14
24

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CONTADURIA Y ADMINISTRACION

ANALISIS DE EFICIENCIA DE ALGORITMOS DE ORDENAMIENTO

SEMINARIO DE INVESTIGACION INFORMATICA

QUE EN OPCION AL GRADO DE

LICENCIADO EN INFORMATICA

P R E S E N T A N:

MARTHA PATRICIA RICO MEDRANO

MAX OCHOA MORALES

ASESOR: LIC. EN ADMINISTRACION DE EMPRESAS

MARIO NOVOA GAMAS

MEXICO, D. F. 1992

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE

INTRODUCCION

INTRODUCCION

El motivo para llevar a cabo la presente tesis, consiste en introducir al estudiante de los primeros semestres de las carreras afines al área de informática, al análisis del comportamiento de los algoritmos de ordenamiento, de tal manera que:

1) le permita identificar las situaciones en las cuales un determinado algoritmo es susceptible de aplicación.

2) bajo que condiciones un algoritmo opera y/o mantiene su eficiencia para resolver un problema, entendiéndose como eficiencia, resolver un problema de la mejor manera posible.

Dentro de toda la gama de algoritmos existente, la selección de los algoritmos de ordenamiento para desarrollar este trabajo se realizó pretendiendo facilitar al informático la manipulación de información, debido a que para el manejo y recuperación de la misma, resulta conveniente poder accesarla en una forma ordenada. Lo anterior refleja la interconexión que existe entre los algoritmos de ordenamiento y los de búsqueda.

La totalidad de los algoritmos de ordenamiento pueden dividirse en:

- externos, si es que no se realizan en la memoria principal de la computadora, y en
- internos, si es que se efectúan en la memoria principal

Los que tratamos en esta tesis están dentro de los algoritmos de ordenamiento internos, y son los siguientes:

1) Selección Directa.
2) QuickSort.
3) Árbol Binario.
4) Con Cálculo de dirección.

siendo el objetivo realizar un análisis acerca del comportamiento de estos algoritmos de ordenamiento, comparando la velocidad de crecimiento del tiempo de ejecución de n elementos, tanto en el peor caso, es decir el máximo valor del tiempo de ejecución para entradas de tamaño n, como en el caso promedio, el cual es el valor medio del tiempo de ejecución de todas las entradas de tamaño n, considerando como medida fundamental la complejidad de tiempo asintótica, y englobando a los siguientes factores en una constante de proporcionalidad:

- calidad del código generado por el compilador utilizado para crear el programa objeto,

- naturaleza y rapidez de las instrucciones empleadas,

- forma de implantación en el programa (estructura de datos, rutinas empleadas, etc.).

Los cuatro algoritmos mencionados fueron seleccionados debido a que el tiempo de ejecución, el cual es nuestra medida fundamental en cuanto a análisis de eficiencia se refiere, va mejorando de un algoritmo a otro, dependiendo en algunos casos del tamaño de la entrada y de las características de ésta.

En estos métodos esta reflejado el uso de diferentes conceptos en materia de estructuras de datos, manejo de memoria, y técnicas de programación en la realización de los programas fuente que resuelven el problema de ordenamiento, con la finalidad de aterrizar sistematizadamente la aplicación de un algoritmo específico a un problema determinado, contribuyendo a mejorar el tiempo de ejecución requerido para resolver el problema.

El problema del ordenamiento durante el desarrollo de la tesis consiste en ordenar una secuencia de elementos de tal forma que sus valores, formen una secuencia creciente. Esto es, dados los elementos r_1, r_2,..., r_n, con valores k_1, k_2,..., k_n, respectivamente debe resultar la misma

secuencia de elementos en orden r_{i1}, r_{i2}, ..., r_{in}, tal que $k_{i1} <= k_{i2}, ..., <= k_{in}$. No es necesario que todos los elementos tengan valores distintos.

En cuanto a los apartados que forman la estructura de los capítulos de la presente tesis, existe un capítulo introductorio que tiene la finalidad de aclarar los conceptos fundamentales bajo los cuales se desarrolla el material expuesto acerca de cada uno de los algoritmos de ordenamiento seleccionados, y por cada algoritmo se desarrolla un capítulo.

Los aspectos fundamentales que se tratan por cada algoritmo son los siguientes:

1. Descripción del método.

2. Modelo matemático.
   2.1 Descripción del comportamiento.

3. Algoritmo.

4. Programa fuente de la primera versión.
   4.1 Análisis del programa.

5. Programa fuente de la segunda versión.

6. Comparación entre las dos versiones.

7. Análisis de eficiencia del algoritmo.

8. Determinación de las condiciones en las que el algoritmo presenta un comportamiento eficiente.

A continuación se describe en que consiste cada uno de los apartados mencionados.

1. Descripción del método.

Este apartado tiene como finalidad exponer en que consiste el algoritmo, haciendo uso tanto de definiciones como de gráficas y ejemplos, con la

intención de hacerlo más explícito.

2. Modelo Matemático.

Tiene como objetivo el concretizar en una representación gráfica la forma en que se comporta el algoritmo de que se trate, de tal manera que se pueda identificar cual es el proceso por medio del cual se resolverá el problema de ordenamiento utilizando dicho algoritmo, lo cual se logra mediante un modelo matemático denominado grafo.

2.1 Descripción del comportamiento.

Explica con una narrativa, la abstracción del comportamiento del algoritmo en cuestión (que representa el grafo), relacionando el tratamiento y secuencia de los objetos (variables) utilizados en el modelo matemático con su descripción.

3. Algoritmo.

Es el nexo entre la abstracción para resolver el problema (modelo matemático) y el programa sistematizado, capaz de ser ejecutado en una computadora. Se realiza de tal manera que permita identificar la relación que guarda el modelo matemático con la secuencia de instrucciones que constituirán el programa fuente que resuelva en computadora el problema de ordenamiento.

4. Programa fuente de la primera versión.

Tiene como objetivo presentar el código fuente de la forma en que se resuelve el problema en cuestión, utilizando las estructuras de datos mínimas suficientes para resolver el problema, aún cuando haya posibilidades de mejorar el tiempo de ejecución utilizando por ejemplo diferentes estructuras de datos.

4.1 Análisis del programa.

El análisis de comportamiento del programa fuente de la primera versión consiste en el desglose de la forma en que se resolvió el problema, en el

que se identifiquen:

a) Estructuras de datos utilizadas.
b) Consecuencias del uso de las estructuras de datos usadas
c) Congruencia entre el esquema de solución presentado en el programa (estilo de programación, rutinas utilizadas, estructuras de datos, etc.) y la naturaleza del algoritmo en cuestión.

5. Programa fuente de la segunda versión.

Consiste en presentar el código fuente utilizado para resolver el problema, subsanando las deficiencias identificadas después de realizar el análisis del comportamiento de la primera versión. En esta segunda versión se presentará un esquema de programación mejorado, en el cual la combinación de estructuras de datos utilizadas, manejo de rutinas, etc., contribuirá a mejorar la eficiencia del algoritmo bajo un nuevo esquema de implantación.

6. Comparación entre las dos versiones.

Consiste en contraponer las ventajas y desventajas de las dos versiones presentadas, determinando cual y en que situación es más apropiada.

7. Análisis de eficiencia del algoritmo.

Este análisis se refiere a la presentación de la función asintótica de tiempo para el algoritmo en cuestión, explicando su comportamiento tanto en el peor caso como en el caso promedio. Para ello, se hace uso de ejemplos y gráficas que ayuden a hacer más clara la exposición del comportamiento asintótico de la función de tiempo de ejecución, el cual se refiere al número de operaciones que requiere el algoritmo para resolver el problema de ordenamiento cuando el tamaño de la entrada es suficientemente grande.

La razón principal por la que este tipo de análisis (asintótico) es importante, reside en que si el tamaño de la entrada es pequeño,

generalmente cualquier algoritmo será útil para resolver el problema, pero si el tamaño de la entrada aumenta considerablemente, es entonces cuando un algoritmo puede ser ventajoso en comparación con otro para resolver el mismo problema.

8. Determinación de las condiciones en las que el algoritmo opera eficientemente.

Tiene como finalidad aportar las consideraciones necesarias para poder utilizar de la mejor manera el algoritmo, dependiendo de la situación en que se encuentre el problema; es decir, si se conoce o no la distribución de la entrada, el tamaño de la misma, disponibilidad de espacio, frecuencia de uso del algoritmo, etc.

En cuanto a las condiciones de hardware requeridas para la ejecución de los programas expuestos en la presente tesis, existen las siguientes consideraciones:

- microcomputadora XT con procesador 8088.
- memoria principal de 512 kb.
- drive de 5 1/4" (360 Kb) o de 3 1/2" (720 Kb).
- sistema operativo MS DOS versión 3.2.

Por lo que se refiere al software, la programación del código fuente de los algoritmos se realizó en ANSI C, de acuerdo al estándar X3.159-1989 del comité X3J11.

En lo que respecta a las notas bibliográficas, se asignó una clave a cada una de las obras que se consultaron, formada por las tres primeras letras del apellido paterno del primer autor del libro, seguido del año de publicación de la obra.

De esta forma, si el autor es Pedro Jiménez López y el año de la obra es 1989, la clave sería JIM89. Si se tiene más de una obra del mismo año del mismo autor, se le agregaría una letra minúscula empezando por la letra a. Si tuviéramos tres obras del autor anterior, las claves serían JIM89a, JIM89b y JIM89c.

Las claves asignadas a cada libro se encuentran en la bibliografía al final de la tesis. Cuando se indica una cita bibliográfica, se indica la clave del libro y la página de dónde se tomo encerradas entre corchetes; por ejemplo [JIM89, pág. 180].

CAPITULO I

CONCEPTOS GENERALES

CONCEPTOS GENERALES

En este capítulo, se presentarán los conceptos que se manejarán durante el desarrollo de la tesis, con el fin de uniformar criterios.

El tema medular en cada uno de los siguientes capítulos es el análisis de la eficiencia de los algoritmos de ordenamiento, conociendo el comportamiento asintótico de la función tiempo, o lo que es lo mismo, de que orden es el número de operaciones que requiere el algoritmo para resolver el problema de ordenamiento, cuando el tamaño de la entrada (n) es suficientemente grande:

> "...Determinar qué es una entrada grande o pequeña depende de los tiempos de ejecución exactos de los algoritmos implicados". [AHO88, pág. 21]

Lo anterior se debe a que si un algoritmo se aplicará sólo con entradas pequeñas, la velocidad de crecimiento del tiempo de ejecución puede ser menos importante que el factor constante de la fórmula del mismo.

Los criterios bajo los cuales se evalúa el comportamiento del tiempo de ejecución son los siguientes:

1) Se definirá el tiempo de ejecución $T_{(n)}$, como el tiempo de ejecución en el peor caso, es decir el máximo valor de tiempo de ejecución para entradas de tamaño **n**, debido a que para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica, es decir de la distribución de la misma, y no sólo del tamaño de ella.

   Sin embargo, como en ocasiones el peor caso no se presenta siempre con la misma frecuencia (a veces a menudo y a veces casi nunca), tiene interés la revisión del tiempo de ejecución en el caso promedio (probabilístico).

   El estudio del caso promedio entra dentro del campo de la

probabilidad y estadística, y siendo de gran interés en la evaluación de los algoritmos, se tomarán como válidas la existencia de ciertas relaciones dadas por la naturaleza del algoritmo de que se trate, tales como número de operaciones básicas requeridas, número de comparaciones, etc., aún cuando el presente trabajo no especificará a detalle la explicación estadística para llegar a dichas relaciones.

Una segunda situación en la cual conviene desviarse de la noción de eficiencia en el peor caso ocurre cuando se conoce la distribución esperada de las entradas a un algoritmo.

En dichas situaciones, el análisis del caso promedio puede ser mucho más significativo que el análisis del peor caso. Además, existen algoritmos cuyo tiempo de ejecución, es menor en el caso promedio que en el peor caso, como el algoritmo de QuickSort.

2) Para hacer referencia a la velocidad de crecimiento de los valores de una función de tiempo se usará la notación conocida como notación asintótica ("o mayúscula"), la cual se define de la siguiente forma:

> "...decir que el tiempo de ejecución T_n de un programa es $O(n^2)$, que se lee "o mayúscula de n al cuadrado", o tan sólo "o de n al cuadrado", significa que existen constantes enteras positivas c y n_0 tales que para n mayor o igual a n_0, se tiene que $T_n <= cn^2$". [AHO88, pág. 18]

A continuación se supondrá que todas las funciones de tiempo están definidas en los enteros no negativos, y que sus valores son siempre no negativos pero no necesariamente enteros.

3) El termino eficiencia será una función de la complejidad asintótica de tiempo por una constante de proporcionalidad (que depende del compilador, el programa fuente, entre otros factores).

Una razón para tratar con la complejidad asintótica de tiempo e ignorar factores constantes se refiere a que es la complejidad asintótica de tiempo lo que determina para que tamaño de entradas puede usarse el algoritmo en la solución de un problema específico.

4) La función de complejidad asintótica de tiempo $T(n)$, no se expresará en unidades estándares de tiempo como segundos. Este hecho implica que sólo pueden hacerse observaciones como "el tiempo de ejecución de tal algoritmo es proporcional a n^2", sin especificar la constante de proporcionalidad.

Cada algoritmo tendrá dos versiones con la finalidad de poder comparar el comportamiento del mismo, cuando se mejora el factor constante, es decir la constante de proporcionalidad.

El criterio en el cual se basa la creación de las diferentes versiones de un mismo algoritmo es el canjear tiempo por espacio. La decisión de intercambiar tiempo por espacio debe ir de acuerdo a las condiciones particulares en las que se encuentre el problema, considerando tanto la disponibilidad de tiempo como de espacio.

Los ordenamientos estudiados en cuanto a el valor de su función asintótica de tiempo, están en el rango de $O(n\log n)$ hasta $O(n^2)$. La base del logaritmo es irrelevante para determinar el orden del método de ordenamiento. Respecto a esto, se aclara que a medida que el tamaño de la entrada aumenta, la velocidad de crecimiento del tiempo de ejecución también aumenta y viceversa, pasando de un comportamiento logarítmico $O(n\log n)$ a uno exponencial $O(n^2)$.

El desarrollo del análisis objeto de este trabajo se realiza deductivamente, partiendo de una visión general del algoritmo, hasta su desglose matemático. Es por ello que para representar el comportamiento del algoritmo en una forma integral, se hace uso de una representación gráfica que lo haga explícito, denominada grafo.

El grafo es un modelo matemático que representa esquemáticamente una situación real, en este caso la manera de resolver el problema de ordenamiento utilizando un algoritmo específico. Ejemplos de grafos son: un mapa de carreteras, un plano de la red del metro de una ciudad,

un plano de un circuito eléctrico, un plano de la red telefónica de una compañía, etc.

