen Activado
402            activarBit(enc[Adest],bitdest);
403            bitorigen--;
404          }
405          else
406          {//Bit de Origen Desactivado
407            desactivarBit(enc[Adest],bitdest);
```

```
408             bitorigen--;
409           }
410
411           if(bitorigen==1)
412            {
413             Adest++;//queda bit 1 y 0
414             bitdest=6;//al decrementar queda en 5 --truco--
415            }
416      }
417
418      desactivarBit(enc[Adest],3);
419      desactivarBit(enc[Adest],2);
420      desactivarBit(enc[Adest],1);
421      desactivarBit(enc[Adest],0);
422      enc[Adest+1]='<';
423      enc[Adest+2]='<';
424      enc[Adest+3]='\0';
425
426      return enc;
427     }
428
429     char* queda2Bytes(char* data)
430     {
431      if(data==NULL && strlen(data)!=2)
432      {
433       cout<<"\nError en el Procesamiento de los dos £ltimos Caracteres";
434       exit(0);
435      }
436
437      char* enc=new char[5];
438      int bitorigen=7;
439      int bitdest=5;
440
441      int ready=1;
442      for(int Aorigen=0,Adest=0;!(Adest==2 && bitorigen==7);Aorigen++)
443       {
444          desactivarBit(enc[Adest],7);
445          desactivarBit(enc[Adest],6);
446
447          for(;bitdest>=0 && ready;bitdest--)
448          {
449           if(data[Aorigen]&((short int)pow(2,bitorigen)))
450           {//Bit de Origen Activado
451              activarBit(enc[Adest],bitdest);
452              bitorigen--;
453           }
454           else
455           {//Bit de Origen Desactivado
456              desactivarBit(enc[Adest],bitdest);
457              bitorigen--;
```

```
458          }
459
460           if(bitorigen<0)
461             {
462              ready=0;
463              bitorigen=7;
464             }
465         }//fin del for interno
466
467         if(bitdest<0)
468         {
469          Adest++;
470          bitdest=5;
471
472          if(bitorigen>0)//No termino de examinar el byte de origen
473              Aorigen--;//trampa para que permanezca en el mismo origen
474         }
475
476         if(bitorigen==7) ready=1;
477      }
478
479     desactivarBit(enc[Adest],1);
480     desactivarBit(enc[Adest],0);
481     enc[Adest+1]='<';
482     enc[Adest+2]='\0';
483
484     return enc;
485    }
486
487    char* extraerTriplo(char* data)
488    {
489     if(data==NULL)
490     {
491      cout<<"\nError en la Extracci¢n del Triplo";
492      cout<<"\n cad: NULA";
493      exit(0);
494     }
495
496     if(strlen(data)!=4)
497     {
498      cout<<"\nError en la Extracci¢n del Triplo";
499      cout<<"\n cad: "<<data;
500      exit(0);
501     }
502
503     //Obtengo los valores numericos asociados a los simbolos
504     data[0]=(char)tablaDescifra(data[0]);
505     data[1]=(char)tablaDescifra(data[1]);
506     data[2]=(char)tablaDescifra(data[2]);
```

```
507     data[3]=(char)tablaDescifra(data[3]);
508
509     char* extrae=new char[4];
510     int Aorigen=0;
511     int Adest=0;
512     int ready=1;
513     int bitO=5;
514     int bitD=7;
515
516     for(;Aorigen<4;Adest++)
517     {
518       for(;bitO>=0 && ready;bitO--)
519       {
520        if(data[Aorigen]&((short int)pow(2,bitO)))
521          {//Bit de Origen Activado
522            activarBit(extrae[Adest],bitD);
523            bitD--;
524          }
525          else
526          {//Bit de Origen Desactivado
527            desactivarBit(extrae[Adest],bitD);
528            bitD--;
529          }
530
531           if(bitD<0)
532            {
533             ready=0;
534             bitD=7;
535            }
536       }//fin del for interno
537
538      if(bitO<0)
539       {
540        Aorigen++;
541        bitO=5;
542
543        if(bitD>0)//No termino de examinar el byte de origen
544         Adest--;//trampa para que permanezca en el mismo destino
545        }
546
547      if(bitD==7) ready=1;
548     }
549
550     extrae[3]='\0';
551     return extrae;
552    }
553
554    char* descifrarBase64(char* data)
555    {
556     int longitud=strlen(data);
```

```
557
558     if(data==NULL || longitud==0)
559     {
560      clrscr();
561      cout<<"\nDato a Encriptar Invalido";
562      exit(0);
563     }
564
565     char aextraer[5];
566     char* des=new char[longitud];
567
568     for(int i=0,j=0;(i+4)<=(longitud-4);i+=4,j+=3)
569     {
570      aextraer[0]=data[i];
571      aextraer[1]=data[i+1];
572      aextraer[2]=data[i+2];
573      aextraer[3]=data[i+3];
574      aextraer[4]='\0';
575
576      char* rdo=extraerTriplo(aextraer);
577
578      if(rdo==NULL)
579      {
580       cout<<"\nError en la Extracci¢n";
581       exit(0);
582      }
583
584      des[j]=rdo[0];
585      des[j+1]=rdo[1];
586      des[j+2]=rdo[2];
587
588      delete rdo;
589      rdo=NULL;
590     }
591
592      aextraer[0]=data[i];
593      aextraer[1]=data[i+1];
594      aextraer[2]=data[i+2];
595      aextraer[3]=data[i+3];
596      aextraer[4]='\0';
597
598      if(aextraer[2]=='<')
599       {
600        char* rdo=extraer1Byte(aextraer);
601
602        if(rdo==NULL)
603        {
604         cout<<"\nError en la Extracci¢n";
605         exit(0);
```

