
Estructuras de Datos y Algoritmos

**Código de Curso: CY330
Versión 2.2**

Volumen 2:

Estructuras de Datos Avanzadas

Unidad 1:

Grafos

Objetivos del Aprendizaje

- Definir grafos dirigidos y no dirigidos
- Explicar las propiedades de los grafos no dirigidos
- Definir términos asociados con grafos
- Discutir la representación de un grafo como un conjunto, una matriz de adyacencia y una lista de adyacencia
- Describir las aplicaciones de los grafos

Introducción a Grafos

- Los grafos, como estructuras matemáticas, se usan en diferentes dominios:
 - Química
 - ingeniería electrónica
 - Informática
 - Sociología
 - Geografía

Ejemplos de Representaciones de Grafos

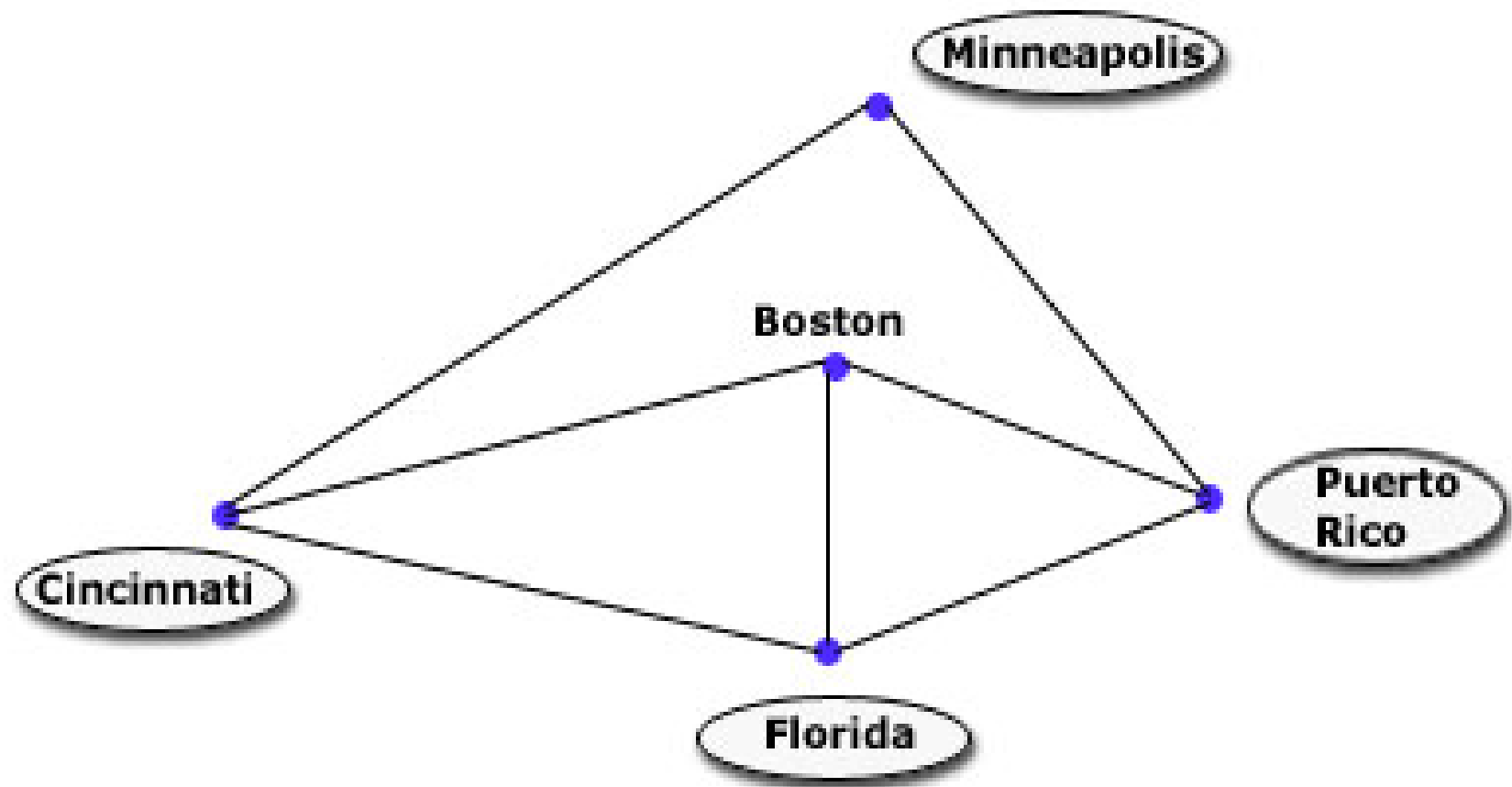
Se pueden representar mediante un grafo:

- Líneas ferroviarias, caminos o conexiones aéreas entre ciudades
- Los componentes y las conexiones entre estos en un circuito lógico digital
- Las líneas de autoridad y responsabilidad entre las personas en una compañía