Existe una clase particular de grafos, a la que se denomina árboles, los cuales juegan un papel importante tanto en el estudio de algoritmos como en el estudio de estructuras de datos útiles para almacenar información que manipula un algoritmo. Se describen más adelante.

Una vez que se ha podido visualizar gráficamente el comportamiento del método de ordenamiento de que se trate, llámese este **Selección Directa, Quicksort, Arbol Binario, y/o Con Cálculo de Dirección,** resulta viable su transformación a un algoritmo, que es

> "...una secuencia finita de instrucciones, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo finito." [AHO88, pág. 2].

El algoritmo nos aporta una forma menos abstracta que el modelo matemático, acerca de la manera de resolver el problema de ordenamiento utilizando un método en particular.

El algoritmo siendo susceptible de sistematizarse funge como intermediario entre el modelo matemático y el programa para ser ejecutado en una computadora. Desde este momento se esta en posibilidad de hacer uso de estructuras de información, a lo cual nos referimos como sinónimo del concepto de estructura de datos.

Las estructuras de datos consisten en conjuntos de variables, quizá de tipos distintos, conectadas entre sí de diversas formas. El uso de ellas es necesario para hacer más clara y objetiva la representación de la solución del problema de ordenamiento. Estas estructuras están plasmadas en los programas fuente, tratando de ejemplificar en cada uno de ellos el comportamiento del algoritmo con elementos al azar, con la finalidad de eliminar la condición determinística en el desarrollo de la solución del problema. Existen dos versiones de los programas fuente, siendo el criterio para la creación de éstas, el permitir la disminución de la velocidad de crecimiento del tiempo de ejecución cuando se mejora el factor constante.
En la creación de las dos diferentes versiones de un algoritmo dado, la

mejoría de la segunda versión en relación a la primera, mantendrá una relación inversamente proporcional en tiempo y espacio: es decir, a mayor tiempo menor espacio y viceversa; de ahí que las diferentes versiones sacrifiquen un parámetro en virtud de mejorar el otro.

Para ello, en la segunda versión se emplean apuntadores a los elementos a ordenar, con lo cual es posible mover sólo los apuntadores y no los elementos, lográndose un ahorro en el tiempo de ejecución, sobre todo si se considera que el tamaño de los elementos puede afectar perjudicialmente al tiempo de ejecución de un programa. Sin embargo, con el ahorro de tiempo de ejecución va implícito un aumento en el espacio requerido por el algoritmo, ya que no sólo se necesita espacio para los elementos a ordenar, sino también un espacio adicional para el manejo de los apuntadores. La mejoría del factor constante es relativa pues depende de factores como disponibilidad de tiempo, espacio, etc.

En la programación de los algoritmos de ordenamiento se implantó una rutina en la que se genera una semilla (conjunto de valores de entrada al algoritmo) distinta cada vez que se ejecuta éste, con el fin de observar su comportamiento con conjuntos diferentes de números a los cuales se les practica el ordenamiento.

Para lograr una mejor comprensión de cada uno de los algoritmos seleccionados, es necesario entender los conceptos bajo los cuales se desarrolla cada uno de ellos. Como es sabido, todos los algoritmos desarrollados en esta tesis se refieren a soluciones de problemas de ordenamiento. A pesar de ello, dos algoritmos requieren atención especial, por manejar conceptos nuevos para el estudiante al que esta dirigido este material. Ellos son **Arbol Binario** y **Con Cálculo de Dirección.**

Respecto al algoritmo de **Arbol Binario,** se tiene que:

"Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación (de <<paternidad>>) que impone una estructura jerárquica sobre los nodos.....Formalmente, un árbol se puede definir de manera recursiva como sigue:

1. Un solo nodo es, por sí mismo un árbol. Ese nodo es también la

raíz de dicho árbol.

2. Supóngase que n es un nodo, y que A_1, A_2,... A_k son árboles con raíces n_1, n_2,..., n_k, respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos n_1, n_2,..., n_k. En dicho árbol, n es la raíz y A_1, A_2,..., A_k son los subárboles de la raíz. Los nodos n_1, n_2,..., n_k, reciben el nombre de hijos del nodo n." [AHO88, pág. 76].

La estructura jerárquica determinada por el árbol, nos permite identificar diferentes papeles en cuanto a los nodos, ya que hay antecesores, padres, hojas, etc.

Un antecesor o un descendiente de un nodo que no sea él mismo recibe el nombre de antecesor propio o descendiente propio, respectivamente. En un árbol, la raíz es el único nodo que no tiene antecesores propios. Un nodo sin descendientes propios se denomina hoja. Un subárbol de un árbol es un nodo junto con todos sus descendientes.

Asimismo, la línea jerárquica en un árbol, conecta a sus nodos, esto es:

> Si n_1, n_2,..., n_k, es una sucesión de nodos de un árbol tal que n_i es el padre de $n_i + 1$ para $1 <= i < k$, entonces la secuencia se denomina camino del nodo n_i al nodo n_k. La longitud de un camino es igual al número de nodos del camino menos 1. Por lo tanto, hay un camino de longitud cero de cualquier nodo a sí mismo.

Es interesante observar que si existe un camino de un nodo *a* hacia otro *b*, entonces *a* es un antecesor de *b*, y *b* es un descendiente de *a*.

A los árboles explicados anteriormente se les denomina ordenados orientados, porque los hijos de cada nodo están ordenados de izquierda a derecha, y existe un camino orientado (un camino con una dirección particular) de cada nodo a sus descendientes.

Otro concepto de árbol es el de **"Arbol Binario"**, que define un árbol vacío o un árbol en el que cada nodo no tiene hijos, tiene un hijo izquierdo, un hijo derecho o un hijo izquierdo y un hijo derecho. El hecho de que cada hijo se designe en este caso como hijo izquierdo o como

hijo derecho hace que un "**Arbol Binario**" sea distinto de un árbol ordenado orientado.

Un "**Árbol Binario**" es una estructura de datos muy útil cuando se tienen que hacer decisiones de dos caminos en cada punto en un proceso, como es el caso de ordenamientos y búsquedas; por tal motivo es el tipo de árbol utilizado en la presente tesis.

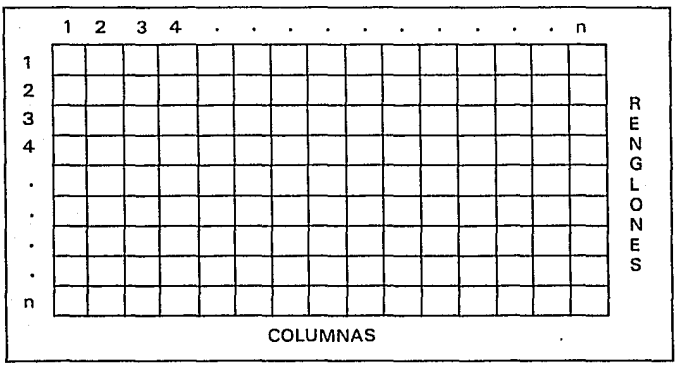
En lo que respecta al ordenamiento **Con Cálculo de Dirección o Hashing**, se basa en la aplicación de una función que dé como resultado la posición definitiva del elemento de entrada al cual se le aplica. Esta función hash, en términos ideales, colocaría a cada elemento de nuestro conjunto inicial a ordenar en una posición distinta, es decir para cada elemento de entrada habría una dirección, implicando una relación unívoca. Sin embargo, cabe destacar que "...ninguna función hash es perfecta,..." [SED90, pág. 231], en la medida que no se dominen indefinidamente las características del conjunto a ordenar, por lo que no se puede realizar una función que no genere direcciones o posiciones repetidas para más de un elemento de entrada, para todas las características posibles del conjunto a ordenar.

Por lo tanto, si ocurre que dos o más elementos del conjunto a ordenar, son direccionados por la función hash a la misma posición, entonces tiene lugar una colisión, la cual tiene que ser solucionada para poder lograr el resultado de ordenamiento.

El método de solución de colisiones puede ser implementado por cualquier método de ordenamiento. Esto quiere decir que una vez que la aplicación de la función hash ha dado la posición del elemento de entrada a ordenar, si la posición ya esta ocupada entonces para solucionar esta colisión, puede aplicarse cualquier método de ordenamiento. Dadas estas circunstancias, al programar el algoritmo de ordenamiento **Con Cálculo de Dirección**, se debe contemplar el uso de estructuras de datos que permitan el ordenamiento del máximo número de colisiones. En la presente tesis, para este caso son de gran ayuda las estructuras de datos con una configuración matricial, cuyas dimensiones dependerán del tamaño del conjunto de elementos a ordenar (n).

Esta configuración matricial, permite que si la aplicación de la función hash genera una dirección ocupada por otro elemento, se pueda aplicar

cualquier algoritmo de ordenamiento para solucionar la colisión, teniendo espacio disponible para esta operación, el esquema que se utiliza para representar la estructura de datos matricial es el siguiente:



donde los renglones y columnas van desde 1 hasta n, el número total de elementos a ordenar, dando con esto la posibilidad de que si la función hash genera la posición (n-1, 1), por ejemplo, y ésta ya ha sido ocupada, entonces la colisión puede resolverse utilizando la primera posición desocupada, con lo cual se mantendría fija la posición (n-1) del renglón y sólo se variaría la segunda coordenada correspondiente a la columna. Posteriormente, se puede aplicar cualquier método de clasificación, que en nuestro caso particular es el de **Selección Directa**, para producir el resultado del algoritmo de ordenamiento **Con Cálculo de Dirección.**

Como ya se mencionó, la médula del algoritmo **Con Cálculo de Dirección**, es la aplicación de una función a los elementos del conjunto a ordenar para determinar su posición y en caso de colisión, la aplicación de algún método de ordenamiento para producir el conjunto de elementos ordenado; sin embargo a diferencia de los otros algoritmos de ordenamiento tratados en esta tesis, **Hashing** no cuenta con una técnica universal a seguir para resolver el problema de ordenamiento, debido a

que depende de la aplicación de una función particular para cada tipo de problema a resolver, puesto que no existe una función hash de aplicación general para todos los problemas de ordenamiento.

Una función hash para una aplicación específica puede producir un número de colisiones reducido, pero si esta se transporta a otro problema de ordenamiento el número de colisiones se puede incrementar tanto, que de un método de ordenamiento eficiente se puede pasar a uno muy deficiente. Lo anterior implica que se considera que una función hash es conveniente en la medida que provoque el menor número de colisiones.

El algoritmo de ordenamiento **Con Cálculo de Dirección** "...es un buen ejemplo de intercambio de tiempo por espacio." [SED90, pág. 231], utilizado en el análisis del tiempo de ejecución tratado en esta tesis, debido a que depende de la selección de la función hash aplicable al conjunto de elementos a ordenar y el método utilizado en la resolución de las colisiones generadas por la función, el ahorro o gasto de tiempo o espacio necesario en la solución del problema de ordenamiento.

CAPITULO II

SELECCION DIRECTA

2.1 DESCRIPCION.

El ordenamiento de **Selección Directa** es aquel en el cual los elementos sucesivos de un conjunto a ordenar, se seleccionan y se colocan en su posición apropiada: el elemento más pequeño es colocado en la primera posición, el siguiente más grande se coloca en la posición contigua y así sucesivamente.

Es decir, si se pretende ordenar un conjunto de n elementos, éste se recorre n - 1 veces, y en cada recorrido se selecciona el elemento menor y se coloca en su posición final dentro del conjunto de elementos, de tal forma que en el recorrido n - 1 se tenga ordenado el conjunto inicial de elementos.

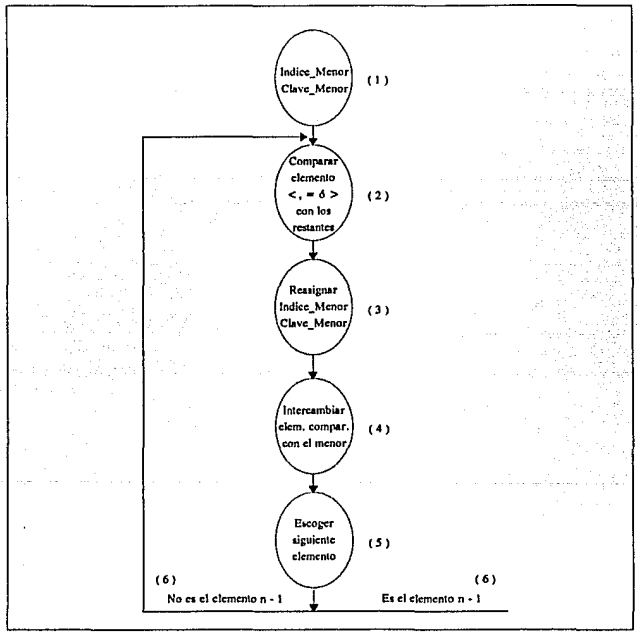
En la siguiente figura se muestran los recorridos del algoritmo de **Selección Directa.**

9	2	2	2	2
2	9	3	3	3
4	4	9	4	4
3	3	4	9	6
6	6	6	6	9
ESTADO INICIAL	DESPUES DE i = 1	DESPUES DE i = 2	DESPUES DE i = 3	DESPUES DE i = 4

Por ejemplo en el recorrido 1, el elemento menor seleccionado en orden ascendente es dos, el cual es intercambiado con nueve en la primera posición. Las líneas de la figura anterior indican el punto sobre el cual se sabe que los elementos menores aparecen ordenados. Después de n - 1 recorridos, el elemento n (el número nueve), está también en el lugar correcto, ya que es el elemento que se sabe que no está entre los n - 1 más pequeños.

2.2 MODELO MATEMATICO.

El grafo que describe el comportamiento mencionado es el siguiente:



En donde:

Clave_menor = variable que contiene el valor más pequeño de la

estructura de la información en una Posición determinada.

Indice_menor = posición en la que se encuentra la variable clave_menor.

2.2.1 Descripción del comportamiento.

1.- Escoger el primer elemento y determinar su posición y valor, dando la posibilidad de poder comparar el valor de este elemento (clave_menor), con el que corresponda a los elementos restantes y así poder detectar la existencia de algún valor más pequeño, lo que llevaría a la reorganización del conjunto de elementos a ordenar, reasignando las variables tanto de Índice_menor y clave_menor, respectivamente.

2.- Comparar desde el siguiente elemento hasta el último, de la estructura de información a ordenar, para poder identificar la existencia de un valor más pequeño que el determinado en el paso anterior.

3.- Si en el segundo paso se encontró un valor más pequeño que el actual de la variable de clave_menor, entonces se reasignan las variables tanto de clave_menor como de Índice_menor con los nuevos valores encontrados en el segundo paso.