```
606         }
607
608         des[j]=rdo[0];
609         des[j+1]='\0';
610
611         delete rdo;
612         rdo=NULL;
613
614         return des;
615        }
616       else
617        {
618         if(aextraer[3]=='<')
619         {
620         char* rdo=extraer2Bytes(aextraer);
621
622         if(rdo==NULL)
623         {
624          cout<<"\nError en la Extracci¢n";
625          exit(0);
626         }
627
628         des[j]=rdo[0];
629         des[j+1]=rdo[1];
630         des[j+2]='\0';
631
632         delete rdo;
633         rdo=NULL;
634
635         return des;
636         }
637        }
638
639       //No tiene caracteres de rellenos
640       char* rdo=extraerTriplo(aextraer);
641
642       if(rdo==NULL)
643       {
644        cout<<"\nError en la Extracci¢n";
645        exit(0);
646       }
647
648       des[j]=rdo[0];
649       des[j+1]=rdo[1];
650       des[j+2]=rdo[2];
651       des[j+3]='\0';
652
653       delete rdo;
654       rdo=NULL;
655
```

```
656      return des;
657    }
658
659    char* extraer2Bytes(char* data)
660    {
661     if(data==NULL && strlen(data)!=4)
662     {
663      cout<<"\nError en el Procesamiento de los cuatro £ltimos Caracteres";
664      exit(0);
665     }
666
667     char* enc=new char[5];
668     data[0]=(char)tablaDescifra(data[0]);
669     data[1]=(char)tablaDescifra(data[1]);
670     data[2]=(char)tablaDescifra(data[2]);
671     //el restante es un caracter de relleno
672
673     int bitorigen=5;
674     int bitdest=7;
675
676     int ready=1;
677     for(int Adest=0,Aorigen=0;!(Aorigen==2 && bitdest==7);Adest++)
678      {
679         for(;bitorigen>=0 && ready;bitorigen--)
680         {
681          if(data[Aorigen]&((short int)pow(2,bitorigen)))
682          {//Bit de Origen Activado
683             activarBit(enc[Adest],bitdest);
684             bitdest--;
685          }
686          else
687          {//Bit de Origen Desactivado
688             desactivarBit(enc[Adest],bitdest);
689             bitdest--;
690          }
691
692           if(bitdest<0)
693             {
694              ready=0;
695              bitdest=7;
696             }
697         }//fin del for interno
698
699         if(bitorigen<0)
700         {
701          Aorigen++;
702          bitorigen=5;
703
704          if(bitdest>0)//No termino de examinar el byte de origen
```

```
705             Adest--;//trampa para que permanezca en el mismo origen
706         }
707
708         if(bitdest==7) ready=1;
709      }
710
711     enc[2]='\0';
712
713     return enc;
714    }
715
716    char* extraer1Byte(char* data)
717    {
718     if(data==NULL && strlen(data)!=4)
719     {
720      cout<<"\nError en el Procesamiento de los cuatro £ltimos Caracteres";
721      exit(0);
722     }
723
724     char* enc=new char[5];
725     data[0]=(char)tablaDescifra(data[0]);
726     data[1]=(char)tablaDescifra(data[1]);
727     //los dos restantes son de relleno
728
729     int bitorigen=5;
730     int bitdest=7;
731
732     int ready=1;
733     for(int Adest=0,Aorigen=0;!(Aorigen==1 && bitdest==7);Adest++)
734      {
735         for(;bitorigen>=0 && ready;bitorigen--)
736         {
737          if(data[Aorigen]&((short int)pow(2,bitorigen)))
738          {//Bit de Origen Activado
739             activarBit(enc[Adest],bitdest);
740             bitdest--;
741          }
742          else
743          {//Bit de Origen Desactivado
744             desactivarBit(enc[Adest],bitdest);
745             bitdest--;
746          }
747
748           if(bitdest<0)
749             {
750              ready=0;
751              bitdest=7;
752             }
753         }//fin del for interno
754
```