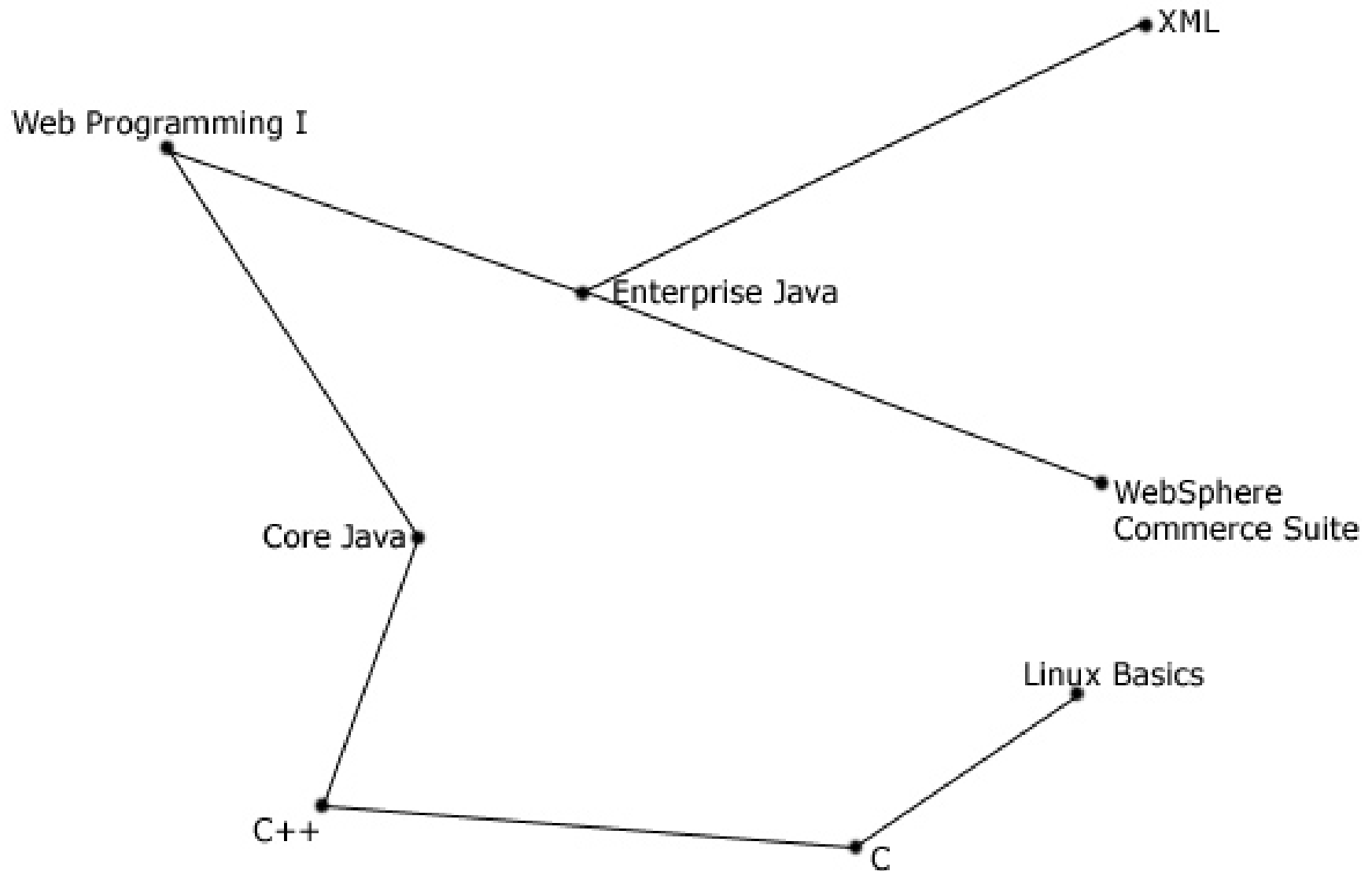
Preliminares de Grafos

- Un grafo está compuesto de un conjunto de vértices y un conjunto de aristas
- Los vértices y aristas son también llamados nodos y arcos respectivamente
- Una arista es un par de vértices dado que conecta dos vértices cualesquiera en un grafo

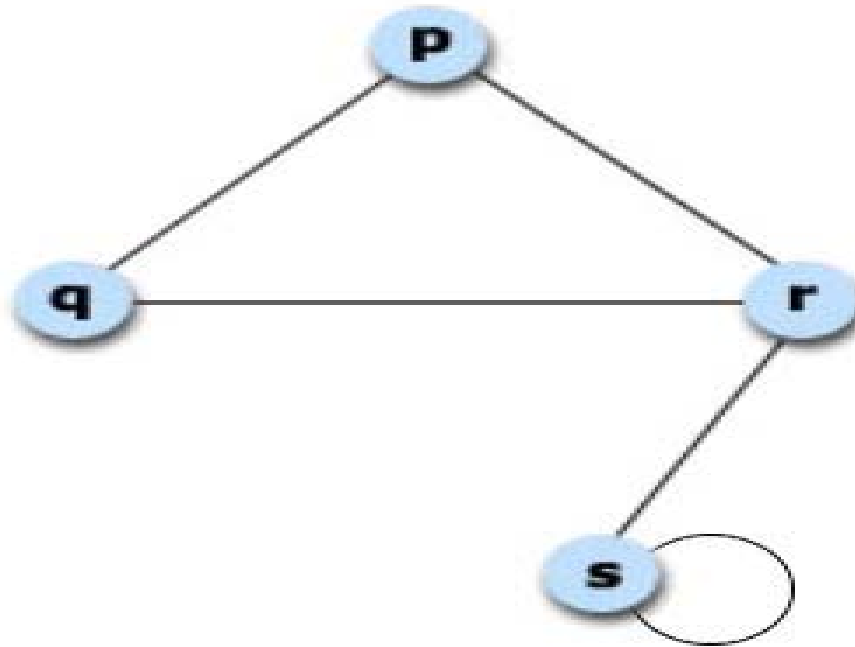
Ejemplo de Grafos – Rutas Aéreas Parciales



Ejemplo de Grafos – Prerrequisitos para Cursos



Aristas y Vértices



- Conjunto V , de vértices, donde $V = (p, q, r, s)$.
- Conjunto E , de aristas, donde $E = (pq, pr, qr, rs, ss)$
- Una arista se escribe simbólicamente como:
 $x = (p, q)$ donde x es una arista que conecta a los vértices p y q

Términos de Grafos

➤ Los términos comunes:

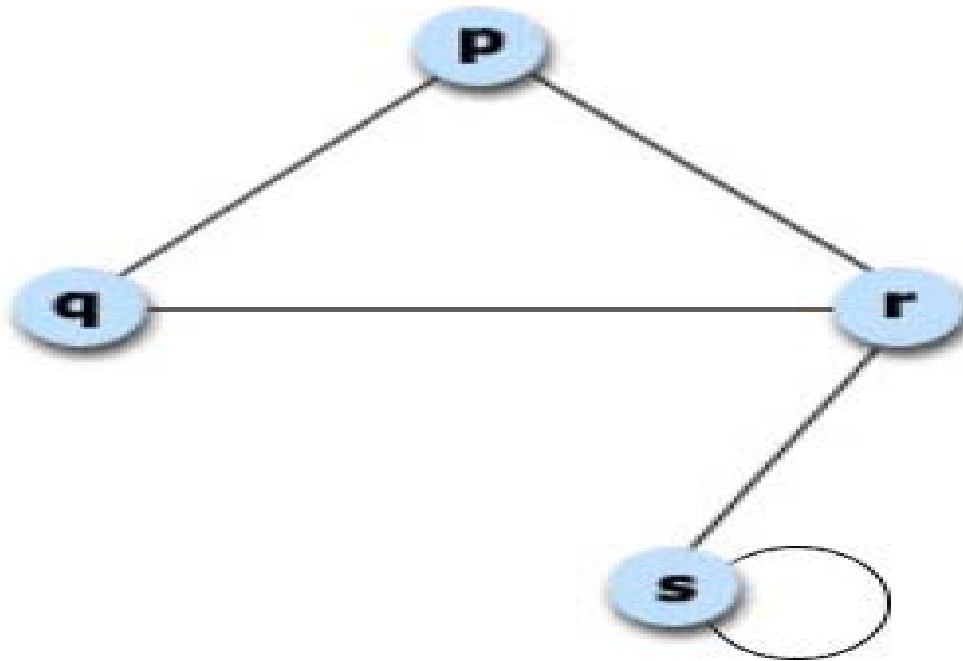
- Adyacente
- Incidente
- Yacer

➤ Considere la arista $x = (p, q)$

- El vértice p se dice que es **adyacente** al vértice q , dado que existe una arista entre estos dos vértices.
- La arista x es **incidente** con los vértices p y q , dado que conecta estos dos vértices.
- Los vértices p y q **yacen** en la arista x , dado que son conectados por esta arista