4.- Intercambiar el elemento direccionado por Índice_menor por el elemento original en función del cual se realizó la comparación de su valor, con lo que se esta colocando el elemento menor encontrado en la posición correspondiente al más pequeño del conjunto de elementos que se compararon, dejándolo en su posición final.

5.- Escoger el siguiente elemento de la estructura de información a ordenar.

6.- Si el elemento escogido en el paso No. 5 no es el n - 1, entonces se debe repetir desde el paso No. 2 hasta el paso No. 6; en caso contrario se terminó el problema.

2.3 ALGORITMO.

El ordenamiento por **Selección Directa** puede describirse de la siguiente manera:

Desde i = 1 hasta n-1

Seleccionar el elemento más pequeño entre la posición i y la posición n, e intercambiarlo con el elemento de la posición i.

Un algoritmo más detallado se presenta a continuación:

Desde i = 1 hasta n-1
{ Elegir el menor elemento entre i y n - 1 }

índice_menor = i
clave_menor = Contenido de la estructura de información en la posición i

Desde j = i + 1 hasta n

Si el contenido de la estructura de información en la posición j < clave_menor, entonces:

clave_menor = Contenido de la estructura de información en la posición j.
índice_menor = j

Después de comparar el contenido de la estructura de información en la posición j con clave_menor, si índice_menor es distinto de i, hacer:
{ Intercambiar el contenido de la estructura de la

información en la posición i con el contenido de la posición j }

Temp	=	Contenido de la estructura de información en la posición i
Estructura de la información en la posición i	=	Contenido de la estructura de la información en la posición j
Estructura de la información en la posición j	=	Temp

2.4 PROGRAMA FUENTE DE LA PRIMERA VERSION.

A continuación se muestra la primera versión del algoritmo de **Selección Directa.**

```
/* Programa Selección Directa, que realiza un ordenamiento de
   elementos utilizando dicho método */

# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10  /* Máximo número de elementos en el arreglo */
# define POS_INI 0   /* Posición del primer elemento del arreglo */


main()
{
 int i;                      /* Indice para recorrer el arreglo */
 int elementos[NUM_ELE];     /* Arreglo de los elementos a
                                ordenar */
```

```
time_t semilla;                    /* Variable para generar diferentes
                                      series de números aleatorios */
void seleccion_directa();          /* Función para ordenar los elementos
                                      de acuerdo a dicho método */

semilla = time(NULL); /* Se le asigna el tiempo actual del
                         sistema en segundos */
if (&semilla == NULL) /* Si no se pudo obtener el tiempo del
                         sistema, salir */
  printf("\n\nError al generar los números aleatorios, intentar de
        nuevo\n");
else
 {
  srand(semilla);
  printf("\n\nEl arreglo a ordenar es:\n");
  for (i = POS_INI; i < NUM_ELE; i++)
    printf("%d ", elementos[i] = rand()); /* Se selecciona un
                                             entero de forma
                                             aleatoria */
  seleccion_directa(elementos);
  printf("\n\nEl arreglo ordenado es:\n");
  for (i = POS_INI; i < NUM_ELE; i++)
    printf("%d ", elementos[i]);
  printf("\n");
 }
}


void seleccion_directa(elem)

 int elem[];          /* Arreglo de los elementos a ordenar */
{
 int clave_menor; /* Variable para almacenar la posición
                     del menor elemento encontrado hasta
                     el momento */
 int temp;          /* Variable para realizar intercambio de
                       elementos*/
 int i, ,j;         /* Indices para recorrer el arreglo */

 for (i = POS_INI; i < NUM_ELE - 1; i++) /* Se recorre el arreglo
```

```
                                        de la primera posición
                                        a la penúltima */
{
 clave_menor = i; /* Se toma como elemento menor al que apunta i
                     actualmente, ya que se compara con todos los
                     que estén de la posición i + 1 al final */
 for (j = i + 1; j < NUM_ELE; j++) /* Se recorre el arreglo desde
                                      la siguiente posición de i
                                      al final */
   if (elem[j] < elem[clave_menor])
     clave_menor = j;
   if (clave_menor != i) /* Si se encontró un elemento más pequeño,
                            intercambiarlos */
 {
   temp = elem[i];
   elem[i] = elem[clave_menor];
   elem[clave_menor] = temp;
  }
 }
}
```

2.4.1 Análisis del programa

En la función de *selección_directa*, se recorre el arreglo de la primera posición a la penúltima, y se compara el elemento en la posición actual con cada uno de los elementos restantes, determinándose cual es el más pequeño, para colocarlo en la posición actual.

La condición *if (Clave_menor != i)*, nos indica si se encontró un elemento más pequeño que el que él que se encuentra en la posición actual del arreglo, en cuyo caso se realiza el intercambio, y en caso contrario, no hacer nada y continuar con la siguiente posición en el arreglo.

Cabe señalar que en el lenguaje C la primera posición de un arreglo es 0, y no 1 como se ejemplifica en el algoritmo.

Para mostrar el funcionamiento del programa con diferentes conjuntos de números, se utiliza una semilla diferente con la función rand, la cual

es la variable semilla.

A dicha variable se le asigna el tiempo actual del sistema en segundos, dando siempre un número diferente con el cual generar la serie de números aleatorios.

2.5 PROGRAMA FUENTE DE LA SEGUNDA VERSION.

Las diferencias con la versión anterior se muestran en negrillas.

```
/* Programa Selección Directa, que realiza un ordenamiento de
   elementos utilizando dicho método (segunda versión con
   apuntadores) */

# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10  /* Máximo número de elementos en el arreglo */
# define POS_INI 0   /* Posición del primer elemento del arreglo */


main()
{
 int i;                          /* Indice para recorrer el arreglo */
 int *elementos[NUM_ELE];        /* Arreglo de apuntadores a ordenar */
 time_t semilla;                 /* Variable para generar diferentes
                                    series de números aleatorios */
 void insercion_directa();       /* Función para ordenar los elementos
                                    de acuerdo a dicho método */

 semilla = time(NULL);  /* Se le asigna el tiempo actual del
                           sistema en segundos */
 if (&semilla = = NULL)  /* Si no se pudo obtener la fecha del
                            sistema, salir */
   printf("\n\nError al generar los números aleatorios, intentar de
          nuevo\n");
 else
```

```
{
 srand(semilla);
 printf("\n\nEl arreglo a ordenar es:\n");
 for (i = POS_INI; i < NUM_ELE; i++)
 {
  elementos[i] = malloc(sizeof(int)); /* Asignar espacio en
                                         memoria al arreglo */
  printf("%d ", *elementos[i] = rand()); /* Se selecciona un
                                            entero de forma
                                            aleatoria */
 }
  insercion_directa(elementos);

 printf("\n\nEl arreglo ordenado es:\n");
 for (i = POS_INI; i < NUM_ELE; i++)
   printf("%d ", *elementos[i]);
 printf("\n");
 for (i = POS_INI; i < NUM_ELE; i++) /* Liberar la memoria
                                        ocupada por el arreglo */
   free(elementos[i]);
}
}


void insercion_directa(elem)

 int *elem[];        /* Arreglo de apuntadores a ordenar */

{
 int clave_menor; /* Variable para almacenar la posición del menor
                     elemento encontrado hasta el momento */
 int *temp;       /* Variable para realizar intercambio de
                     apuntadores*/
 int i,j;         /* Indices para recorrer el arreglo */

 for (i = POS_INI; i < NUM_ELE - 1; i++) /* Se recorre el arreglo de la
                                            primera posición a la
                                            penúltima */
 {
  clave_menor = i; /* Se toma como elemento menor al que apunta i
```

```
                    actualmente, ya que se compara con todos los
                    que estén de la posición i + 1 al final */
for (j = i + 1; j < NUM_ELE; j++) /* Se recorre el arreglo desde
                                     la siguiente posición de i
                                     al final */
   if (*elem[j] < *elem[clave_menor])
     clave_menor = j;
 if (clave_menor != i) /* Si se encontró un elemento más pequeño,
                          intercambiarlos */
{
  temp = elem[i];
  elem[i] = elem[clave_menor];
  elem[clave_menor] = temp;
 }
}
}
```

2.6 COMPARACION ENTRE LAS DOS VERSIONES.

La diferencia entre las versiones presentadas, es que en la segunda se utiliza un arreglo de apuntadores a los elementos en vez de un arreglo de elementos. Lo anterior requiere de espacio adicional, ya que se tienen los datos, y apuntadores a los datos que se van a ordenar. Sin embargo, se tiene la ventaja de que se intercambian apuntadores y no elementos.

Ahora, las comparaciones se realizan de manera indirecta, ya que el contenido de cada posición del arreglo es un apuntador a un entero y no un entero. Por ello, se antepone el operador *, para indicar que lo que se compara es el contenido de la dirección a dónde apunta el elemento actual del arreglo, y no el contenido de esa posición.

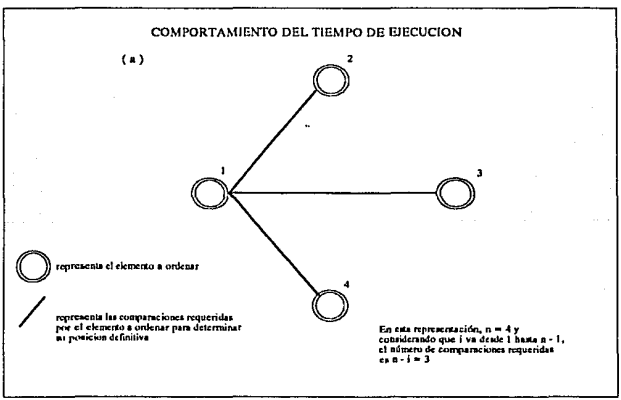
La instrucción *elementos[i] = getmem(sizeof(int))*, se utiliza para crear una variable de tipo apuntador que reserva el tamaño necesario para apuntar a un entero y posteriormente inicializarlo. Esto es con el fin de controlar a dónde señala el apuntador, ya que de otra manera los resultados son impredecibles.

La última instrucción del programa, *free(elementos[i]*, se utiliza para liberar la memoria ocupada por el arreglo de apuntadores, ya que ésta no se vuelve disponible por sí sola, sino que hay que indicarlo.

Como vemos, también se introduce una mayor complejidad al programa, ya que al utilizar apuntadores se tiene que tener mucho cuidado en como se inicializan y como se asignan valores a éstos, ya que de otro modo los resultados son impredecibles, y es muy difícil rastrear errores.

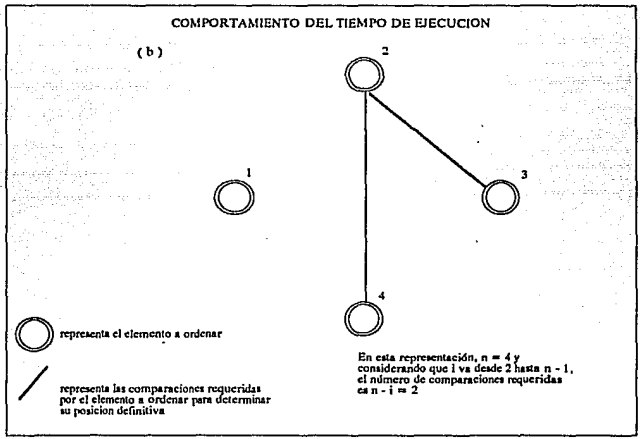
2.7 ANALISIS DE EFICIENCIA.

El análisis del ordenamiento de **Selección** es directo. En el primer paso se hacen n-1 comparaciones, tal como se muestra en la siguiente ilustración:

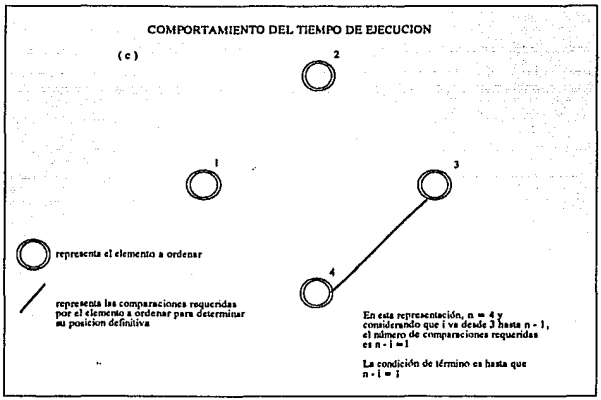


en el segundo paso n-2 comparaciones, como sigue:

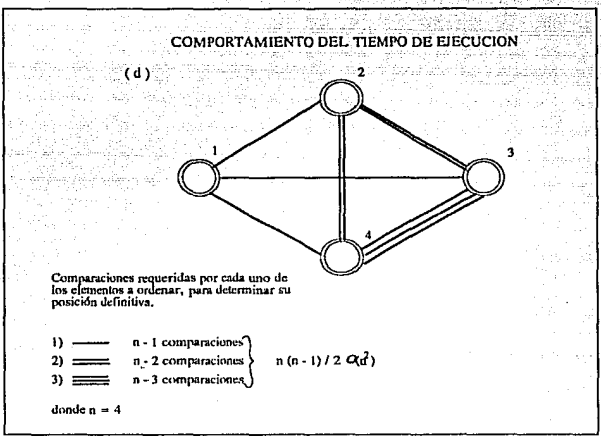
en el segundo paso n-2 comparaciones, como sigue:



y así sucesivamente,



comparaciones, lo cual es equivalente a $O(n^2)$" [KRU88, pág. 150]; esto se refleja en la siguiente ilustración:



Por lo tanto, el algoritmo de **Selección Directa** tiene un tiempo de ejecución proporcional a:

$$c\sum_{i=1}^{n-i}(n-i) = c(n-1)/2 = O(n^2)$$

Este ordenamiento puede entonces ser clasificado como del orden $O(n^2)$ tanto en el peor como en el caso promedio, ya que en la cantidad de comparaciones que realiza, prescinde del ordenamiento del conjunto inicial de elementos; en consecuencia, su comportamiento no ofrece ninguna ventaja en cuanto a que el conjunto inicial este o no ordenado,

debido a que el procedimiento de prueba se hace hasta que quede completo, independientemente de dicho estado inicial.

2.8 CONDICIONES EN EL ALGORITMO PRESENTA UN COMPORTAMIENTO EFICIENTE.

Como ya se mencionó, la función asintótica de tiempo del algoritmo de **Selección Directa**, presenta un comportamiento equivalente tanto en el caso promedio como en el peor caso, ya que no hay características de la entrada que lo hagan más eficiente, por lo que ofrece la ventaja de predicibilidad en el peor y en el caso promedio, orientando el análisis de eficiencia en los mismos criterios.