```
755         if(bitorigen<0)
756         {
757          Aorigen++;
758          bitorigen=5;
759
760          if(bitdest>0)//No termino de examinar el byte de origen
761              Adest--;//trampa para que permanezca en el mismo origen
762         }
763
764         if(bitdest==7) ready=1;
765      }
766
767     enc[1]='\0';
768
769     return enc;
770    }
```

Referencias

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995.

- "Seguridad en Java"; Jamie Kaworski y Paul Perrone; Editorial Prentice Hall; España, 2001.

- "Reviewed Pappers: Cryptography and Security protocols course for undergraduate IT students"; Alaaeldin Aly y Shakil Akhtar; ACM SIGCSE Bulletin, Volumen 36, Edición 2; Junio de 2004.

- "RFC 3548 – The Base 16, 32 and 64 Data Encodings", S. Jofesson Ed.; The Internet Society (Julio de 2003); http://www.faqs.org/rfcs/rfc3548.html, disponible al 23 de Marzo de 2005.

10 Introducción a la Teoría de Grafos

Introducción al Concepto de Grafos

Se entiende por ***grafos,*** a un conjunto de nodos (o aristas) que se interrelacionan o vinculan a través de un conjunto de arcos (bordes).

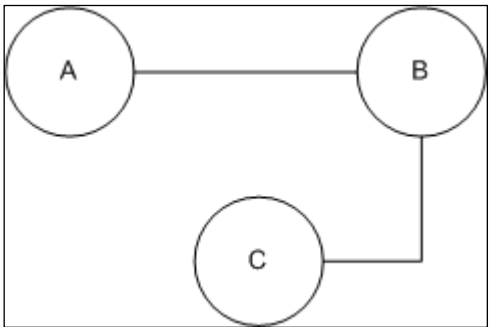


Figura 171: "Conceptualización de un Grafo"

El Conjunto de Nodos de la Figura anterior, queda definido como:

$$N = \{A, B, C\}$$

Mientras que el conjunto de Arcos queda definido como:

$$Ar\cos = \{(A;B),(B;C)\}$$

Este caso de grafos, donde el orden de los componentes del arco son conmutativos, entendiéndose por ello que:

(A;B) = (B;A)

Se los denomina grafos ***no dirigidos o sin orientación definida.***

Cuando el orden de los componentes del arco dejan de ser conmutativos y se tiene que el primer componente del par, representa el origen y, el segundo componente del par, representa el destino, estamos en presente de un grafo ***dirigido o con orientación definida.***

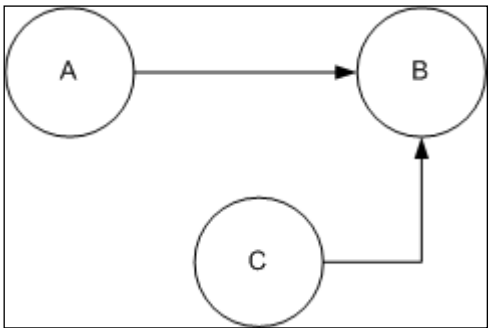


Figura 172: "Conceptualización de un Grafo Dirigido"

El Grado dirigido, queda formalmente definido como sigue:

Nodos = {A,B,C}
Arcos: {(A;B),(C;B)}

Note que los componentes de los elementos del conjunto Arcos, no son conmutativos, sino que su orden refleja el punto de origen y destino, Ej: (C;B) el arco parte desde el nodo 'C' y llega hasta el nodo 'B'.

Un nodo cualquiera 'N' del conjunto de nodos, es ***incidente*** en un arco 'A' del conjunto de Arcos, si el primero forma parte del origen y/o el destino del arco. El ***grado*** de un nodo, es la cantidad de arcos incidentes en él Ej: El nodo B de la figura anterior tiene un grado de 2, ya que es incidente al arco (A;B) y al arco (C;B). El ***grado interno*** de un nodo, es la cantidad de arcos entrantes al mismo Ej: El nodo A tiene un grado interno de 0 (cero), mientras que el 'B' tiene un grado interno de 2 (dos). El ***grado externo*** de un nodo, es la cantidad de arcos salientes al mismo Ej: El nodo A tiene un grado externo de 1 (uno), mientras que el nodo B, tiene un grado externo de 0 (cero). Se dice que un nodo 'A' es ***adyacente*** a otro nodo 'B', si y solo si existe un arco que los vincula, independientemente de si es dirigido o no Ej: El nodo 'A' con el nodo 'B', están vinculados a través del arco (A;B).

Una ***relación*** 'R' en un conjunto 'Nodos', como el analizado, es un conjunto de pares ordenados de elementos de A, donde el par ordenado cumple con una lógica de vinculación entre nodos. Si <a,b> es un arco que representa la instanciación de la relación, se dice que los nodos 'a' y 'b' cumplen con la lógica de vinculación entre nodos, entonces 'a' está vinculado con 'b' a través de la relación 'R'. Ej: Dado el conjunto 'Nodos', podríamos definir la Relación 'R', como que el origen deberá ser siempre inferior, menor o antecesor al destino. Esto último, a partir del conjunto 'Nodos', daría el siguiente conjunto de Arcos:

Arcos: {(A;B),(B;C),(A;C)}

A partir del conjunto de Arcos, interpretándolos como dirigidos de acuerdo a lo que explicita su relación, se obtendría el siguiente grafo:

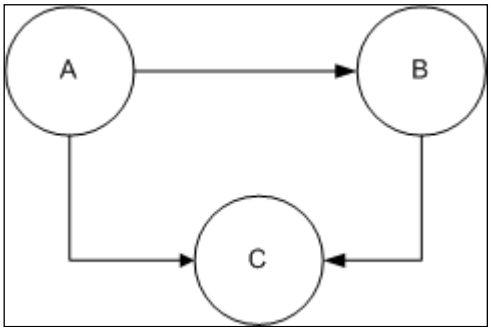


Figura 173: "Conceptualización de un Grafo Dirigido surgido de una Relación 'R'"

Asociado a cada arco, se puede vincular un ***peso o costo de red***, que representa una magnitud concreta (de dinero, de caudal, de flujo en general, entre otras dependiendo la aplicación del grafo) de lo que cuesta utilizar un arco determinado para trasladarse entre el origen y el destino asociado por éste.

Una ***Trayectoria de longitud k*** de un nodo 'A' a otro 'C', se define como una secuencia *(k+1)* nodos n_1, n_2, n_3,… , n_{k+1}, tal que n_1='A' y n_{k+1}='C' y, n_i con n_{i+1} son adyacentes para todos los 'i' comprendidos entre '1' y 'k'. Ejemplo: Se desea conocer la longitud de la trayectoria entre el nodo 'A' y 'C', teniendo en cuenta el siguiente grafo:

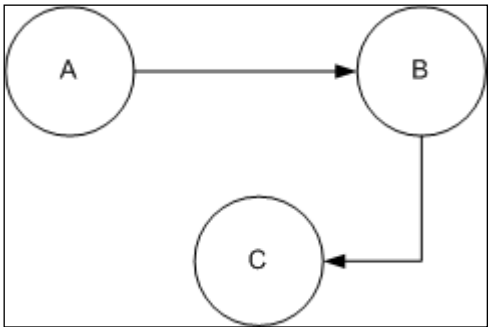


Figura 174: "Grafo Dirigido Acíclico"

Se tiene que n_A, n_B y n_C son los componentes del trayecto, donde n_A=A y n_C=C, gráficamente se puede observar que existe adyacencia entre (A;B) y (B;C), por lo que el trayecto tiene una longitud de 2. Una forma alternativa de definir la Trayectoria de longitud k entre dos nodos, está asociada a los arcos que vinculan a los nodos que componen el trayecto y dice:

"La Longitud de la Trayectoria entre dos nodos, 'A' y 'C', está dada por la cantidad de arcos que permiten llegar desde el nodo origen hasta el destino, para un trayecto determinado"

Empleando esta última definición, note que para el camino escogido, y en este caso el único, para llegar al nodo 'C' a partir del nodo 'A', se atraviesan dos arcos (A→B y B→C), por lo que la longitud de la trayectoria sería de 2.

Una trayectoria de un nodo a sí mismo, se denomina ***ciclo***. Si un grafo posee al menos un ciclo, se lo denomina ***grafo cíclico***, mientras que si no posee ningún ciclo (como los vistos hasta el momento), se lo denomina ***grafo acíclico***, tambien conocidos como ***dag*** por sus siglas en inglés.

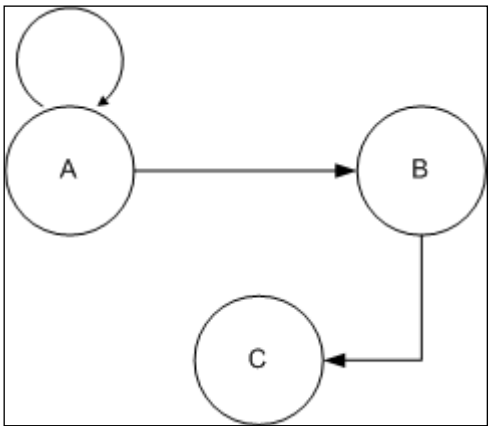


Figura 175: "Grafo Dirigido Cíclico"

Note que en el caso de la existencia de ciclos, dentro del conjunto 'Arcos', deberán existir elementos donde sus componentes coincidan en cuanto al origen y el

destino. Ejemplo: El grafo de la figura anterior quedaría formalmente definido como sigue:

Nodos = {A,B,C}
Arcos: {**(A;A)**,(A;B),(B;C)}

Matriz de Adyacencia

La Matriz de adyacencia es una matriz bidimensional cuadrada que representa los arcos existentes entre los nodos que constituyen el grafo. Los encabezados de las filas (origen) y columnas (destino) de la matriz, representan los nodos del grafo y, en la intersección entre ellos puede ocurrir que:

- Exista un '1': lo que implica que existe un arco que vincula ambos nodos,
- Exista un '0': lo que implica que no existe un arco que vincula los nodos.

Si se toma el grafo cíclico de la figura del apartado anterior, la matriz de adyacencia sería:

	A	**B**	**C**
A	1	1	0
B	0	0	1
C	0	0	0

Figura 176: "Matriz de Adyacencia del Grafo Dirigido Cíclico de la Figura Anterior"

Recuerde que en el encabezado de la fila representa el origen, mientras que el encabezado de la columna representa el destino, entonces, al leer la matriz de adyacencia se tiene:

- Existe un arco desde el nodo 'A' hasta el nodo 'A', un ciclo, dado por:

	A	**B**	**C**
A	**1**	1	0
B	0	0	1
C	0	0	0

Figura 177: "Matriz de Adyacencia, Representación de un Ciclo"

- Existe un arco desde el nodo 'A' al nodo 'B', dado por:

	A	**B**	**C**
A	1	**1**	0
B	0	0	1
C	0	0	0

Figura 178: "Matriz de Adyacencia, Representación de un Arco"

De esta misma forma, se lee el resto de la matriz, considerando el origen en el encabezado de la fila y el destino en el encabezado de la columna. Tenga en cuenta que la matriz de adyacencia es una matriz cuadrada, ya que debe de considerar todas las

alternativas para un conjunto de 'N' nodos, lo que da en el máximo de los casos N^2 arcos (Una Vinculación entre Nodos Absoluta –Todos con Todos-).

A partir de esta situación, se puede inferir que es posible implementar en C/C++, en forma sencilla, la matriz de adyacencia como una arreglo bidimensional cuadrado, donde el orden quedará establecido por la cantidad de nodos del grafo.

Cierre Transitivo

Suponga que un grafo se representa solo por su matriz de adyacencia, lo que implica que se omiten los detalles de pesos en el arco entre otras cuestiones, se podría saber si tengo forma de llegar hasta un destino no conectado directamente mediante un arco al origen, como sigue:

Ady[i][t] AND Ady[t][k]

- **Ady:** Representa, en forma abreviada, a la matriz de adyacencia.
- **Ady[i][t]:** Representa el arco, si existiese un 1 en dicha posición, que vincula al nodo 'i' con el 't'.
- **Ady[t][k]:** Representa el arco, si existiese un 1 en dicha posición, que vincula al nodo 't' con el 'k'.
- **AND:** Representa la operación 'Y' lógica, que en C se implementa con && a nivel de variable.

Lo interesante de la operación enunciada anteriormente es que, de dar:

- **TRUE**: Implica que existe un camino para llegar desde el nodo 'i' al nodo 'k', a través de 't', con una longitud 2,
- **FALSE**: Implica que no existe un camino para llegar desde el nodo 'i' al nodo 'k', a través de 't', con una longitud 2.

Si aplicamos dicha operación a la matriz de adyacencia:

Ady_2=Ady AND Ady

Se puede obtener una nueva matriz denominada ***"Matriz de Trayectoria de Longitud 2"***, donde se encuentrarán solo los trayectos que permiten llegar con una longitud de '2', desde un nodo origen a otro destino. Se dice también que Ady_2 es el producto boleano de la matriz de adyacencia consigo misma.

1	1	1
0	0	0
0	0	0

Matriz de Trayectoria de Longitud 2

=

1	1	0
0	0	1
0	0	0

Matriz de Adyacencia

AND

1	1	0
0	0	1
0	0	0

Matriz de Ayacencia

Vea el cómo se obtiene, paso a paso mediante la operación del producto boleano de matrices, la matriz de trayectoria de longitud 2:

Para i = 0 y j = 0

Ady_2[0][0]=(Ady[**0**][0] && Ady [0][**0**]) ||

1	**1**	**0**

(Ady[**0**][1] && Ady[1][**0**]) ||
(Ady[**0**][2] && Ady[2][**0**]) ||

0	0	1
0	0	0

Para i = 0 y j = 1

Ady$_2$[0][1]=(Ady[**0**][0] && Ady [0][**1**]) ||
(Ady[**0**][1] && Ady[1][**1**]) ||
(Ady[**0**][2] && Ady[2][**1**]) ||

1	**1**	**0**
0	**0**	1
0	**0**	0

Para i = 0 y j = 2

Ady$_2$[0][2]=(Ady[**0**][0] && Ady [0][**2**]) ||
(Ady[**0**][1] && Ady[1][**2**]) ||