Grafos No Dirigidos

- Un **grafo no dirigido** se define como un grafo donde los pares de vértices no están ordenados



Grafos No Dirigidos...1

- **Adyacente :**

Si existe una arista entre dos vértices, por ejemplo, p y q , entonces son **vértices adyacentes**

- **Camino :**

Un **camino** es un conjunto que consiste de una secuencia de vértices adyacentes distintos

- **Ciclo:**

Un **ciclo** es un camino que contiene un número n de vértices, colocados de forma que el vértice 1 es adyacente al vértice 2, el vértice 2 es adyacente al vértice 3, el vértice $n-1$ es adyacente al vértice n , y finalmente el vértice n es adyacente al vértice 1

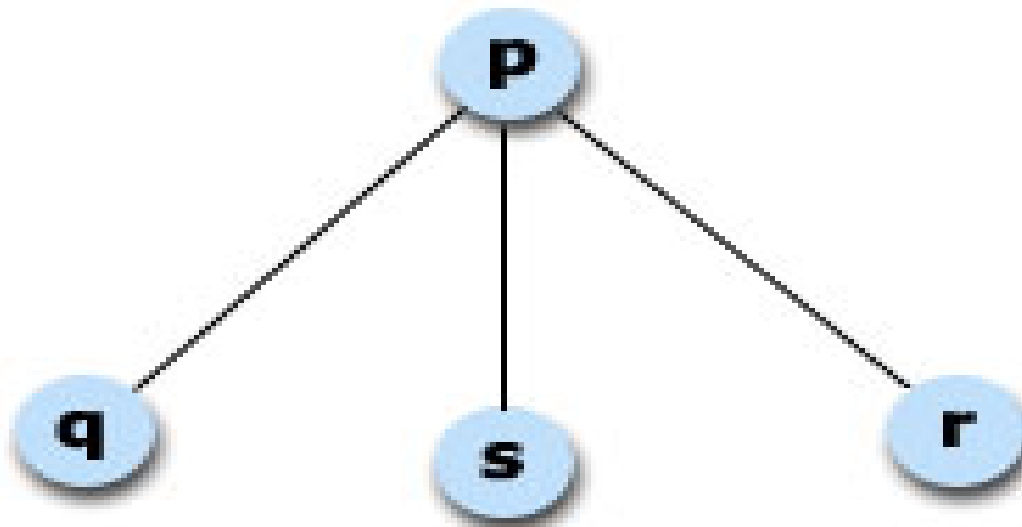
Grafos No Dirigidos ...2

- **Grafo Conectado:**

- Un **grafo conectado** es aquel donde existe un camino desde cada vértice en el grafo a cualquier otro vértice

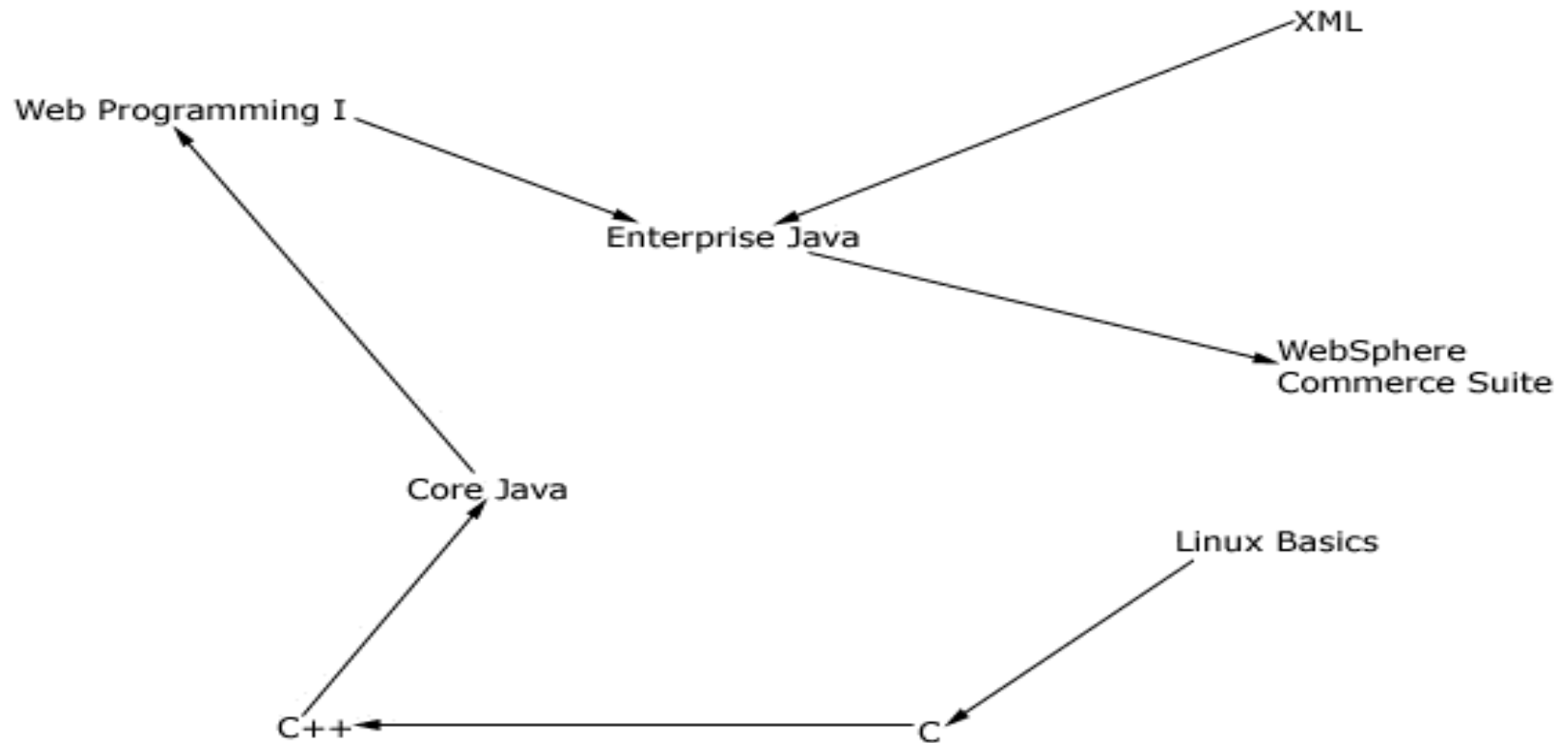
- **Árbol Libre:**

- Un **árbol libre** es un grafo no dirigido que no tiene ciclos.