Dada la naturaleza de este algoritmo, su tiempo de ejecución siempre tendrá un comportamiento proporcional a $O(n^2)$, por lo que tendrá su mejor tiempo de ejecución para n pequeñas, ya que a medida que n se incrementa, aumenta exponencialmente la velocidad de crecimiento del tiempo de ejecución.

La ventaja principal de la clasificación por **Selección Directa** se centra en el movimiento de los datos. Si un elemento se encuentra en su posición correcta nunca será desplazado. Cada vez que un par de elementos se intercambia, por los menos uno se mueve hacia su posición definitiva y por consiguiente, a lo máximo n-1 intercambios se llevan a cabo en la clasificación de un conjunto de n elementos.

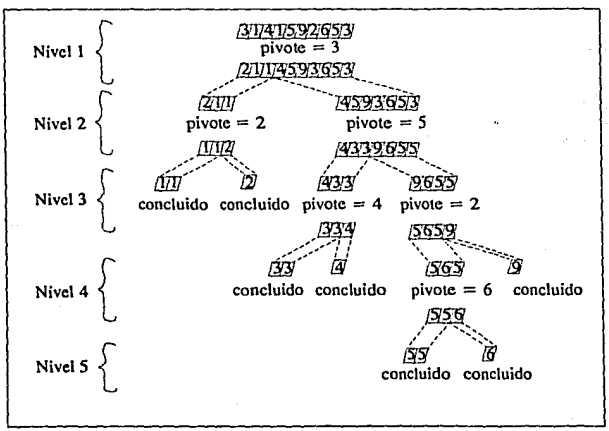
CAPITULO III

QUICKSORT

3.1 DESCRIPCION.

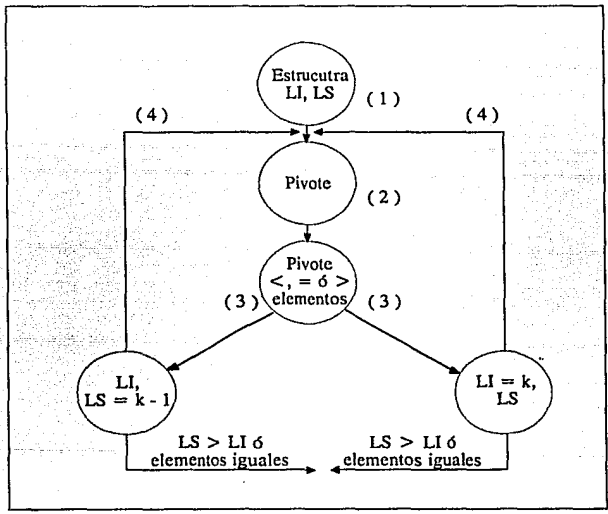
El método de clasificación de **Quicksort** consiste en ordenar un conjunto de n elementos, escogiendo a un elemento como pivote, a partir del cual se divide el conjunto a ordenar en dos subconjuntos con la característica de que el primer conjunto contendrá los elementos menores que el pivote y el segundo conjunto los mayores o iguales, esto se logra mediante un proceso de comparación del valor del elemento pivote con los elementos del conjunto a ordenar. Aplicando este proceso recursivamente hasta que sólo exista un elemento, ó todos los elementos sean iguales en cada uno de los dos subconjuntos finales, el conjunto inicial quedará ordenado, dado que los valores del primer subconjunto preceden a todos los valores del segundo subconjunto.

Esto esta representado en el siguiente ejemplo:



3.2 MODELO MATEMATICO.

El grafo que describe la forma en que se comporta el algoritmo de Quicksort es el siguiente:



En donde:

LI = Límite inferior del conjunto de elementos a ordenar.

LS = Límite superior del conjunto de elementos a ordenar.

Pivote = Valor a partir del cual se ordenan todos los elementos del conjunto delimitado por LI y LS. A la posición en dónde se encuentra el pivote se le denomina k. A partir

de dicha posición, se reorganiza el conjunto a ordenar. Los elementos menores se colocan a la izquierda de k y los elementos mayores o iguales son colocados a la derecha de k.

3.2.1 Descripción del comportamiento.

1.- Se tiene un conjunto de elementos a ordenar de entrada con un cierto número de elementos (n) finito, delimitados por LI y LS, y en el cual se designa al mayor elemento encontrado de izquierda a derecha como pivote.

2.- Se compara el pivote con los elementos restantes del conjunto, para determinar si es menor, en cuyo caso se acomoda a la izquierda del pivote, o si es mayor o igual, para colocarse a la derecha del pivote.

3.- Se obtienen dos subconjuntos a ordenar. Uno que va de LI a k - 1, y otro de k a LS, respectivamente.

4.- Si cada uno de los subconjuntos, cuenta con un solo elemento o todos los elementos son iguales, se termina el problema; en caso contrario, se repite el proceso para cada una de los dos subconjuntos desde el paso No. 1 hasta el No. 4.

3.3 ALGORITMO.

En términos generales el ordenamiento de **Quicksort** puede describirse mediante el siguiente algoritmo:

- determinar el elemento pivote

- dividir el conjunto de elementos a ordenar en dos subconjuntos, en función al pivote, mediante la comparación de éste con cada uno de los elementos del conjunto a ordenar, colocando a la izquierda del pivote los elementos

menores y a su derecha los mayores o iguales.

- repetir los dos pasos anteriores hasta que los 2 subconjuntos a ordenar tengan un sólo elemento o todos sus elementos sean iguales.

Un algoritmo más detallado se presenta a continuación:

Indice_pivote	=	posición del elemento mayor de más a la izquierda del conjunto de elementos a ordenar
Pivote	=	valor de la posición referida por Indice_menor
LI	=	límite inferior del conjunto de elementos a ordenar
LS	=	límite superior del conjunto de elementos a ordenar

Hacer mientras los subconjuntos de elementos a ordenar contengan más de un elemento o sus elementos sean diferentes:

Dividir el conjunto de elementos a ordenar en dos subconjuntos, colocando los elementos menores a pivote a su izquierda y los elementos mayores o iguales a su derecha

k = posición en la que inicia el subconjunto que contiene a los elementos mayores o iguales al pivote

Reasignar los límites superior e inferior de los subconjuntos de elementos a ordenar de acuerdo a k:

Límite inferior del subconjunto de elementos a ordenar que contiene los elementos menores que el pivote = LI

Límite superior del

subconjunto de elementos a ordenar que contiene los elementos menores que el pivote	=	k-1
Límite inferior del subconjunto de elementos a ordenar que contiene los elementos mayores que el pivote	=	k
Límite superior del subconjunto de elementos a ordenar que contiene los elementos mayores que el pivote	=	LS

3.4 PROGRAMA FUENTE DE LA PRIMERA VERSION.

A continuación se muestra la primera versión del algoritmo de quicksort.

```
/* Programa Quicksort, que realiza un ordenamiento de elementos
   utilizando dicho método */

# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10  /* Máximo número de elementos en el arreglo */
# define POS_INI 0   /* Posición del primer elemento del arreglo */


main()
{
 int i;                    /* Indice para recorrer el arreglo */
 int elementos[NUM_ELE];   /* Arreglo de los elementos a ordenar */
 time_t semilla;           /* Variable para generar diferentes
```

```
                                    series de números aleatorios */
void quicksort();                  /* Función quicksort para ordenamiento */

semilla = time(NULL);    /* Se le asigna el tiempo actual en
                            segundos */
if (&semilla == NULL)  /* Si no se pudo obtener la fecha del
                            sistema, salir */
  printf("\n\nError al generar los números aleatorios, intentar de
        nuevo\n");
else
 {
 srand(semilla);
 printf("\n\nEl arreglo a ordenar es:\n");
 for (i = POS_INI; i < NUM_ELE; i++)
   printf("%d ", elementos[i] = rand());  /* Se selecciona un
                                             entero de forma
                                             aleatoria */
 quicksort(elementos, POS_INI, NUM_ELE - 1);
 printf("\n\nEl arreglo ordenado es:\n");
 for (i = POS_INI; i < NUM_ELE; i++)
   printf("%d ", elementos[i]);
 printf("\n");
 }
}


void quicksort(elem, li, ls)

 int elem[];  /* Arreglo de elementos a ordenar */
 int li;      /* Límite inferior del arreglo a ordenar */
 int ls;      /* Límite superior del arreglo a ordenar */

{
 int pivote;                    /* Posición del elemento del arreglo
                                   seleccionado como pivote */
 int encuentra_pivote();        /* Función para determinar la posición
                                   del elemento pivote */
 int reorganiza_arreglo();  /* Función para acomodar los elementos
                               del arreglo con respecto al
                               pivote */
```

```
 pivote = encuentra_pivote(elem, li, ls); /* Seleccionar pivote */
 if (pivote != -1)
 {
  pivote = reorganiza_arreglo(elem, li, ls, elem[pivote]);
  if ((pivote - 1) > li) /* Checar que la partición izquierda
                            tenga más de un elemento */
    quicksort(elem, li, pivote - 1);
  if (ls > pivote)  /* Checar que la partición derecha
                       tenga más de un elemento */
    quicksort(elem, pivote, ls);
 }
}


int encuentra_pivote(elem, li, ls)

  /* Devuelve la posición del mayor de los dos primeros elementos
     que encuentre que son diferentes, empezando de izquierda a
     derecha.  Si todos los elementos son iguales, devuelve -1. */

  int elem[];  /* Arreglo de elementos a ordenar */
  int li;      /* Límite inferior del arreglo a ordenar */
  int ls;      /* Límite superior del arreglo a ordenar */

{
 int j;              /* Indice para recorrer el arreglo */
 int pos_inicial;  /* Posición del primer elemento del arreglo */

 pos_inicial = li;
 for (j = li + 1; j <= ls; j++)
   if (elem[j] > elem[pos_inicial])
     return j;
   else
     if (elem[j] < elem[pos_inicial])
       return pos_inicial;
   return -1;
}
```

```
int reorganiza_arreglo(elem, li, ls, piv)

 /* Coloca todos los elementos menores a pivote a su izquierda, y
    a todos los elementos mayores a su derecha */

 int elem[]; /* Arreglo de elementos a ordenar */
 int li;     /* Límite inferior del arreglo a ordenar */
 int ls;     /* Límite superior del arreglo a ordenar */
 int piv;    /* Elemento del arreglo seleccionado como pivote */

{
 int izq, der; /* Indices para recorrer el arreglo de izquierda a
                  derecha y viceversa, respectivamente */
 int temp;     /* Variable para realizar intercambio de
                  elementos */
 izq = li;
 der = ls;
 do
 {
  temp = elem[izq]; /* Hacer el intercambio de elementos para
                       acomodarlos con respecto al pivote */
  elem[izq] = elem[der];
  elem[der] = temp;
  while (elem[izq] < piv) /* Recorre el arreglo de izquierda a
                             derecha hasta encontrar un elemento
                             mayor a pivote */
     izq++;
  while (elem[der] >= piv) /* Recorre el arreglo de derecha a
                              izquierda hasta encontrar un
                              elemento menor o igual a
                              pivote */
     der--;
 } while (izq < der);
 return izq; /* Posición del elemento pivote */
}
```

3.4.1 Análisis del programa

El algoritmo se divide en tres partes: encontrar el elemento pivote, acomodar a los elementos restantes del arreglo con respecto a éste, y mandar llamar recursivamente a la función de *quicksort*.

La función de *encuentra_pivote* asegura la eficiencia del algoritmo al no seleccionar nunca al elemento más pequeño como pivote. Además, checa el caso de que todos los elementos sean iguales, para no aplicar la función de quicksort.

La función de *reorganiza* arreglo acomoda a los demás elementos con respecto del pivote, para que todos los que sean mayores o iguales a él se coloquen a su derecha, y todos los que sean menores a él se coloquen a su izquierda, y se devuelve la posición del pivote dentro del arreglo.

Por último, se manda llamar recursivamente a *quicksort* con el subconjunto de elementos menores a pivote, y con el de elementos mayores o iguales a pivote, y así sucesivamente hasta que se tiene ordenado todo el arreglo. Antes de mandarse llamar recursivamente, checa si hubo elementos diferentes, y si las particiones son de más de un elemento.

3.5 PROGRAMA FUENTE DE LA SEGUNDA VERSION.

Las diferencias con la versión anterior se muestran en negrillas.

```
/* Programa Quicksort, que realiza un ordenamiento de elementos
   utilizando dicho método (segunda versión con apuntadores) */
# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10  /* Máximo número de elementos en el arreglo */
# define POS_INI 0   /* Posición del primer elemento del arreglo */
```

```
main()
{
 int i;                          /* Indice para recorrer el arreglo */
 int *elementos[NUM_ELE];        /* Arreglo de apuntadores a los
                                    elementos a ordenar */
 time_t semilla;                 /* Variable para generar diferentes
                                    series de números aleatorios */
 void quicksort();               /* Función quicksort para
                                    ordenamiento */

 semilla = time(NULL); /* Se le asigna el tiempo actual en
                          segundos */
 if (&semilla == NULL) /* Si no se pudo obtener la fecha del
                          sistema, salir */
  printf("\n\nError al generar los números aleatorios, intentar de
          nuevo\n");
 else
  {
   srand(semilla);
   printf("\n\nEl arreglo a ordenar es:\n");
   for (i = POS_INI; i < NUM_ELE; i++)
    {
     elementos[i] = malloc(sizeof(int)); /* Asignar espacio
                                            al arreglo */
     printf("%d ", *elementos[i] = rand()); /* Se selecciona un
                                               entero de forma
                                               aleatoria */
    }
   quicksort(elementos, POS_INI, NUM_ELE - 1);
   printf("\n\nEl arreglo ordenado es:\n");
   for (i = POS_INI; i < NUM_ELE; i++)
     printf("%d ", *elementos[i]);
   printf("\n");
   for ( i = POS_INI; i < NUM_ELE; i++)
     free(elementos[i]); /* Liberar el espacio ocupado por
                            el arreglo */
  }
}
```

```
void quicksort(elem, li, ls)

 int *elem[];              /* Arreglo de apuntadores a los elementos a
                              ordenar */
 int li;                   /* Límite inferior del arreglo a ordenar */
 int ls;                   /* Límite superior del arreglo a ordenar */
 int pivote;               /* Variable que apunta a la posición
                              del elemento del arreglo
                              seleccionado como pivote */
 int encuentra_pivote();   /* Función para determinar la posición
                              del elemento pivote */
 int reorganiza_arreglo(); /* Función para acomodar los elementos
                              del arreglo con respecto al
                              pivote */

 pivote = encuentra_pivote(elem, li, ls); /* Seleccionar pivote */
 if (pivote != -1)
 {
  pivote = reorganiza_arreglo(elem, li, ls, elem[pivote]);
  if ((pivote - 1) > li) /* Checar que la partición izquierda
                            tenga más de un elemento */
    quicksort(elem, li, pivote - 1);
  if (ls > pivote) /* Checar que la partición derecha
                      tenga más de un elemento */
    quicksort(elem, pivote, ls);
 }
}


int encuentra_pivote(elem, li, ls)

  /* Devuelve la posición del mayor de los dos primeros elementos
     que encuentre que son diferentes, empezando de izquierda a
     derecha.  Si todos los elementos son iguales, devuelve -1. */

  int *elem[]; /* Arreglo de apuntadores a los elementos a
                  ordenar */
  int li;      /* Límite inferior del arreglo a ordenar */
  int ls;      /* Límite superior del arreglo a ordenar */
{
```

```
int j;                  /* Indice para recorrer el arreglo */

int pos_inicial;  /* Posición del apuntador al primer elemento del
                     arreglo */

pos_inicial = li;
for (j = li + 1; j <= ls; j++)
  if (*elem[j] > *elem[pos_inicial])
    return j;
  else
    if (*elem[j] < *elem[pos_inicial])
      return pos_inicial;
return -1;
}


int reorganiza_arreglo(elem, li, ls, piv)

 /* Coloca todos los elementos menores a pivote a su izquierda, y
    a todos los elementos mayores a su derecha */

 int *elem[];  /* Arreglo de apuntadores a los elementos a
                  ordenar */
 int li;       /* Límite inferior del arreglo a ordenar */
 int ls;       /* Límite superior del arreglo a ordenar */
 int *piv;     /* Variable que apunta al elemento del arreglo
                  seleccionado como pivote */

{
 int izq, der;  /* Indices para recorrer el arreglo de izquierda a
                   derecha y viceversa, respectivamente */
 int *temp;  /* Variable para realizar intercambio de
                apuntadores */

 izq = li;
 der = ls;
 do
 {
  temp = elem[izq];  /* Hacer el intercambio de apuntadores a los
                        elementos para acomodarlos con respecto al
```

```
                        pivote */
 elem[izq] = elem[der];
 elem[der] = temp;
 while (*elem[izq] < *piv) /* Recorre el arreglo de izquierda a
                              derecha hasta encontrar un elemento
                              mayor a pivote */
    izq++;
  while (*elem[der] >= *piv) /* Recorre el arreglo de derecha a
                                izquierda hasta encontrar un
                                elemento menor o igual a pivote */
    der--;
 } while (izq < der);
 return izq; /* Posición del elemento pivote */
}
```