(Ady[**0**][2] && Ady[2][**2**]) ||

1	**1**	**0**
0	0	**1**
0	0	**0**

Para i = 1 y j = 0

Ady$_2$[1][0]=(Ady[**1**][0] && Ady [0][**0**]) ||
(Ady[**1**][1] && Ady[1][**0**]) ||
(Ady[**1**][2] && Ady[2][**0**]) ||

1	1	0
0	**0**	**1**
0	0	0

Para i = 1 y j = 1

Ady$_2$[1][1]=(Ady[**1**][0] && Ady [0][**1**]) ||
(Ady[**1**][1] && Ady[1][**1**]) ||
(Ady[**1**][2] && Ady[2][**1**]) ||

1	**1**	0
0	**0**	**1**
0	**0**	0

Para i = 1 y j = 2

Ady$_2$[1][2]=(Ady[**1**][0] && Ady [0][**2**]) ||
(Ady[**1**][1] && Ady[1][**2**]) ||
(Ady[**1**][2] && Ady[2][**2**]) ||

1	1	**0**
0	**0**	**1**
0	0	**0**

Para i = 2 y j = 0

Ady$_2$[2][0]=(Ady[**2**][0] && Ady [0][**0**]) ||
(Ady[**2**][1] && Ady[1][**0**]) ||
(Ady[**2**][2] && Ady[2][**0**]) ||

1	1	0
0	0	1
0	**0**	**0**

Para i = 2 y j = 1

Ady$_2$[2][1]=(Ady[**2**][0] && Ady [0][**1**]) ||
(Ady[**2**][1] && Ady[1][**1**]) ||
(Ady[**2**][2] && Ady[2][**1**]) ||

1	**1**	0
0	**0**	1
0	**0**	**0**

Para i = 2 y j = 2

Ady$_2$[2][2]=(Ady[**2**][0] && Ady [0][**2**]) ||
(Ady[**2**][1] && Ady[1][**2**]) ||
(Ady[**2**][2] && Ady[2][**2**]) ||

1	1	**0**
0	0	**1**
0	**0**	**0**

Si ahora, se desea obtener la matriz de trayectoria con longitud 3 (*Ady$_3$*), se puede:

1. Efectuar un triple producto booleano, donde:
   Ady$_3$=Ady and Ady and Ady
2. Efectuar un producto Booleano entre la matriz de trayectoria de longitud de 2 y la matriz de adyacencia:
   Ady$_3$=Ady$_2$ and Ady

En forma análoga se puede calcular Ady$_4$ una vez obtenida Ady$_3$ y las restantes Matrices de Trayectoria de longitud 'n'.

Si se desea responder a la pregunta ¿Existe una ruta entre el nodo 'A' y el 'B' con longitud menor o igual a 3? Puede utilizar las matrices de trayectoria como sigue:

Ady[0][1] or Ady$_2$[0][1] or Ady$_3$[0][1]

En caso de dar TRUE la anterior operación lógica, implicaría que entre el nodo 'A' (posición '0' de un arreglo en C/C++) y el nodo 'B' (posición '1' de un arreglo en C/C++), existe una trayectoria menor o igual a tres.

Si desea en cambio saber si existe algun camino entre dos nodos determinados, suponga 'i' y 'j', se debería efectuar la siguiente operación:

Path[i][j] = Ady[i][j] or Ady$_2$[i][j] |or Ady$_3$[i][j]

O

$$Path[i][j] = \sum_{z=1}^{3} Ady_z[i][j]$$

Donde el límite de la sumatoria queda establecido por la cantidad de nodos del grafo (z = 3). Esta matriz Path recibe el nombre de ***"Cierre Transitivo de la matriz de Adyacencia"*** y representa la existencia de caminos o no, entre dos nodos, indistintamente de la longitud del mismo.

Implementación del Cierre Transitivo en C/C++

El programa a continuación, denominado "ctransit.cpp", muestra la implementación en C/C++ del algoritmo básico de cierre transitivo. El programa utiliza una función auxiliar denominada "producto", donde el objetivo de esta última, es efectuar un producto entre una matriz intermedia de longitud "k" (donde 1 <= k <= nodos del grafo) y la matriz de adyacencia, de cara a obtener la matriz de adyacencia de longitud k+1.

Archivo: **"ctransit.cpp"**

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       const int maxnodos=3;
5
6       void cierreTransitivo(int [][maxnodos],int [][maxnodos]);
7       void producto(int [][maxnodos],int [][maxnodos],int [][maxnodos]);
8
9       void main(void)
10      {
11       int ady[3][3]={1, 1, 0,
12                      0, 0, 1,
13                      0, 0, 0};
14       int ct[3][3];
15
16       clrscr();
17       //Muestro la Matriz de Adyacencia
18       cout<<"Matriz de Adyacencia:"<<"\n";
19       for(int i=0;i<maxnodos;i++)
20        {
21         for(int j=0;j<maxnodos;j++)
22           cout<<" "<<ady[i][j]<<" ";
23
24         cout<<endl;
25        }
26       cout<<"\n\nCierre Transitivo para la Matriz de Adyacencia\n";
27       cierreTransitivo(ady,ct);
28       //Muestro el Cierre Transitivo de la Matriz de Adyacencia Ady
29       for(i=0;i<maxnodos;i++)
30        {
31         for(int j=0;j<maxnodos;j++)
32           cout<<" "<<ct[i][j]<<" ";
33
34         cout<<endl;
35        }
36      }
37
38      void producto(int a[][maxnodos],int b[][maxnodos],int c[][maxnodos])
39      {
40       int i,j,k,val;
41
42       for(i=0;i<maxnodos;++i)
43        for(j=0;j<maxnodos;++j)
44         {
45          val=0;
46          for(k=0;k<maxnodos;++k)
47           val=val || (a[i][k] && b[k][j]);
```

```
48
49          c[i][j]=val;
50         }
51     }
52
53     void cierreTransitivo(int ady[][maxnodos],int ct[][maxnodos])
54     {
55      int i,j,k;
56      int nuevoproducto[maxnodos][maxnodos];
57      int adyprod[maxnodos][maxnodos];
58
59      for(i=0; i<maxnodos; ++i)
60       for(j=0;j<maxnodos;++j)
61        adyprod[i][j]=ct[i][j]=ady[i][j];//iguala las matrices
62
63      for(i=1;i<maxnodos;++i)
64      {//Por Cada Nodo, desde 1 a maxnodos, obtengo el cierre transitivo
       multiplicando
65       // en forma sucesiva ady(i-1)*adyacencia
66       producto(adyprod,ady,nuevoproducto);
67
68       for(j=0;j<maxnodos;++j)
69
70        for(k=0;k<maxnodos;++k)
71        {
72          ct[j][k]= ct[j][k] || nuevoproducto[j][k];
73          adyprod[j][k]=nuevoproducto[j][k];
74        }
75      }
76     }
```