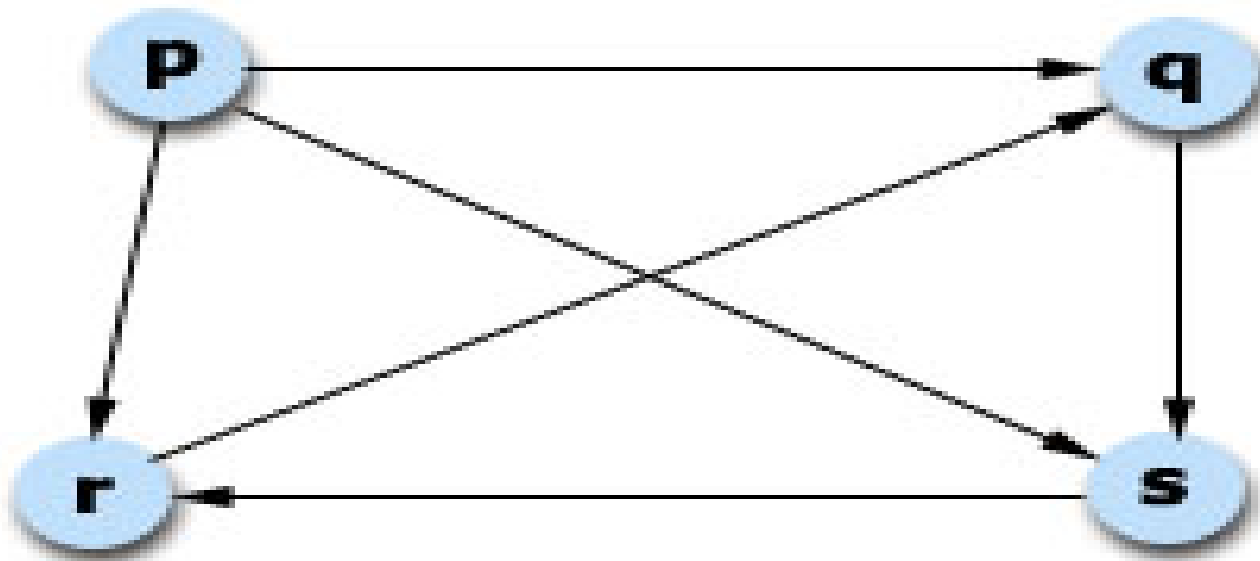
Grafos Dirigidos

- Un grafo dirigido se define como aquel donde los pares de vértices son ordenados
- Es un grafo con un arco representando las aristas y una flecha indicando la dirección



Propiedades de un Grafo Dirigido

- **Camino Dirigido:** En un grafo dirigido todas las aristas en un camino tienen la misma dirección, indicada por las flechas
- **Ciclos Dirigidos:** En un grafo dirigido, todas las aristas en un ciclo deben tener la misma dirección. Los ciclos en un grafo dirigido se denominan ciclos dirigidos



Propiedades de un Grafo Dirigido

Camino:

p, q

p, q, s

p, q, s, r

p, r, q, s

p, r, q, s, r

p, s

p, s, r

p, q, s, r, q

q, s

q, s, r

s, r

s, r, q

s, r, q, s

r, q, s, r

Ciclos:

q, s, r, q

r, q, s, r

s, r, q, s

Grafos y Estructuras de Datos

Los grafos se pueden representar mediante:

- **Representación de Conjunto:** Se usan uno o dos conjuntos para representar el grafo
- **Tablas de Adyacencia:** un grafo se puede representar como una tabla o una lista usando el concepto de adyacencia

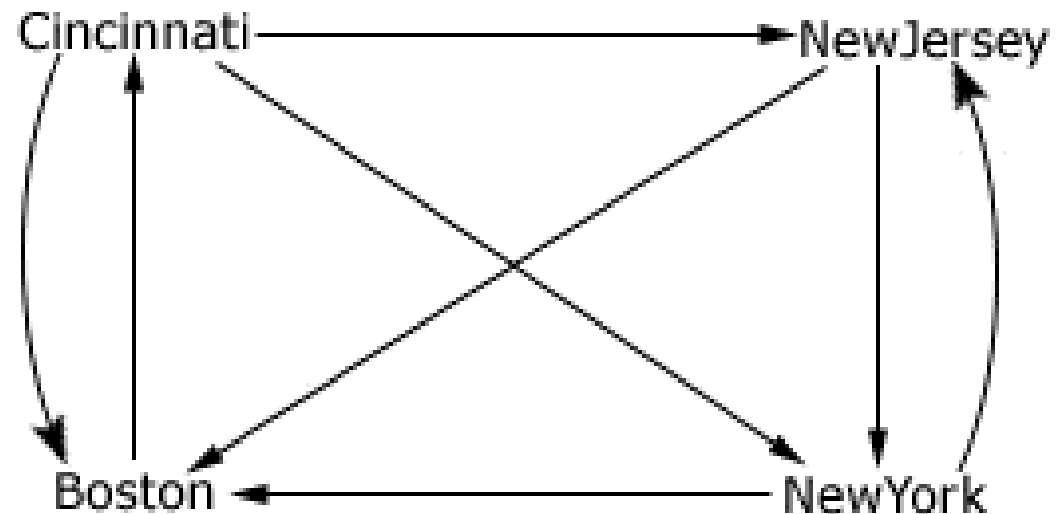
Representación de Grafos en Conjuntos

- Usando conjuntos se puede representar
 - Conjunto de vértices
 - Conjunto de pares de vértices que representan aristas
- Dos métodos para representar conjuntos
 - Mantener dos conjuntos diferentes
 - Mantener sólo un conjunto, que, en efecto, proporciona la misma información como la de dos conjuntos diferentes

Representación de Grafos en Conjuntos

Vértices

- Cincinnati
- New Jersey
- New York
- Boston



Aristas

- Cincinnati, New Jersey
- Cincinnati, New York
- Cincinnati, Boston
- New Jersey, New York
- New Jersey, Boston
- New York, New Jersey
- New York, Boston
- Boston, Cincinnati