3.6 COMPARACION ENTRE LAS DOS VERSIONES.

La diferencia es que en la segunda versión se crea un arreglo de apuntadores a los elementos del arreglo que se va a ordenar. Se efectúan las comparaciones entre los valores de los elementos apuntados, pero sin mover los elementos; en vez de esto se mueven los apuntadores a los elementos de la misma forma que la primera versión mueve los elementos. Al final los apuntadores, leídos de izquierda a derecha, apuntan a los elementos en el orden deseado.

Por el lado negativo, se requiere espacio adicional para el arreglo de apuntadores, y el acceso a los elementos para efectuar las comparaciones es más lento que antes, ya que debe seguir primero el apuntador, y luego ir al elemento. Asimismo, se requiere asignar espacio al arreglo de apuntadores y luego liberarlo.

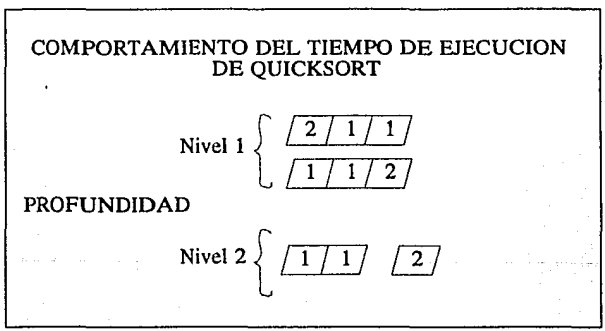
3.7 ANALISIS DE EFICIENCIA.

El algoritmo de **Quicksort** en cuanto a tiempo de ejecución "...lleva en promedio un tiempo $O(n\log n)$,..." [SED90, pág. 115], para clasificar n elementos, y "...en el peor caso lleva un tiempo proporcional a $O(n^2)$,..."

[SED90, pág. 115].

Así, con excepción de las llamadas recursivas que hace **Quicksort**, cada llamada individual de la función *Quicksort* lleva un tiempo máximo proporcional al número de elementos que se le pide clasificar.

En otras palabras, "...el tiempo total consumido por *quicksort* es la suma en todos los elementos, de las veces que el elemento forma parte del subarreglo en el que se hizo la llamada a *quicksort*." [AHO88, pág. 265]. Es evidente que ningún elemento puede incluirse en dos llamadas del mismo nivel, así que el tiempo consumido por *Quicksort* puede expresarse como la suma en todos los elementos de la profundidad o máximo nivel, en el cual se encuentra ese elemento, como se muestra en la siguiente ilustración:



3.8 DETERMINACION DE LAS CONDICIONES EN LAS CUALES EL ALGORITMO PRESENTA UN COMPORTAMIENTO EFICIENTE.

A medida que el número de elementos a ordenar aumenta, la función asintótica de tiempo $T_{(n)}$ se transforma de un comportamiento logarítmico a uno exponencial, por lo que el algoritmo de **Quicksort** se hace menos eficiente, es decir tiene un mayor tiempo de ejecución en tanto **n** aumenta. Debido a que el comportamiento de la función asintótica de tiempo para **Quicksort**, en el caso promedio esta dada por la relación logarítmica de $O(n\log n)$, y en el peor caso por $O(n^2)$, la razón que justifica la utilización de este algoritmo en la práctica, es precisamente su comportamiento en el caso promedio y no en el peor caso.

CAPITULO IV

ARBOL BINARIO

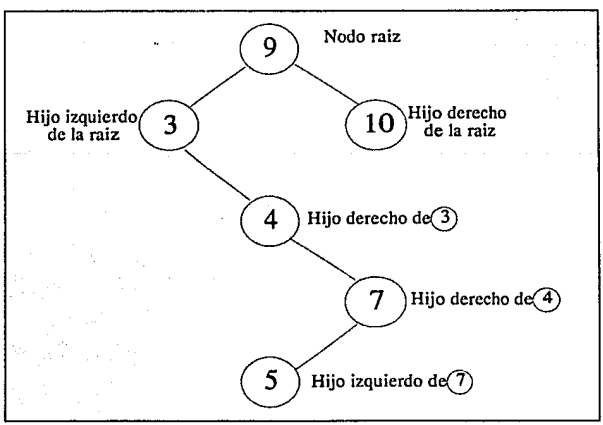
4.1 DESCRIPCION.

El método de ordenamiento por **Árbol Binario** consiste en explorar cada uno de los elementos del conjunto a ordenar y colocarlo en su posición apropiada en un árbol binario. Para encontrar la posición adecuada de un elemento x, una ramificación izquierda o derecha se toma en cada nodo, dependiendo de si x es menor que el elemento en el nodo, o mayor que éste, o igual.

Una vez que cada elemento de entrada esta en su posición correcta en el árbol, el conjunto ordenado puede obtenerse mediante un recorrido de entreorden del árbol, por ejemplo del siguiente conjunto de elementos a ordenar:

9,3,4,7,5,10

sus posiciones en un árbol binario son:

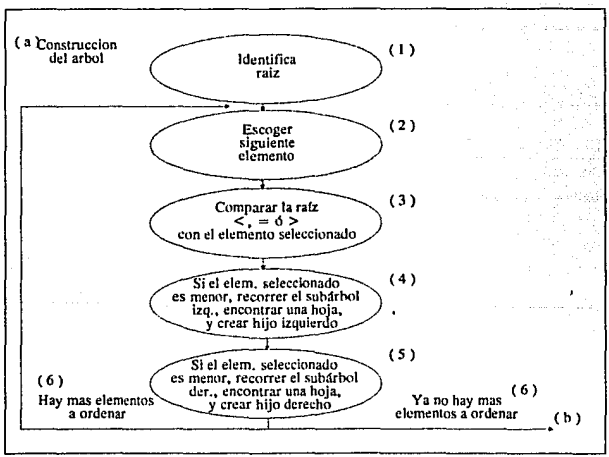


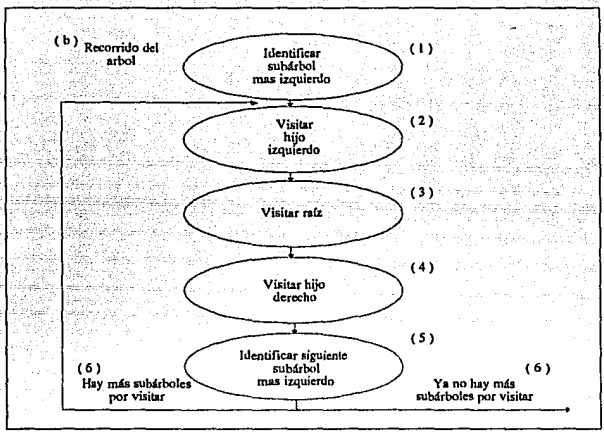
Para poder obtener el conjunto de elementos ordenado, se recorre en entreorden el árbol binario anterior, con lo que resulta el siguiente conjunto de elementos ordenado:

3,4,5,7,9,10

4.2 MODELO MATEMATICO.

El grafo que describe el algoritmo de ordenamiento de **Árbol Binario** se presenta a continuación:





4.2.1 Descripción del comportamiento.

(a) Construcción del árbol.

1.- Escoger el primer elemento del conjunto de elementos a ordenar, como elemento Raíz.

2.- Escoger el siguiente elemento del conjunto a ordenar.

3.- Comparar si el elemento seleccionado en el paso número 2 es menor, igual, o mayor con respecto a la raíz.

4.- Si el elemento seleccionado en el paso número 2 es menor que la raíz, entonces se debe recorrer el subárbol izquierdo hasta encontrar una hoja, y crear el hijo izquierdo.

5.- Si el elemento seleccionado en el paso número 2 es mayor

que la raíz, entonces se debe recorrer el subárbol derecho hasta encontrar una hoja, y crear el hijo derecho.

6.- Si hay más elementos en el conjunto a ordenar, entonces repetir del paso número 2 hasta el número 6; en caso contrario, continuar con el recorrido del árbol en **entreorden.**

(b) Recorrido del árbol.

1.- Identificar el subárbol más izquierdo.

2.- Realizar el acceso al hijo izquierdo, del subárbol identificado.

3.- Realizar el acceso a la raíz, del subárbol identificado.

4.- Realizar el acceso al hijo derecho, del subárbol identificado.

5.- Identificar el siguiente subárbol más izquierdo.

6.- Si hay más subárboles, entonces repetir del paso número 2 al paso número 5; en caso contrario, se termina el problema.

4.3 ALGORITMO.

(a) Construcción del árbol.

raíz = primer elemento del conjunto a ordenar

Hacer mientras haya elementos a ordenar:

elemento = siguiente elemento del conjunto a ordenar

comparar elemento con la raíz:

si es menor entonces:

- recorrer subárbol izquierdo, hasta encontrar una hoja

- crear hijo izquierdo

si es mayor o igual entonces:

- recorrer subárbol derecho, hasta encontrar una hoja

- crear hijo derecho

(b) Recorrido del árbol.

Identificar subárbol más izquierdo a partir de la raíz.

hacer mientras existan subárboles por visitar:

- visitar hijo izquierdo
- visitar raíz
- visitar hijo derecho
- identificar siguiente subárbol más izquierdo

4.4 PROGRAMA FUENTE DE LA PRIMERA VERSION.

```
/* Programa Árbol Binario, que realiza un ordenamiento de elementos
   utilizando dicho método. Para ello, primero construye el árbol,
   ordena los elementos a través de apuntadores a los hijos, y
   después lo recorre en entreorden para imprimirlos ordenados */

# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10  /* Máximo número de elementos en el árbol */
# define POS_INI 0   /* Posición del primer elemento del árbol */
```

```
struct nodo_arbol  /* Estructura de cada nodo del árbol */
 {
  int elemento;     /* Elemento a ordenar */
  int hijo_izq;     /* Posición del hijo izquierdo */
  int hijo_der;     /* Posición del hijo derecha */
 };


main()
 {
  struct nodo_arbol arbol[NUM_ELE];      /* Arbol de los elementos a
                                            ordenar */
  int i;                                 /* Indice para recorrer el
                                            árbol */
  time_t semilla;                        /* Variable para generar
                                            diferentes series de
                                            números aleatorios */
  void ordenar_elemento_en_el_arbol();   /* Función para colocar a
                                            cada elemento en la
                                            posición que le
                                            corresponde en el
                                            árbol */


  void entreorden();                     /* Función que recorre e
                                            imprime el árbol en
                                            entreorden */

  semilla = time(NULL);
  if (&semilla = = NULL)  /* Si no se pudo obtener la fecha del
                             sistema, salir */
    printf("\n\nError al generar los números aleatorios, intentar de
          nuevo\n");
  else
    {
     srand(semilla);
     printf("\n\nLos elementos a ordenar son:\n");
     for (i = POS_INI; i < NUM_ELE; i+ +)  /* a través de los
```

```
                                        apuntadores a los hijos
                                        se define el orden de
                                        cada elemento en el
                                        árbol */
  {
   printf("%d ", arbol[i].elemento = rand()); /* Se selecciona
                                                un entero de
                                                forma aleatoria */
   arbol[i].hijo_izq = -1; /* Se inicializan los hijos de cada
                              nuevo nodo para que no apunten a
                              nada */
   arbol[i].hijo_der = -1;
   if (i > POS_INI) /* Si no es la raíz */
    ordenar_elemento_en_el_arbol(arbol, i, arbol[i].elemento);
  }
  printf("\n\nLos elementos ordenados son:\n");
  entreorden(arbol, POS_INI);
  printf("\n");
 }
}


void ordenar_elemento_en_el_arbol(arb, pos_nodo, elem)

 struct nodo_arbol arb[]; /* Arbol de elementos a ordenar */
 int pos_nodo;            /* Posición del elemento a ordenar */
 int elem;                /* Elemento a ordenar */

{
 int j, k; /* Indices para recorrer el árbol */

 j = k = POS_INI;
 while (k != -1) /* Mientras k apunte a algún elemento en el árbol */
 {
  j = k;
  if (elem < arb[j].elemento) /* Se determina si el elemento es
                                 hijo izquierdo o derecho del
                                 elemento actual, hasta que se
                                 encuentra su posición en el
                                 árbol, es decir, al encontrar un
```

```
                                elemento sin el hijo
                                correspondiente (derecho o
                                izquierdo */
  k = arb[j].hijo_izq;
 else
  k = arb[j].hijo_der;
}
if (elem < arb[j].elemento) /* Se actualiza el hijo izquierdo o
                               derecho del elemento
                               correspondiente para que apunte al
                               nuevo elemento */
  arb[j].hijo_izq = pos_nodo;
 else
  arb[j].hijo_der = pos_nodo;
}


void entreorden(arb, pos_nodo)

 struct nodo_arbol arb[]; /* Arbol de elementos */
 int pos_nodo;            /* Posición del elemento en el árbol */
{
 if (pos_nodo != -1) /* Si se está apuntando a un elemento del
                        árbol */
 {

  entreorden(arb, arb[pos_nodo].hijo_izq); /* Se llama
                                              recursivamente para
                                              procesar primero
                                              todo el subárbol izquierdo
                                              del elemento actual (es
                                              decir, todos los elementos
                                              menores a él */
  printf("%d ", arb[pos_nodo].elemento); /* Se imprime el nodo
                                            actual */
  entreorden(arb, arb[pos_nodo].hijo_der); /* Por último, se llama
                                              recursivamente para
                                              procesar todo el
                                              subárbol derecho del
                                              elemento actual (es
```

```
                                    decir, todos los
                                    elementos mayores a
                                    él */
}
}
```

4.4.1 Análisis del programa

Primero, se define una estructura denominada *nodo_árbol*, la cual se encuentra dividida en tres partes: el elemento contenido en dicho nodo, la posición del hijo izquierdo, y la posición del hijo derecho. Lo anterior se debe a que los elementos se irán almacenando en posiciones contiguas de memoria (arreglo *árbol*) según como vayan llegando, pero no estarán ordenados, por lo que se necesita algo que nos indique en que orden están los elementos, los cuales son las posiciones del hijo izquierdo y derecho de cada nodo.