Aplicación de la Teoría de Grafos: Determinación de la Ruta Más Corta

La determinación de la ruta más corta es un problema típico de Modelización con grafos. Cada punto axial o nodo puede representar un hito del recorrido y los arcos simularían el trayecto. El Algoritmo que se analizará en el presente apartado para la determinación del camino más corto se denomina 'Algoritmo de Dijkstra' por su creador, Edsger Wybe Dijkstra.

El Algoritmo recibe como parámetros de entrada el punto de partida y el desitno, sobre los cuales se desea determinar la ruta más corta. Inicialmente el algoritmo parte del origen, considerándolo como un nodo eje o central, y calcula sobre los nodos adyacente el peso relativo al nodo central. Una vez que determina los pesos asociados a los caminos de los nodos adyacentes, escoge el arco de menor peso y transita al destino vinculado convirtiéndose éste, en el nuevo nodo central o eje. A partir de allí, los pesos relativos comienzan a acumular los pesos previos calculados y escogen directamente los caminos de menor peso acumulado hasta llegar al destino acumulado.

A los efectos de analizar el algoritmo en detalle, suponga el siguiente grafo:

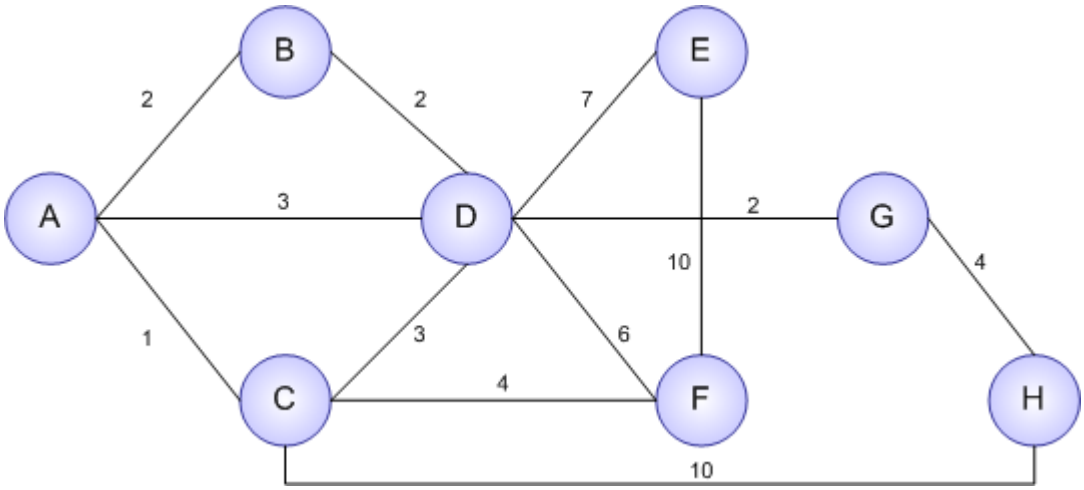


Figura 179: "Grafo a utilizar con el Algoritmo de Dijkstra"

El grafo se compone de ochos nodos con sus correspondientes arcos, el sentido en los arcos es bidireccional por lo que se omiten las flechas de circulación para claridad de la figura. Un aspecto fundamental a comprender del algoritmo del trayecto más corto, es que la magnitud por la que se compara si un camino es más corto que otro, estará en función del peso o costo asociado a la transición entre los nodos que conforman el 'path' o camino. De esta forma el camino más corto entre el origen y destino especificado, no debe entenderse como la menor cantidad de arcos o trayectos necesariamente, sino sobre el trayecto que menos costo requiere para lograr el destino a partir del origen señalado. Esta particularidad, da lugar a que cada transición entre nodos tenga un costo asociado, por lo que en realidad el costo de transición estará vinculado al arco que relaciona dichos nodos, de esta forma en lugar de tener una matriz de asociación de nodos se tiene una matriz de asociación-costo de nodos, ya que ésta úlima establece la relación entre los nodos a través del propio costo de transición.

Note que en la figura anterior los costos de transitar desde un nodo origen a otro destino se indican sobre el arco, indistintamente en qué sentido se produzca la transición. Esta situación, permite plantear la matriz de asociación-costo, en función del grafo de ejemplo anterior, como sigue:

		A	B	C	D	E	F	G	H
		0	1	2	3	4	5	6	7
A	0	I	2	1	3	I	I	I	I
B	1	2	I	I	2	I	I	I	I
C	2	1	I	I	3	I	4	I	10
D	3	3	2	3	I	7	6	2	I
E	4	I	I	I	7	I	10	I	I
F	5	I	I	4	6	10	I	I	I
G	6	I	I	I	2	I	I	I	4
H	7	I	I	10	I	I	I	4	I

Figura 180: “Matriz de Asociación-Costo”

Note que la matriz de asociación-costo representa una generalización de la matriz de asociación original; si en la matriz de asociación-costo toda transición entre nodos tuviese el valor de una unidad, se estaría en presencia de la matriz de asociación original.