Representación de Grafos usando Dos Conjuntos

➤ Se definen dos conjuntos:

- Conjunto de vértices
- Conjunto de aristas

➤ Operaciones:

- **Existencia de un Vértice:** Primero, se busca en el conjunto vértices para determinar si un vértice existe o no en el grafo
- **Existencia de un Camino:** Para encontrar un camino en un grafo se requieren dos vértices como entrada, un vértice origen y un vértice destino.
- Usando los valores de entrada y el conjunto aristas, se determina si un camino existe en el grafo

Representación de Grafos usando un Conjunto

- Se utilizan muchas instancias del conjunto **aristas**
- Para cada vértice se tiene un conjunto de aristas representado separadamente
- Ejemplo:
 - Vértice Cincinnati {Cincinnati-New Jersey, Cincinnati-New York, Cincinnati-Boston}
 - Vértice New Jersey {New Jersey-New York, New Jersey-Boston}
 - Vértice New York {New York-New Jersey, New York-Boston}
 - Vértice Boston {Boston-Cincinnati}

Representación de Grafos usando un Conjunto

- El conjunto contiene los vértices adyacentes a un vértice en el grafo
- Se mantienen instancias múltiples del conjunto **adyacente**, una para cada vértice.
- Ejemplo:
 - Conjunto adyacente para el vértice Cincinnati
{New Jersey, New York, Boston}
 - Conjunto adyacente para el vértice New Jersey
{New York, Boston}
 - Conjunto adyacente para el vértice New York
{New Jersey, Boston}
 - Conjunto adyacente para el vértice Boston
{Cincinnati}

Representación de Grafos usando Tablas de Adyacencia

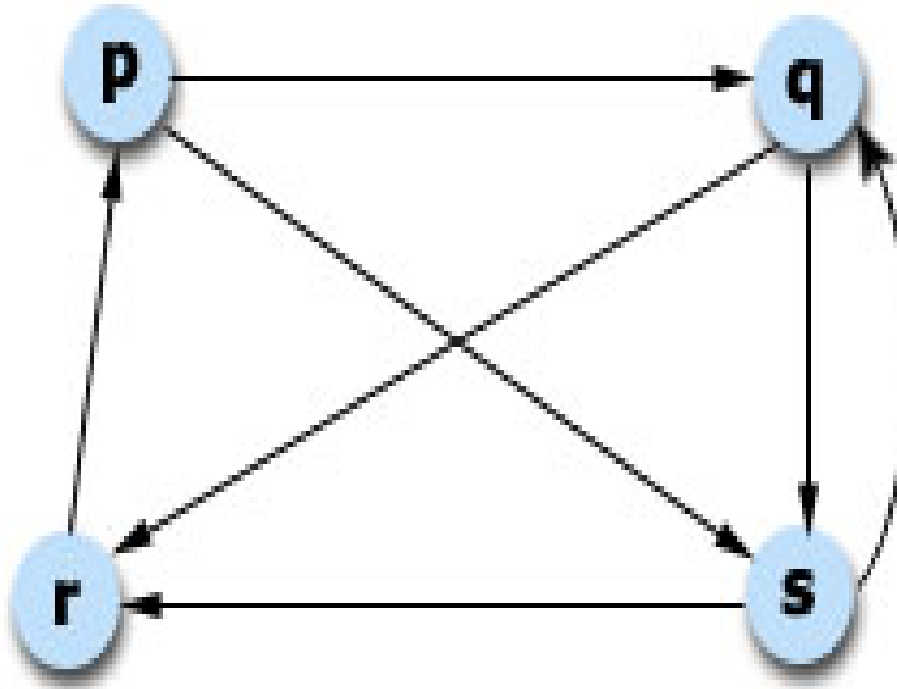
- Una matriz, llamada matriz de adyacencia, se usa para representar los datos acerca del grafo.

<u>Vértice</u>	Cincinnati	New Jersey	New York	Boston
Cincinnati	0	1	1	1
New Jersey	0	0	1	1
New York	0	1	0	1
Boston	1	0	0	0

Representación de Grafos usando Listas de Adyacencia

Vértices:

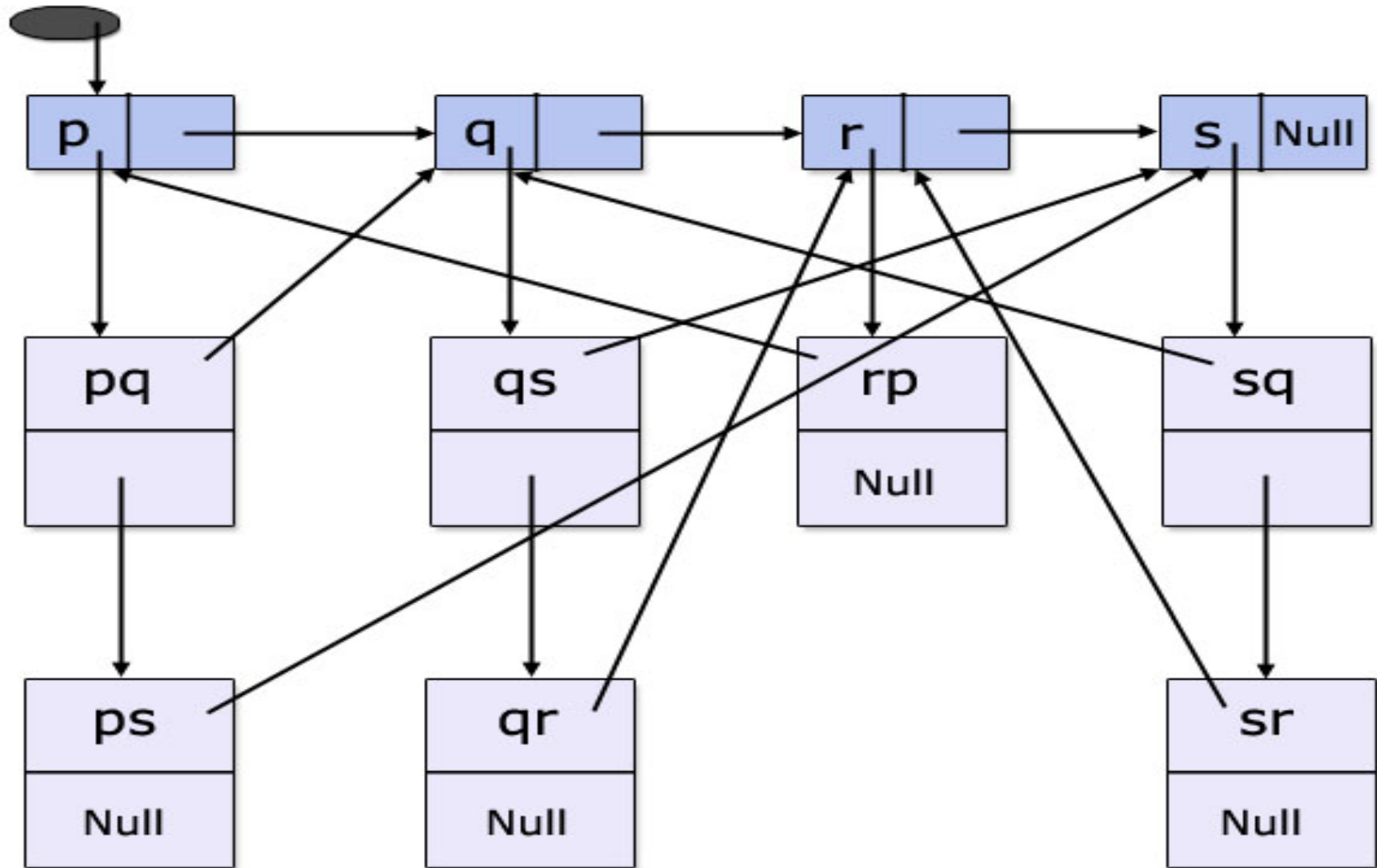
- p
- q
- r
- s



Aristas:

- pq
- ps
- qs
- qr
- sq
- sr
- rp

Listas de Adyacencia usando Listas Enlazadas



Listas de Adyacencia usando Listas Enlazadas

- Cada nodo en la lista enlazada (p , q , r , y s) tiene:
 - Un puntero al primer nodo de la lista enlazada, que representa las aristas en que participa el vértice.
 - Ejemplo: El nodo que contiene a p apunta a la lista enlazada que representa todas las aristas que empiezan con p , esto es, pq y ps .
 - Un puntero al siguiente nodo en su lista.
- Cada nodo en la lista enlazada que representa las aristas pq , qs , rp , y sq , tiene:
 - Un puntero al vértice final de la arista.
 - Ejemplo: El nodo que representa la arista pq apunta al vértice q .
 - Un puntero a la siguiente arista en su lista.

Recorrido de un Grafo

Dado el nodo de cabecera $graph$ y vértice p se puede:

- Recorrer la lista enlazada que representa los vértices.

Se visita p . Desde p se puede visitar q , desde q se puede visitar r , y desde r se puede visitar s .

- Obtener todas las aristas que empiezan desde p .

Se visita p . Desde p se puede visitar pq y desde pq se puede visitar ps .

- Recorrer un ciclo en el grafo dirigido.

Se visita p . Desde p se puede visitar pq . Desde pq se puede visitar q y desde q se puede visitar qr . Desde qr se puede visitar r y desde r se puede visitar rp . El primer vértice de pq es el vértice final de rp que se visita finalmente.

Aplicaciones de Grafos

- Representación de una Red de Caminos
- Diagramas de flujo
- Implementación de Autómatas
- Algoritmos de Grafos
- Problemas de Optimización a través de PERT/CPM
- Mapas y Bases de Datos Geográficas
- Diseño de Dispositivos de Integración a Gran Escala (Very Large Scale Integration - VLSI)

Aplicaciones de Grafos

- Reconocimiento de Caracteres y Correspondencia de Huellas Dactilares
- Estructuras Químicas
- Modelado Biológico
- Reconocimiento de Imágenes y Patrones
- Conceptos de Modelado Orientado a Objetos
- Laberintos y Detección de Rutas
- Detección de Semejanza

Resumen

- Se definió qué son grafos dirigidos y no dirigidos
- Se explicaron las propiedades de los grafos no dirigidos
- Se definieron los términos asociados con grafos
- Se discutió la representación de un grafo como un conjunto, una matriz de adyacencia y una lista de adyacencia
- Se dieron ejemplos de aplicaciones de los grafos

Unidad 2:

Árboles

Objetivos del Aprendizaje

- Definir un árbol como una estructura de datos
- Discutir acerca de árboles binarios, árboles de búsqueda binaria y árboles en general
- Explicar los tres métodos de recorrido de un árbol binario
- Definir el concepto un heap
- Distinguir entre un heap mínimo y un heap máximo

Introducción a los Árboles

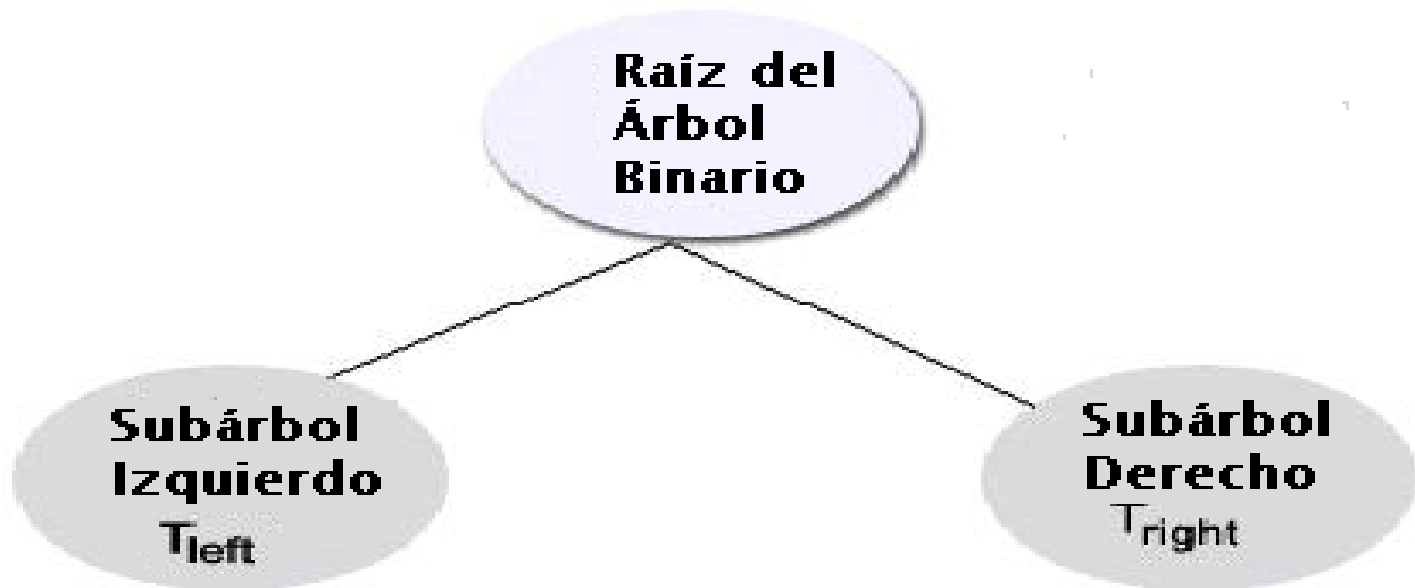
- Un árbol es una opción natural para representar ordenamiento jerárquico
- Un árbol se puede definir como una colección de nodos representando un ordenamiento jerárquico
- El ordenamiento jerárquico se puede basar en la información presente en los nodos
- Un ejemplo de ordenamiento jerárquico es la estructura de directorios del sistema operativo Linux