Es así que el programa se divide en tres partes: inicialización del arreglo, modificación de las posiciones del hijo izquierdo y derecho de los nodos que corresponda para mantener el árbol en orden, y recorrido e impresión del árbol en el orden adecuado.

Primero, en el programa principal se asigna cada elemento aleatorio a la posición que le corresponda en el arreglo, es decir, el primer número aleatorio a la primera posición, el segundo a la segunda posición y así sucesivamente. Los hijos izquierdo y derecho de cada nuevo nodo se inicializan a -1, para indicar que no apuntan a nada, ya que las posiciones válidas de nodos van de 0 a 9.

El primer elemento del arreglo siempre será la raíz, por lo que no se requiere definir una variable raíz como se mencionó en el algoritmo, ya que la variable *POS_INI* siempre apunta al primer elemento del arreglo.

Como los demás elementos se organizan con respecto a la raíz, a ésta no se le aplica la función de *ordenar_elemento_en_el_árbol*, cuya función es colocar en el árbol a cada nuevo nodo en el orden que le corresponde.

En dicha función para cada nuevo nodo a partir del segundo, se recorren

los elementos en el árbol a partir de la raíz. Si el nuevo elemento es menor a la raíz, se busca el elemento al que señala el hijo izquierdo de la raíz, y en caso contrario al del hijo derecho. Se sigue este proceso hasta que se encuentra un elemento (nodo) sin el hijo correspondiente (es decir, que contenga -1), y se actualiza para que apunte a la posición en el árbol en que se encuentra el nuevo elemento.

Por último, se tiene la función para recorrer el árbol en entreorden, denominada *entreorden*, para la cual a partir de la raíz se sigue por los hijos izquierdos hasta encontrar uno que contenga -1, lo que nos indica que llegamos al final de dicho rama, y se imprime el elemento, después la raíz y posteriormente, se realiza el mismo proceso con el hijo derecho. Aplicando recursivamente el proceso anterior para cada subárbol, se imprimen los elementos en el orden correcto.

4.5 PROGRAMA FUENTE DE LA SEGUNDA VERSION.

```
/* Programa Arbol Binario, que realiza un ordenamiento de elementos
   utilizando dicho método. Para ello, primero construye el árbol,
   ordena los elementos a través de apuntadores a los hijos, y
   después lo recorre en entreorden para imprimirlos ordenados */

# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10  /* Máximo número de elementos en el árbol */
# define POS_INI 0   /* Posición del primer elemento del árbol */


struct nodo_arbol  /* Estructura de cada nodo del
                      árbol */
 {
  int elemento;                 /* Elemento a ordenar */
  struct nodo_arbol *hijo_izq;  /* Apuntador al hijo izquierdo */
  struct nodo_arbol *hijo_der;  /* Apuntador al hijo derecho */
 };
```

```
main()
{
 struct nodo_arbol *raiz_arbol;          /* Apuntador al primer
                                            elemento del árbol */
 struct nodo_arbol *apuntador_nodo; /* Apuntador a un elemento
                                            del árbol */
 int i;                                  /* Indice para recorrer el
                                            árbol */
 time_t semilla;                         /* Variable para generar
                                            diferentes series de
                                            números aleatorios */
 void ordenar_elemento_en_el_arbol(); /* Función para colocar a
                                            cada elemento en la
                                            posición que le
                                            corresponde en el
                                            árbol */
 void entreorden();                      /* Función que recorre e
                                            imprime el árbol en
                                            entreorden */


 semilla = time(NULL);
 if (&semilla == NULL)  /* Si no se pudo obtener la fecha del
                           sistema, salir */
   printf("\n\nError al generar los números aleatorios, intentar de
         nuevo\n");
 else
   {
    srand(semilla);
    printf("\n\nLos elementos a ordenar son:\n");
    for (i = POS_INI; i < NUM_ELE; i++)  /* A través de los
                                            apuntadores a los
                                            hijos se define el
                                            orden de cada
                                            elemento en el árbol */
    {
     apuntador_nodo = malloc(sizeof(struct nodo_arbol));
     printf("%d ",apuntador_nodo->elemento=rand()); /* Se
                                                      selecciona
```

```
                                                    un entero de
                                                    forma
                                                    aleatoria */
    apuntador_nodo->hijo_izq = NULL;  /* Se inicializa a los hijos
                                         de cada nuevo nodo para
                                         que no apunten a nada */
    apuntador_nodo->hijo_der = NULL;
    if (i == POS_INI)  /* Preguntar si es el primer elemento */
     raiz_arbol = apuntador_nodo;
    else
     ordenar_elemento_en_el_arbol(raiz_arbol, apuntador_nodo);
   }
   printf("\n\nLos elementos ordenados son:\n");
   entreorden(raiz_arbol);
   printf("\n");
  }
}


void ordenar_elemento_en_el_árbol(raiz, apun_nodo)

  struct nodo_arbol *raiz;          /* Apuntador al primer elemento
                                       del árbol de elementos a
                                       ordenar */
  struct nodo_arbol *apun_nodo;  /* Apuntador al elemento a
                                    ordenar */

{
 struct nodo_arbol *j, *k;  /* Apuntadores para recorrer el
                               árbol */

 j = k = raiz;
 while (k != NULL)  /* Mientras k apunte a algún elemento en el
                       árbol */
 {
  j = k;
  if (apun_nodo->elemento < j->elemento)  /* Se determina si el
                                             elemento es hijo
                                             izquierdo o derecho
                                             del elemento actual,
```

```
                                                    hasta que se
                                                    encuentra su posición
                                                    en el árbol, es decir,
                                                    al encontrar un
                                                    elemento sin el hijo
                                                    correspondiente
                                                    (derecho o
                                                    izquierdo */
    k = j->hijo_izq;
  else
    k = j->hijo_der;
 }
 if (apun_nodo->elemento < j->elemento)  /* Se actualiza el hijo
                                            izquierdo o derecho del
                                            elemento
                                            correspondiente para
                                            que apunte al nuevo
                                            elemento */
    j->hijo_izq = apun_nodo;
  else
    j->hijo_der = apun_nodo;
 }


 void entreorden(apunt_nodo)

   struct nodo_árbol *apunt_nodo;  /* Posición del elemento en el
                                      Arbol */

 {
  if (apunt_nodo != NULL) /* Si se está apuntando a un elemento del
                             árbol */
  {
   entreorden(apunt_nodo->hijo_izq);  /* Se llama recursivamente
                                         para procesar primero todo
                                         el subárbol izquierdo del
                                         elemento actual (es decir,
                                         todos los elementos
                                         menores a él) */
   printf("%d ", apunt_nodo->elemento);  /* Se imprime el elemento
```

```
                                        actual */
 entreorden(apunt_nodo->hijo_der); /* Por último, se llama
                                      recursivamente para
                                      procesar todo el subárbol
                                      derecho del elemento actual
                                      (es decir, todos los
                                      elementos mayores a él) */
 free(apunt_nodo); /* Liberar la memoria ocupada */
 }
}
```

4.6 COMPARACION ENTRE LAS DOS VERSIONES.

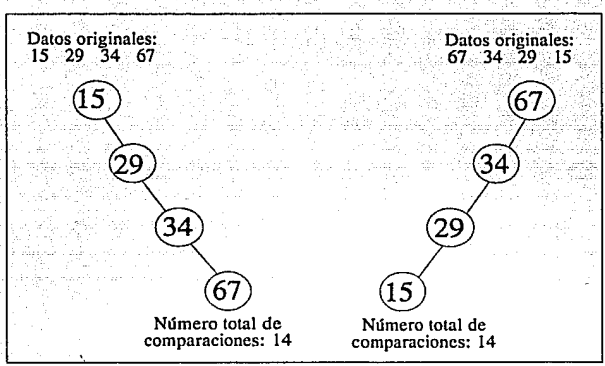
Al igual que en los programas anteriores, la diferencia estriba en que en la segunda versión se reduce el tiempo de ejecución ya que se intercambian apuntadores y no elementos, pero se requiere de mayor espacio en memoria ya que se tiene que reservar espacio a los apuntadores además de a los elementos.

Además, al manejar apuntadores, ya no se requiere de posiciones contiguas en memoria, por lo que no se define un arreglo para guardar los elementos en el árbol, sino únicamente un apuntador denominado raíz, el cual apunta al primer elemento del árbol.

El utilizar apuntadores permite hacer más rápido las operaciones de recorrido y actualización de los hijos del árbol.

4.7 ANALISIS DE EFICIENCIA.

La eficiencia de este algoritmo, depende del estado inicial de los elementos del conjunto inicial: si éstos están ordenados (u ordenados en sentido inverso), el árbol resultante es una cadena, es decir, aparece como una secuencia de únicamente enlaces derecho o izquierdo, como se observa en la siguiente figura:



Para la inserción del primer nodo no se requiere ninguna comparación, el segundo necesita dos comparaciones, el tercer nodo tres comparaciones, y así sucesivamente: "Es decir que el número total de comparaciones es:

$$2 + 3 + \ldots + n = n(n + 1)/2 - 1$$ [TEN85, pág. 399].

con lo que se infiere que el tiempo de ejecución del ordenamiento de árbol binario en el peor caso (cuando el conjunto inicial de elementos esta ordenado) es proporcional a $O(n^2)$.

"Por otro lado, si los elementos a ordenar del conjunto inicial están organizados de tal manera que aproximadamente la mitad de los elementos que siguen a cualquier elemento *a* en el conjunto inicial son menores que *a* y la mitad son mayores o iguales, la profundidad del árbol binario resultante es el entero más pequeño *d* mayor o igual a $\log_2(n + 1) - 1$. El número de nodos a cualquier nivel *l* (excepto posiblemente el último) es 2^l, y el número de comparaciones que se requieren para colocar un nodo en el nivel *l* (excepto cuando *l* = 0) es *l* + 1.

Es decir, que el número total de comparaciones está entre :

$$d + \sum_{l=1}^{d-1} 2^l(l+1) \quad y \quad \sum_{l=1}^{d-1} 2^l(l+1)$$

Se puede demostrar que la suma resultante es del orden $O(n\log n)$." [TEN85, pág. 400].

4.8 DETERMINACION DE LAS CONDICIONES EN LAS QUE EL ALGORITMO PRESENTA UN COMPORTAMIENTO EFICIENTE.

El algoritmo de ordenamiento por **Árbol Binario** y el de **Quicksort** presentan un comportamiento semejante en cuanto a las comparaciones requeridas para realizar el ordenamiento, lo cual se revela por el hecho de que en **Árbol Binario** el primer elemento a ordenar entra directamente en la raíz del árbol de búsqueda, sin comparaciones, y a medida que cada nuevo elemento llega, primero se compara con la raíz y luego entra en el subárbol izquierdo o derecho; por otro lado en **Quicksort** en la primera etapa cada elemento se compara con el pivote y luego se pone en el subconjunto izquierdo o derecho.

No obstante lo anterior, en la clasificación por **Árbol Binario** cada elemento al llegar va a su posición definitiva en la estructura ligada. El segundo elemento se convierte en la raíz del subárbol izquierdo o derecho (según su comportamiento con el valor de la raíz). Después, todos los elementos que entran en el mismo árbol se comparan con ese segundo elemento. De manera análoga, en la clasificación **Quicksort** todos los elementos en un subconjunto se comparan con el segundo pivote, o sea el pivote de este subconjunto, de tal manera que:

> "La clasificación por **Árbol Binario** hace exactamente las mismas comparaciones de elementos que **Quicksort**, cuando el pivote de cada sublista se escoge de manera que sea el primer elemento en la sublista." [KRU88, pág. 203].

Como se analizó en el capítulo III, la clasificación por **Quicksort** es un método excelente en el caso promedio. De ahí que se infiera que la clasificación por **Árbol Binario** constituya también un excelente método en lo tocante a las comparaciones de elementos. Sin embargo, la clasificación de **Quicksort** necesita tener acceso a todos los elementos que se ordenarán en todo el proceso.

Con la clasificación por **Árbol Binario** no es preciso que los elementos estén disponibles al principiar el proceso, sino que van incorporándose al árbol conforme se tornen disponibles. De ahí que esta técnica sea preferible en aplicaciones donde los elementos se reciben uno a la vez. La ventaja principal es que el árbol de búsqueda queda disponible para hacer inserciones y eliminaciones posteriores, y que puede buscarse después en tiempo logarítmico, en cambio los métodos de clasificación anteriores exigían listas contiguas en las cuales son difíciles las inserciones y eliminaciones o bien producían listas simplemente ligadas para las cuales no se dispone más que de la búsqueda secuencial.

Las desventajas fundamentales de la clasificación por **Árbol binario** ya están implícitas en el teorema mencionado. La clasificación por **Quicksort** tiene un rendimiento muy deficiente en el peor caso y, aunque con una cuidadosa selección de los pivotes disminuye muchísimo la probabilidad de este caso, escoger que el pivote sea el primer elemento en cada sublista hace aparecer el peor caso cuando los elementos ya están clasificados. Si se presentan ya clasificados al método de **Árbol Binario**, entonces éste será un desastre, ya que el árbol de búsqueda que construye se reducirá a una cadena. Esta técnica nunca deberá aplicarse cuando los elementos ya estén clasificados o casi clasificados.