Desde	s	0
Hasta	t	7

						Distancia								Perm								Precede							
k	actual	dc	menDist	newDist	i	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
						I	I	I	I	I	I	I	I	NM	NM	NM	NM	NM	NM	NM	NM								
	0													M															
	0	0	I																										
0	0	0	I	I	0	I																							
1	0	0	2	2	1	I	2															0							
2	0	0	1	1	2	I	2	1														0	0						
2	0	0	1	3	3	I	2	1	3													0	0	0					
2	0	0	1	I	4	I	2	1	3													0	0	0					
2	0	0	1	I	5	I	2	1	3													0	0	0					
2	0	0	1	I	6	I	2	1	3													0	0	0					
2	0	0	1	I	7	I	2	1	3													0	0	0					
2	2	1	I			I	2	1	3					M		M													
2	2	1	I		0	I	2	1	3																				
1	2	1	2	1+I	1	I	2	1	3																				
1	2	1	2		2	I	2	1	3							M													
1	2	1	2	1+3=4	3	I	2	1	3	Trunca este trayecto																			
1	2	1	2	1+I	4	I	2	1	3	(Desde A llega con costo 3 y a través de C con costo 4)																			
1	2	1	2	1+4=5	5	I	2	1	3		5											0	0	0		2			
1	2	1	2	1+I	6	I	2	1	3		5											0	0	0		2			
1	2	1	2	1+10=11	7	I	2	1	3		5		11									0	0	0		2			2
1	1	2	I											M	M	M						0	0	0		2			2
1	1	2	I		0	I	2	1	3		5		11	M								0	0	0		2			2
1	1	2	I		1	I	2	1	3		5		11		M							0	0	0		2			2
1	1	2	I		2	I	2	1	3		5		11			M						0	0	0		2			2
3	1	2	3	2+2=4	3	I	2	1	3		5		11									0	0	0		2			2
3	1	2	3	2+I	4	I	2	1	3		5		11									0	0	0		2			2
3	1	2	3	2+I	5	I	2	1	3		5		11									0	0	0		2			2
3	1	2	3	2+I	6	I	2	1	3		5		11									0	0	0		2			2
3	1	2	3	2+I	7	I	2	1	3		5		11									0	0	0		2			2
3	3	3	I											M	M	M	M					0	0	0		2			2
3	3	3	I		0	I	2	1	3		5		11	M								0	0	0		2			2
3	3	3	I		1	I	2	1	3		5		11		M							0	0	0		2			2
3	3	3	I		2	I	2	1	3		5		11			M						0	0	0		2			2
3	3	3	I		3	I	2	1	3		5		11				M					0	0	0		2			2
4	3	3	10	3+7=10	4	I	2	1	3	10	5		11									0	0	0	3	2			2
5	3	3	9	3+6=9	5	I	2	1	3	10	5		11									0	0	0	3	2			2
6	3	3	5	3+2=5	6	I	2	1	3	10	5	5	11									0	0	0	3	2		3	2
6	3	3	5	3+I	7	I	2	1	3	10	5	5	11									0	0	0	3	2	3		2
6	6	5	I			I	2	1	3	10	5	5	11	M	M	M	M			M		0	0	0	3	2	3		2
6	6	5	I		0	I	2	1	3	10	5	5	11	M								0	0	0	3	2	3		2
6	6	5	I		1	I	2	1	3	10	5	5	11		M							0	0	0	3	2	3		2
6	6	5	I		2	I	2	1	3	10	5	5	11			M						0	0	0	3	2	3		2
6	6	5	I		3	I	2	1	3	10	5	5	11				M					0	0	0	3	2	3		2
6	6	5	I	5+I	4	I	2	1	3	10	5	5	11									0	0	0	3	2	3		2
6	6	5	I	5+I	5	I	2	1	3	10	5	5	11									0	0	0	3	2	3		2
6	6	5	I		6	I	2	1	3	10	5	5	11							M		0	0	0	3	2	3		2
7	7	5	9	5+4=9	7	I	2	1	3	10	5	5	9								M	0	0	0	3	2	3		6

Figura 181: “Prueba de Escritorio del Algoritmo de Dijkstra”

El algoritmo de dijkstra inicia el cálculo de trayectos desde el origen indicado (‘s’) y, al mismo tiempo, va determinando el orden de procedencia y relación entre nodos a través de la matriz ‘Precede’.La matriz denominada ‘Distancia’, inicialmente determina el peso de transitar desde el origen hasta los nodos adyacentes y,

posteriormente, calcula el peso acumulado teniendo en consideración la ruta que le precede, dada por 'Precede'.

El Algoritmo recorre desde el punto de giro o nodo central, todos los nodos y analiza si están vinculados concretamente. Para los nodos vinculados calculará y actualizará el peso acumulado y, para aquellos no vinculados, también calculará un peso, pero el hecho de la no vinculación se implementa como un costo muy elevado lo que provoca que el algoritmo descarte automáticamente el análisis de dicho camino y no se considere como parte del trayecto.