Introducción a los Árboles...1

- Los nodos son los elementos de un árbol y guardan información
- Los nodos pueden guardar información de cualquier tipo de datos
- Cada árbol tiene un nodo raíz
- Un nodo raíz es el primer nodo e indica el inicio de un árbol

Árboles Binarios

- Un **árbol binario** es un árbol en donde cada nodo puede tener cero, uno o dos hijos
- Cuando un nodo tiene uno ó dos hijos, los nodos hijos en sí mismos son árboles binarios
- Los hijos de un árbol binario también son conocidos como subárboles

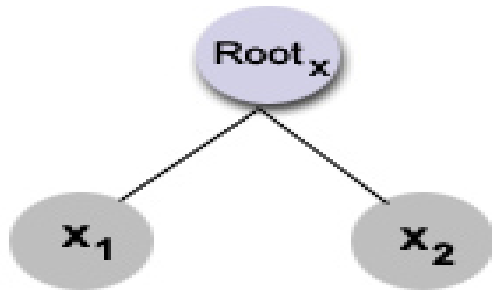


Árboles Binarios

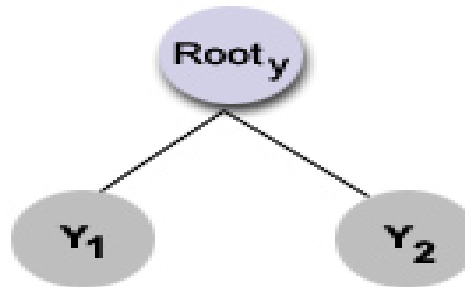
- Un árbol binario puede tener cero, uno o dos subárboles desde un nodo raíz
- Los subárboles pueden estar vacíos en un árbol binario
- Un árbol binario tiene las siguientes características:
 - Un árbol binario puede tener sólo un nodo raíz
 - Un árbol binario en el que no existe ningún nodo se denomina un árbol nulo.
 - Un árbol binario puede tener cero, uno o dos subárboles desde cualquier nodo del árbol