En el caso de los problemas pequeños con dimensiones reducidas, el almacenamiento contiguo suele ser la mejor opción, pero habrá que preferir el almacenamiento ligado para problemas grandes y registros voluminosos.

CAPITULO V

CON CALCULO DE DIRECCION

5.1 DESCRIPCION.

El método de ordenamiento **Con Cálculo de Dirección** (denominado hashing en inglés), considera un conjunto inicial de elementos por ordenar, al cuál se le requiere aplicar una función que determine la posición que será adoptada por cada uno de sus elementos: "La función debe generar que si $x < y$, entonces $f(x) < f(y)$" [TEN85, pág. 421], lo cual significa que si un elemento x es menor que otro y, el resultado de la función hash de x debe ser menor al de la función hash de y.

Como ya se mencionó en el capítulo I, no existe una función hash universal, sino que depende del problema en específico de que se trate. Nosotros partiremos del supuesto de que se tiene que ordenar conjuntos de números de dos dígitos, es decir, en el rango de 0 a 99.

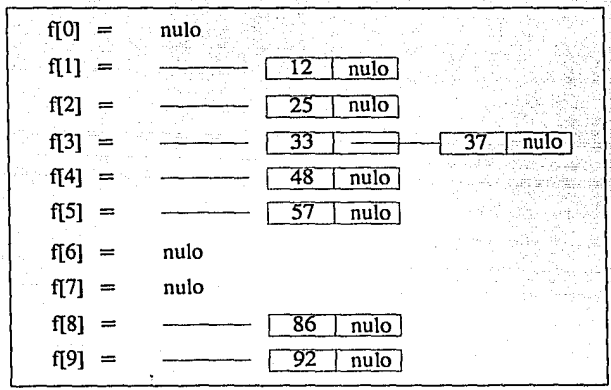
Un elemento se coloca en un subconjunto en la secuencia apropiada utilizando cualquier método de ordenamiento (En los programas de las secciones 5.4 y 5.5 se utilizará el método de **Selección Directa**, debido a que serán subconjuntos de elementos muy pequeños, en promedio de dos a tres). Después de que todos los elementos del conjunto inicial se han colocado en los subconjuntos, éstos pueden ser concatenados para producir el resultado de ordenamiento.

Por ejemplo, si se considera el conjunto siguiente:

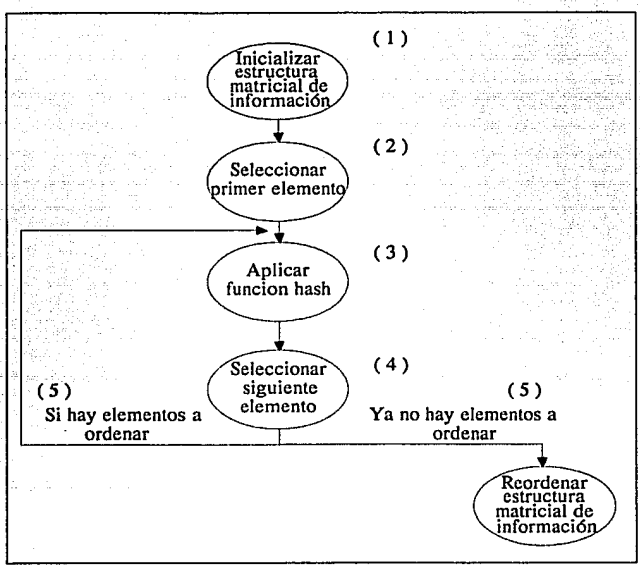
25, 57, 48, 37, 12, 92, 86, 33

se crearían diez subconjuntos, uno para cada uno de los diez primeros dígitos posibles. Inicialmente, cada uno de estos subconjuntos esta vacío. Se declara un arreglo de apuntadores f(0..9), dónde f(i) apunta al primer elemento del archivo cuyo primer dígito es i. Después de explorar el primer elemento (25), este es colocado en el encabezamiento del archivo f(2). Cada uno de los subconjuntos es mantenido como una lista encadenada ordenada del conjunto original de elementos.

Una vez que se procesa cada uno de estos elementos en el conjunto original, los subconjuntos tendrían la forma que se muestra en la siguiente figura:


f[0] = nulo
f[1] = 12 nulo
f[2] = 25 nulo
f[3] = 33 37 nulo
f[4] = 48 nulo
f[5] = 57 nulo
f[6] = nulo
f[7] = nulo
f[8] = 86 nulo
f[9] = 92 nulo

5.2 MODELO MATEMATICO.



5.2.1 Descripción del comportamiento.

1.- Inicializar la estructura matricial que contendrá a los elementos ordenados con la finalidad de poder detectar las posiciones que no contienen los elementos del conjunto inicial a ordenar.

2.- Seleccionar el primer elemento del conjunto de información a ordenar, para aplicarle la función hash.

3.- Aplicar la función hash, para determinar la posición dentro de la estructura matricial del elemento seleccionado.

4.- Si la posición esta vacía, colocar ahí al elemento; si está ocupada, buscar la siguiente posición vacía dentro del subconjunto referido por la función hash, para colocar al elemento.

5.- Si aún hay elementos del conjunto inicial por ordenar, entonces, seleccionar el siguiente elemento, y repetir desde el paso número 3 al número 5.

6.- Si ya no hay más elementos del conjunto inicial por ordenar, entonces, reordenar los subconjuntos de la estructura matricial para que sus elementos se encuentren en orden.

5.3 ALGORITMO.

```
Desde i = 1 hasta n
    Desde j = 1 hasta n
        tabla(i, j)    =    100

Seleccionar primer elemento a ordenar

Mientras aún existan elementos en el conjunto inicial por order
hacer:
{ aplicar la función hash }
    pos      =  elemento / 10;

    { Determinar su posición en la sublista }
    Desde k = 1 hasta n
        Si es una posición vacía
            tabla_hash(pos, k)     =      elemento

    Seleccionar siguiente elemento a ordenar
```

Ordenar_tabla_hash
{ Ordenar las diferentes sublistas }

5.4 PROGRAMA FUENTE DE LA PRIMERA VERSION.

/ Programa Hash, el cual ordenará 10 números. Para ello, se generarán 10 números aleatorios diferentes, los cuales se dividirán sucesivamente entre 100 hasta que queden números de dos dígitos (menores a 100). La tabla constará de 100 posiciones, colocándose al número de dos dígitos en la posición que le corresponde dividiéndolo entre 10 y tomando el cociente. Si ya esta ocupada la primera posición, se colocará en la posición vacía de la sublista que le toca. */*

```
# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10                 /* Máximo número de elementos
                                       de la tabla */
# define CONS_DOS_NUM 100 /* Constante para dividir
                             sucesivamente el número
                             aleatorio para obtener un
                             número de dos dígitos (menor a
                             100) */
# define CONS_DIV 10               /* Constante para la función
                                      hash */


main()
{
 int tabla_hash[NUM_ELE][NUM_ELE]; /* Tabla hash de elementos
                                      a ordenar */
 int i,j;                          /* Indices para recorrer la
                                      tabla */
 int elemento;                     /* Tipo de elementos de la
                                      tabla hash */
```

```
time_t semilla;                          /* Variable para generar
                                            diferentes números
                                            aleatorios */
int numero_dos_dígitos();                /* Función para convertir los
                                            números aleatorios en
                                            números de dos dígitos */
void hash();                             /* Función Hash */
void ordenar_tabla_hash();               /* Función para ordenar los
                                            elementos de la tabla
                                            hash */

for (i = 0; i < NUM_ELE; i++)
 for (j = 0; j < NUM_ELE; j++)
   tabla_hash[i][j] = 100;  /* Como siempre se usarán números
                               menores a 100, se inicializa la
                               tabla con dicho número para indicar
                               que todas las posiciones están
                               vacías */
semilla = time(NULL);
if (&semilla == NULL)  /* Si no se pudo obtener la fecha del
                          sistema, salir */
  printf("\n\nError al generar los números aleatorios, intentar de
        nuevo\n");
else
  {
   srand(semilla);
   printf("\n\nLos elementos a ordenar son:\n");
   for (i = 0; i < NUM_ELE; i++)
   {
    printf("%d ",elemento=numero_dos_dígitos(rand()));/* Se
                                                        selecciona
                                                        un entero
                                                        de forma
                                                        aleatoria,
                                                        convirtién-
                                                        dolo a un
                                                        número de
                                                        dos
                                                        dígitos */
    hash(elemento, tabla_hash);
```

```
    }
}


ordenar_tabla_hash(tabla_hash);
printf("\n\nLos elementos ordenados son:\n");
for (i = 0; i < NUM_ELE; i++)
   for (j = 0; j < NUM_ELE; j++)
      if ( tabla_hash[i][j] > 99)  /* Si se llegó al fin de la
                                      sublista, salir */
       break;
      else
       printf("%d ", tabla_hash[i][j]);
printf("\n");
}


int numero_dos_dígitos(num_ale)

 /* Se divide el número aleatorio entre el 100 y se toma el
    residuo. Si es menor a 100 se regresa, sino, se sigue
    dividiendo entre 100 hasta obtener un número menor a 100 */

 int num_ale;  /* Número aleatorio a convertir */

{
 while (num_ale >= CONS_DOS_NUM)
   num_ale = num_ale % CONS_DOS_NUM;
 return num_ale;
}


void hash(elem, tab_hash)

 int elem;                        /* Elemento al que se le va a aplicar
                                     la función hash */
 int tab_hash[][NUM_ELE];  /* Tabla de hash */

{
 int k;          /* Indice para recorrer la tabla */
```

```
int pos;  /* Número de la sublista a la que pertenece el
             elemento */

pos = elem / CONS_DIV;  /* función hash para encontrar la
                           posición del número en alguna de las
                           sublistas que van del 0 al 9 */
for (k = 0; k < NUM_ELE; k++) /* Se recorre la sublista hasta
                                 encontrar una posición vacía
                                 (que tenga 100), y se coloca ahí
                                 el número */
  if (tab_hash[pos][k] > 99)
   {
    tab_hash[pos][k] = elem;
    break;
   }
}


void ordenar_tabla_hash(tab_hash)

 /* Como cada sublista a lo mucho tiene 10 elementos, se usa el
    ordenamiento de selección directa para ordenar cada
    sublista. */


int tab_hash[][NUM_ELE]; /* Tabla de hash */

{
int l,m,n;          /* Indices para recorrer la tabla */
int temp;           /* Variable para realizar intercambio de
                       elementos */
int clave_menor;    /* Variable para almacenar la posición del menor
                       elemento encontrado hasta el momento */
for (l = 0; l < NUM_ELE; l++)
  for (m = 0; m < NUM_ELE - 1; m++)
    if (tab_hash[l][m] > 99)  /* Si se llegó al fin de la
                                 sublista, salir */
      break;
    else
     {
```

```
    clave_menor = m;
    for (n = m + 1; n < NUM_ELE; n++)

     if (tab_hash[l][n] > 99)  /* Si se llegó al fin de la
                                  sublista, salir */
       break;
     else
       if (tab_hash[l][n] < tab_hash[l][clave_menor])
         clave_menor = n;
    if (clave_menor != m)
    {
     temp = tab_hash[l][m];
     tab_hash[l][m] = tab_hash[l][clave_menor];
     tab_hash[l][clave_menor] = temp;
    }
    }
}
```

5.4.1 Análisis del programa

El programa se divide en cuatro partes: crear e inicializar la tabla de hash, convertir cada uno de los elementos a números de dos dígitos, aplicarle a cada uno la función hash, y después ordenar cada una de las sublistas resultantes.

En el programa principal se define la tabla hash como una estructura de 100 elementos, 10 renglones por 10 columnas. Esto es porque las 10 filas representan uno de los posibles 10 dígitos (0 a 9) con que empieza el primer dígito del número, teniéndo capacidad para diez elementos que empiecen con el mismo dígito, si es que se da el caso de que los diez elementos posean esta característica y evitar quedarnos sin dónde almacenar elementos.

Todos los elementos de la tabla se inicializan con 100 para indicar que se encuentran vacíos, porque estamos partiendo del hecho que los elementos válidos tendrán valores de 0 a 99.

La función *numero_dos_dígitos* asegura que únicamente se utilizarán enteros de dos dígitos como elementos a ordenar en la tabla. Esto se

debe a que la función *rand* para generar números aleatorios, regresa elementos que van desde 0 hasta el entero más grande que soporte la máquina, y que puede ser un número igual o mayor a 100. Es por ello que el número generado por rand se divide sucesivamente entre 100 y se toma el residuo, hasta que se tenga un número menor a 100, y es el que se regresa como elemento a ordenar para la tabla hash.

La función *hash* se utiliza para determinar en que posición de la tabla de hash se debe colocar el elemento actual. La función consiste en dividir el elemento entre 10 y tomar el cociente, el cual nos da la posición de la subtabla (0 a 9) en la que debe colocarse el elemento. Posteriormente, se empieza a recorrer dicha sublista hasta que se encuentra un elemento que tenga un 100, indicando que dicha posición esta vacía, y se inserta ahí al nuevo elemento.

Por último, la función *ordena_tabla_hash* ordena cada una de las diez sublistas de la tabla hash, aplicándoles el algoritmo de Selección Directa. La condición *if(tabla_hash[i][j] > 99* prueba si se llegó a una posición vacía de la sublista para no revisar hasta el final.

Este método a diferencia de los demás, ilustra claramente el cambiar tiempo por espacio, ya que el determinar la posición en donde se debe colocar el elemento es mucho más rápida, pero se tiene que reservar mucho espacio para evitar colisiones, que en nuestro caso es nueve veces más del que se requiere, ya que se tienen 100 posiciones para almacenar únicamente 10 elementos.