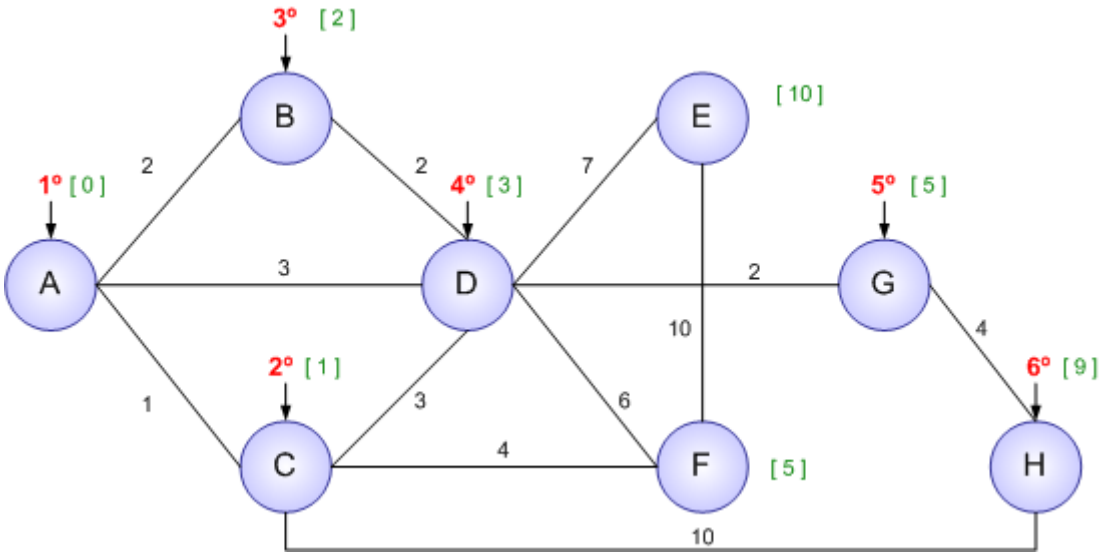


Figura 182: "Recorrido del Algoritmo de Dijkstra para el Grafo de Ejemplo"

Una vez que los pesos son calculados, quien determina en el algoritmo la próxima transición es 'k', 'k' representa el nodo con el menor costo acumulado asociado al nodo eje o central, por lo que la transición se dará a través del arco que vincula al nodo central con el nodo indicado por 'k'. Este mecanismo es repetido en forma sucesiva, hasta que el el 'k' coincide con el nodo destino, a lo que el algoritmo culmina en que en la posición destino en el arreglo distancia se tiene el costo menor total de recorrer el trayecto solicitado y, el recorrido estará dado a través de la matriz unidimensional precede.

Vea que el corte del algoritmo se da cuando k='7' en la prueba de escritoro y, el costo menor total asociado es 9 dado por distancia[7], en donde '7' se corresponde con el código asignado al nodo destino 'H' indicado ('t').

Implementación del Algoritmo de Dijkstra

Archivo: dijkstra.cpp

```
1       #include <iostream.h>
2       #include <conio.h>
3
4       const short I=99;
5       const short maximoNodos=8;
6       const short miembro=1;
7       const short noMiembro=0;
8
9       void caminoMasCorto(short[][maximoNodos],int,int);
10
11      void main(void)
12      {
13       short pesos[8][8]={ {I,2,1,3,I,I,I,I},
14                           {2,I,I,2,I,I,I,I},
15                           {1,I,I,3,I,4,I,10},
16                           {3,2,3,I,7,6,2,I},
17                           {I,I,I,7,I,10,I,I},
18                           {I,I,4,6,10,I,I,I},
19                           {I,I,I,2,I,I,I,4},
20                           {I,I,10,I,I,I,4,I},};
21
22       caminoMasCorto(pesos,0,7);
23      }
24
25      void caminoMasCorto(short pesos[][maximoNodos],int s,int t)
26      {//Implementaci¢n del Algoritmo de Dijkstra
27       short precede[maximoNodos];
28       short distancia[maximoNodos];
29       short perm[maximoNodos];
30
31       int actual,i,k,dc;
32       i=0;//recorre los nodos adyacentes al de giro
33       k=0;//almacena la posicion del nodo con menor peso relativo
34       dc=0;//acumula el peso hasta la posicion actual
35       actual=s;//se¤ala cual es el nodo de giro actual
36
37       int menDist,newDist;
38       menDist=0;//almacena el menor peso relativo
39       newDist=0; //almacena el peso acumulado, mas el de la transicion al
40               //nodo adyacente relativo al punto de giro actual
41
42       //Inicializo los Arreglos
43       for(i=0;i<maximoNodos;i++)
44       {
```

```
45          perm[i]=noMiembro;
46          distancia[i]=I;
47          precede[i]=-5;
48         }
49
50         perm[s]=miembro;//Desde el nodo 'desde' inicia Dijkstra
51         distancia[s]=0;//Cero simboliza peso '0' para punto el punto de partida
52
53         while(actual!=t)
54         {
55           menDist=I;
56           dc=distancia[actual];
57
58           //El Nodo de Giro es el Se¤alado por 'actual'
59           //Con el for se recorren los nodos adyacentes al de giro
60           //calculando los pesos asociados para determinar la
61           //proxima ruta
62           for(i=0;i<maximoNodos;i++)
63           {
64            if(perm[i]==noMiembro)
65            {
66             newDist=dc+pesos[actual][i];
67
68             if(newDist<distancia[i])
69             {
70              distancia[i]=newDist;
71              precede[i]=actual;
72             }
73
74             if(distancia[i]<menDist && distancia[i]!=0)
75             {
76              menDist=distancia[i];
77              k=i;
78             }
79            }//si no miembro
80           }//fin recorrido de nodos adyacentes
81
82           actual=k;
83           perm[actual]=miembro;
84         }
85
86         clrscr();
87
88         cout<<"\nTrayectoria: "<<t<<"->";
89
90         int pos=precede[t];
91
92         while(pos!=s)//el -5 indica el origen, seg£n la precedencia
93         {
```

```
94          cout<<pos<<"->";
95
96          pos=precede[pos];
97        }
98
99       cout<<s;
100      cout<<"\nCosto: "<<distancia[t];
101     }
```

Referencias

- "Algoritmos en C++"; Sedgewick; Editorial Assison-Wesley/Dias de Santos; EE.UU, 1995.

- "Redes de Computadoras", Tercera Edición; Andrew Tanenbaum; Editorial Prentice Hall; México 1997.

- "Concepts of Exact QoS Routing Algorithms"; Piet Van Mieghem y Fernando Kuipers; IEEE/ACM Transactions on Networking, Volumen 12, Edición 15, Octubre de 2004.

- "Estructuras de Datos con C y C++"; Langsam, Aungenstein y Tenenbaum; Editorial Prentice Hall; México, 1997. Capítulo 3: "Recursión".

View publication stats