5.5 PROGRAMA FUENTE DE LA SEGUNDA VERSION.

```
/* Programa Hash, el cual ordenará 10 números. Para ello, se
   generarán 10 números aleatorios diferentes, los cuales se
   dividirán sucesivamente entre 100 hasta que queden números de
   dos dígitos (menores a 100). La tabla constará de 100
   posiciones, colocándose al número de dos dígitos en la posición
   que le corresponde dividiéndolo entre 10 y tomando el cociente.
   Si ya esta ocupada la primera posición, se colocará en la
```

*posición vacía de la sublista que le toca. */*

```
# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# define NUM_ELE 10                /* Máximo número de elementos de la
                                      tabla */
# define CONS_DOS_NUM 100  /* Constante para dividir
                              sucesivamente el número aleatorio
                              para obtener un número de dos
                              dígitos (menor a100) */
# define CONS_DIV 10               /* Constante para la función hash */

main()
{
 int *tabla_hash[NUM_ELE][NUM_ELE]; /* Tabla hash de elementos
                                       a ordenar */
 int i,j;                          /* Indices para recorrer la
                                      tabla */
 int elemento;                     /* Tipo de elementos de la
                                      tabla hash */
 time_t semilla;                   /* Variable para generar
                                      diferentes números
                                      aleatorios */
 int numero_dos_dígitos();         /* Función para convertir los
                                      números aleatorios en
                                      números de dos dígitos */
 void hash();                      /* Función Hash */
 void ordenar_tabla_hash();        /* Función para ordenar los
                                      elementos de la tabla
                                      hash */

 for (i = 0; i < NUM_ELE; i++)
  for (j = 0; j < NUM_ELE; j++)
   {
    tabla_hash[i][j] = malloc(sizeof(int)); /* Asignar memoria a
                                               la tabla */
    *tabla_hash[i][j] = 100; /* Como siempre se usarán números
                                menores a 100, se inicializa la
```

```
                                    tabla con dicho número para
                                    indicar que todas las posiciones
                                    están vacías */
  }
semilla = time(NULL);
if (&semilla == NULL)  /* Si no se pudo obtener la fecha del
                          sistema, salir */
  printf("\n\nError al generar los números aleatorios, intentar de
        nuevo\n");
else
  {
   srand(semilla);
   printf("\n\nLos elementos a ordenar son:\n");
   for (i = 0; i < NUM_ELE; i++)
   {
    printf("%d ",elemento=numero_dos_dígitos(rand())); /* Se
                                                          selecciona
                                                          un entero de
                                                          forma
                                                          aleatoria,
                                                          convirtién-
                                                          dolo a un
                                                          número de
                                                          dos
                                                          dígitos */
    hash(elemento, tabla_hash);
   }
}
ordenar_tabla_hash(tabla_hash);
printf("\n\nLos elementos ordenados son:\n");
   for (i = 0; i < NUM_ELE; i++)
     for (j = 0; j < NUM_ELE; j++)
       if (*tabla_hash[i][j] > 99)  /* Si se llegó al fin de la
                                       sublista, salir */
        break;
        else
        printf("%d ", *tabla_hash[i][j]);
printf("\n");
for (i = 0; i < NUM_ELE; i++)
  for (j = 0; j < NUM_ELE; j++)  /* Liberar el espacio ocupado
```

```
                                        por el arreglo */
      free(tabla_hash[i][j]);
}


int numero_dos_dígitos(num_ale)

  /* Se divide el número aleatorio entre el 100 y se toma el
     residuo. Si es menor a 100 se regresa, sino, se sigue dividiendo
     entre 100 hasta obtener un número menor a 100 */

  int num_ale; /* Número aleatorio a convertir */

{
 while (num_ale >= CONS_DOS_NUM)
   num_ale = num_ale % CONS_DOS_NUM;
 return num_ale;
}


void hash(elem, tab_hash)

  int elem;                    /* Elemento al que se le va a aplicar
                                  la función hash */
  int *tab_hash[][NUM_ELE];   /* Tabla de hash */

{
 int k;       /* Indice para recorrer el arreglo */
 int pos;     /* Número de la sublista a la que pertenece el
                 elemento */

 pos = elem / CONS_DIV;  /* Función hash para encontrar la posición
                            del número en alguna de las sublistas
                            que van del 0 al 9 */
 for (k = 0; k < NUM_ELE; k++)  /* Se recorre la sublista hasta
                                   encontrar una posición vacía
                                   (que tenga 100), y se coloca ahí
                                   el número */
   if (*tab_hash[pos][k] > 99)
     {
```

```
    *tab_hash[pos][k] = elem;
    break;
   }
}


void ordenar_tabla_hash(tab_hash)

 /* Como cada sublista a lo mucho tiene 10 elementos, se usa el
    ordenamiento de selección directa para ordenar cada sublista. */


int *tab_hash[][NUM_ELE]; /* Tabla de hash */

{
 int l,m,n;          /* Indices para recorrer la tabla */
 int *temp;          /* Variable para realizar intercambio de
                        elementos */
 int clave_menor;    /* Variable para almacenar la posición  del
                        menor elemento encontrado hasta el momento */

 for (l = 0; l < NUM_ELE; l++)
   for (m = 0; m < NUM_ELE - 1; m++)
     if (*tab_hash[l][m] > 99)  /* Si se llegó al fin de la
                                   sublista, salir */
       break;
     else
       {
       clave_menor = m;
       for (n = m + 1; n < NUM_ELE; n++)
         if (*tab_hash[l][n] > 99)  /* Si se llegó al fin de la
                                       sublista, salir */
           break;
         else
           if (*tab_hash[l][n] < *tab_hash[l][clave_menor])
             clave_menor = n;
       if (clave_menor != m)
       {
        temp = tab_hash[l][m];
        tab_hash[l][m] = tab_hash[l][clave_menor];
```

```
        tab_hash[l][clave_menor] = temp;
      }
    }
}
```

5.6 COMPARACION ENTRE LAS DOS VERSIONES.

La diferencia entre las dos versiones es la utilización de apuntadores, la cual reduce el tiempo de intercambio de los elementos a ordenar, al únicamente mover apuntadores, pero se requiere de memoria adicional para los mismos.

5.7 ANALISIS DE EFICIENCIA.

En cuanto al tiempo de ejecución requerido para realizar el método de ordenamiento con **Cálculo de Dirección** se tiene que "Si los n elementos originales están aproximadamente uniformemente distribuidos en los subarchivos m, el valor de n/m es aproximadamente 1, el tiempo para el ordenamiento es del orden $O(n)$, puesto que la función asigna cada elemento al archivo apropiado y se requiere muy poco trabajo extra para colocar el elemento en el subarchivo requerido. Por otro lado, si n/m es mayor que 1, o si el archivo original no esta distribuido uniformemente sobre los m subarchivos, se requiere mucho trabajo adicional para insertar un elemento en el subarchivo apropiado y el tiempo por consiguiente es del orden $O(n^2)$." [TEN85, pág. 422].

5.8 DETERMINACION DE LAS CONDICIONES EN LAS CUALES EL ALGORITMO PRESENTA UN COMPORTAMIENTO EFICIENTE.

La eficiencia del algoritmo **Con Cálculo de Dirección** depende de la distribución que guarden los n elementos del conjunto de información a ordenar sobre los m subconjuntos, ya que entre más uniforme sea la distribución, la velocidad de crecimiento de ejecución tiende a $O(n)$, sin embargo en tanto se reduce esta uniformidad se reduce también la eficiencia, tendiendo a un tiempo de ejecución proporcional a $O(n^2)$. Por lo tanto, el algoritmo en cuestión es el más eficiente analizado hasta ahora, mientras se cuente con la distribución uniforme mencionada.

CONCLUSIONES

CONCLUSIONES

La exposición realizada acerca del análisis de la eficiencia de los algoritmos de ordenamiento mediante la identificación de criterios acerca del conjunto a ordenar, tales como el número de elementos y la distribución de los mismos (que tan ordenados y desordenados estén), permite seleccionar la aplicación de un determinado algoritmo en la solución de un problema de ordenamiento. No obstante lo anterior, para elegir un algoritmo no es posible guiarse exclusivamente por los criterios mencionados, pues tienen lugar otras consideraciones, tales como:

- el compilador
- tiempo disponible para resolver el problema
- número de veces por utilizar el algoritmo
- velocidad del procesador utilizado en la ejecución del programa
- capacidad de almacenamiento
- etc.;

que interactúan y afectan el desempeño en la implantación de un algoritmo, por lo que la selección del mismo será más eficiente en tanto más se conozcan las consideraciones anteriores. Sin embargo, el desarrollo de la presente tesis está referido a llevar a cabo un análisis acerca del comportamiento de la velocidad de crecimiento en el tiempo de ejecución del algoritmo, en función al número de elementos a ordenar, para contar con una medida de eficiencia y poder hacer una selección más objetiva.

Los algoritmos de ordenamiento cuyo análisis ha sido expuesto van desde un tiempo de ejecución proporcional a $O(n^2)$ hasta $O(n\log n)$. De los cuatro algoritmos estudiados en la presente tesis, se tiene lo siguiente:

- Selección Directa: es conveniente para entradas pequeñas, debido a que tanto en el caso promedio como en el peor caso el tiempo de ejecución del algoritmo es proporcional a $O(n^2)$, por lo que a mayor número de elementos a ordenar, el desempeño del algoritmo es más ineficiente. Debido al método utilizado en éste algoritmo, al

distribución del conjunto de elementos a ordenar no afecta su desempeño.

- Quicksort: por su parte, en el peor caso presenta un comportamiento en tiempo de ejecución proporcional a $O(n^2)$, pero en el caso promedio es muy eficiente, dado que tiende a $O(n\log n)$, por lo que si se conoce la distribución de la entrada, se puede determinar qué tan adecuado es este algoritmo.

- Árbol Binario: el desempeño de este algoritmo es similar al de Quicksort, ya que también depende de las características de la entrada y el número de elementos a ordenar. El tiempo de ejecución para el peor caso es proporcional a $O(n^2)$, pues el árbol resultante es una cadena, y lo obliga a un recorrido ineficiente; para el caso promedio, su tiempo de ejecución tiende a $O(n\log n)$. Sin embargo, tiene una ventaja con respecto a Quicksort, ya que por la estructura en que está implementado , permite la actualización de elementos de manera más eficiente.

- Con Cálculo de Dirección: el tiempo de ejecución en el caso promedio es proporcional $O(n)$, siendo en éste caso el más eficiente de los algoritmos expuestos en la presente tesis. Para ello, se requiere tener conocimiento de las características del conjunto a ordenar, ya que si la función hashing no realiza una distribución uniforme de los elementos, se tiende al peor caso, el cual es proporcional a $O(n^2)$.

De lo anterior se deduce que a excepción del algoritmo de Selección Directa, la elección de cualquiera de los demás algoritmos expuestos, depende de las características de la entrada, ya que su desempeño está en función de éstas.

Si no es el caso de que se tenga conocimiento de las características del conjunto a ordenar, entonces la selección sólo puede estar referida en función al número de elementos a ordenar.

Aún cuando esta última sea la situación, al conocer el comportamiento de la función tiempo de ejecución de los algoritmos de ordenamiento, se determinará si es conveniente su aplicación al problema particular, ya que para entradas pequeñas los algoritmos cuyo tiempo de ejecución es proporcional a $O(n^2)$ son convenientes, pero a medida que el número de

algoritmos son más ineficientes, y es entonces cuando se debe optar por los algoritmos cuyo tiempo de ejecución en el peor caso tiende a n(logn), asegurando de esta manera un comportamiento eficiente para entradas lo suficientemente grandes.

Una vez que ya se ha seleccionado la aplicación de un determinado algoritmo de ordenamiento a un problema, entonces puede procederse a la implantación del algoritmo en un lenguaje de programación. En este aspecto existen dos criterios en función de los cuales un determinado algoritmo puede tener un mejor desempeño: tiempo y espacio, los cuales guardan siempre una relación inversamente proporcional. Los criterios anteriores pueden ser favorecidos por las estructuras de datos manejadas.

Debido al punto anterior, la decisión de intercambiar tiempo por espacio debe ir de acuerdo a las condiciones particulares en las que se encuentre el problema, considerando tanto la disponibilidad de tiempo y espacio que se tiene, y las repercusiones de incrementar uno o el otro.

Por último, cabe mencionar que en la bibliografía se puede encontrar mayores referencias hacia los temas presentados.

BIBLIOGRAFIA

ABELLANAS, M.; LODARES, D. [ABE91]
1991 Análisis de Algoritmos y Teoría de Grafos.
México: Macrobit Editores, S.A. de C.V.
189 pp.

AHO, Alfred et all [AHO88]
1988 Estructura de Datos y Algoritmos
México: Addison-Wesley/SITESA
438 pp.

ATKINSON, Lee; ATKINSON, Mark [ATK90]
1990 Using C
USA: Que Corporation (Programming Series)
945 pp.

BURK, Ron [BUR92]
1992 "Hashing from Good to Perfect"
The C Users Journal
USA: R & D Publications (Vol. 10, Num. 2, February 1992)
pp. 41-54

DAVIS, Wilbon [DAV92]
1992 "Time Complexity"
The C Users Journal
USA: R & D Publications (Vol. 10, Num. 9, September 1992)
pp. 29-38

HELLER, Steve [HEL91]
1991 "Making a Hash of Your Data"
Computer Language
USA: Miller Freeman Publications (Vol. 8, Num. 8, August 1991)
pp. 47-52

JAESCHKE, Rex [JAE91]
1991 "Standard C: a Status Report"
Dr. Dobb's Journal
USA: M & T Publishing, Inc. (Num. 179, August 1991)
pp. 16-26

KERNIGHAN, Brian; RITCHIE, Dennis [KER86]
1986 El Lenguaje de Programación C
México: Prentice Hall
235 pp.

KRUSE, Robert [KRU88]
1988 Estructura de Datos y Diseño de Programas
México: Prentice Hall
488 pp.

LIFFICK, Blaise (Editor) [LIF78]
1978 Program Desing
USA: Byte Publications (Programming Techniques, Vol. 1)
101 pp.

McTIERNAN, Kelly [MCT91]
1991 "An End to the Mystery of Sorting"
Computer Language
USA: Miller Freeman Publications (Vol 8, Num. 7, July 1991)
pp. 57-73

MOTTELER, Fred [MOT91]
1991 "Statistical Performance Analysis"
Dr. Dobb's Journal
USA: M & T Publishing, Inc. (Num. 183, December 1991)
pp. 68-76

OCHS, Tom [OCH90]
1990 "How Do I Count the Ways"
Computer Language
USA: Miller Freeman Publications (Vol. 7, Num. 12, December 1990)
pp. 91-100

PLAUGER, P.J [PLA92a]
1992 "All Sorts of Sorts"
Computer Language
USA: Miller Freeman Publications (Vol 9, Num. 1, January 1991)
pp. 21-30

PLAUGER, P.J. [PLA92b]
1992 "Standard C, the Header <stdlib.h>"
The C Users Journal
USA: R & D Publications (Vol. 10, Num. 4, April 1992)
pp. 8-20

QUINTERO ZAZUETA, Ricardo [QUI83]
1983 Gráficas y Algoritmos
México: Limusa Wiley
50 pp.

SEDGEWICK, Robert [SED90]
1990 Algorithms in C
USA: Addison-Wesley Publishing Co.
654 pp.

TENENBAUN, Aaron; AUGENSTEIN, Moshe [TEN85]
1985 Estructura de Datos en Pascal
México: Prentice Hall
560 pp.