Unión de Árboles Binarios

Tree_x



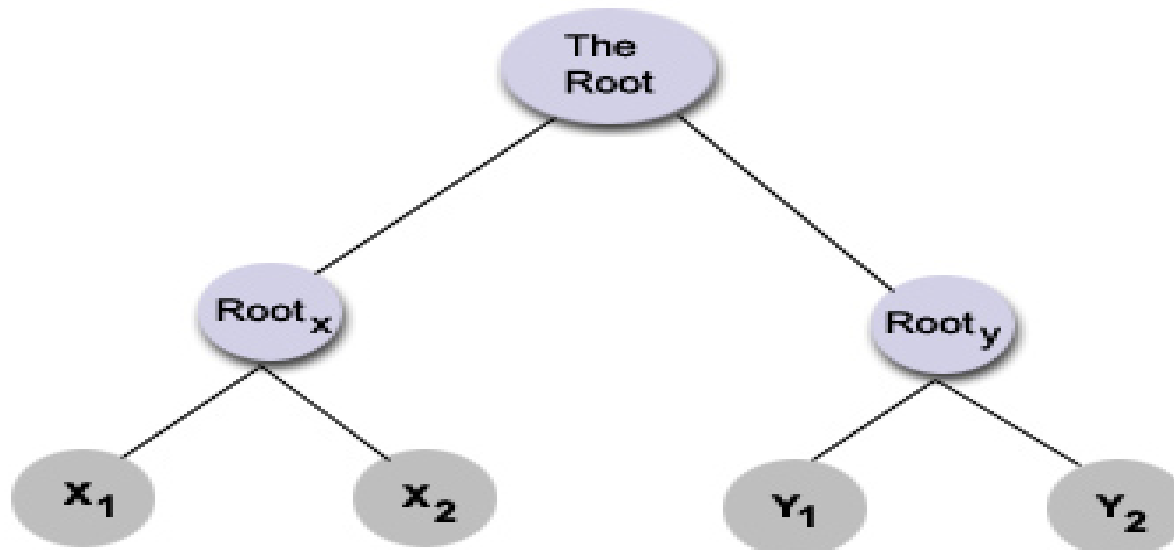
Tree_y



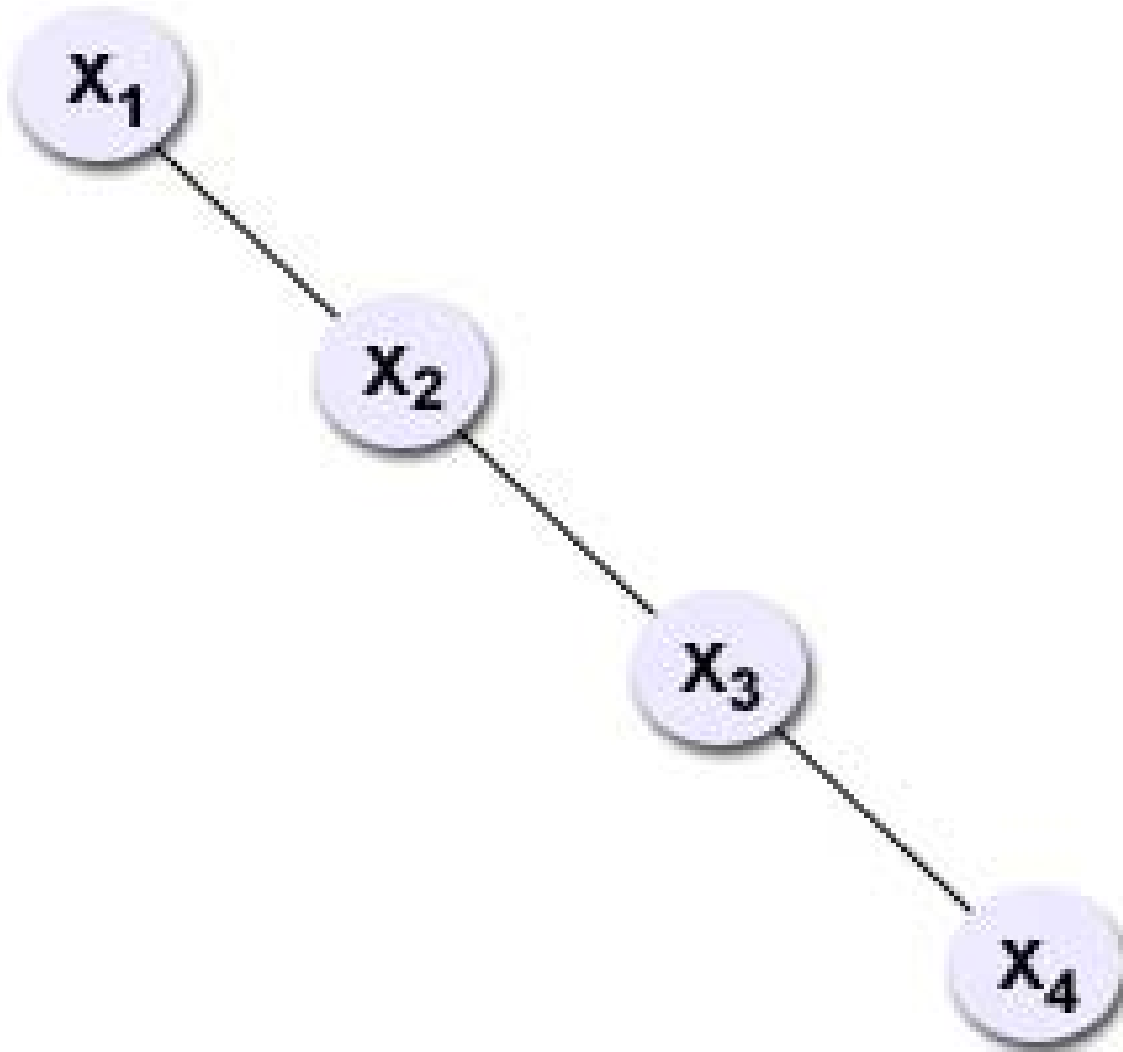
The Tree



Another Tree



Árbol Binario sin Subárbol Izquierdo



Terminologías de Árboles Binarios

- **Padre:** Es aquel nodo que tiene al menos un hijo
- **Hijo Izquierdo:** Es aquel nodo o hijo que se ramifica a la izquierda del un nodo padre
- **Hijo Derecho:** Es aquel nodo o hijo que se ramifica a la derecha del un nodo padre
- **Hoja:** Es aquel nodo sin hijos
- **Nivel de un Nodo:** Se refiere al nivel en que existe el nodo en el ordenamiento jerárquico.
 - Se denota por un número.
 - Al nodo raíz se le asigna 0. A los hijos de la raíz se les asigna 1.
 - Así, en un árbol, el nivel de cada nodo se representa por el nivel de su padre más uno.

Terminologías de Árboles Binarios...1

- **Arista:** Es la conexión entre el padre y sus hijos
- **Camino:** Es una secuencia de aristas consecutivas.
- **Longitud de Camino:** Es uno menos que el número de nodos en el camino. La longitud de un camino puede ser cero
- **Altura o Profundidad del Árbol:** Denota el número máximo de nodos desde la raíz hasta la menor hoja en un árbol.

Definición de un Árbol Binario en C

```
typedef int Tipo_elemento;
typedef struct Nodo_ArbolBinario {
    Tipo_elemento elemento;
    struct Nodo_ArbolBinario *izquierda;
    struct Nodo_ArbolBinario *derecha;
} Tipo_ArbolBinario;
```

Recorrido de Árboles Binarios

- El recorrido involucra visitar cada nodo en el árbol
- Los nodos pueden ser visitados usando cualquiera de éstos tres recorridos:
 - Recorrido Preorden
 - Recorrido Inorden
 - Recorrido Postorden
- Los nodos de un árbol binario se listan usando cualquiera de los métodos de recorrido

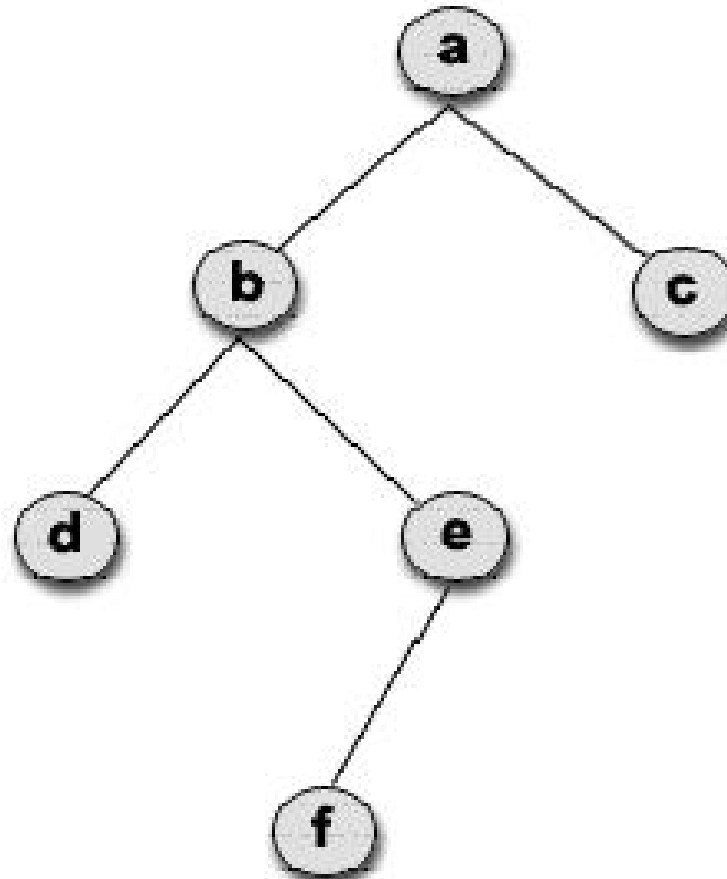
Recorrido de Árboles - Reglas

- **Recorrido Preorden**
 - Visitar el nodo raíz
 - Visitar el nodo izquierdo
 - Visitar el nodo derecho
- **Recorrido Inorden**
 - Visitar el nodo izquierdo
 - Visitar el nodo raíz
 - Visitar el nodo derecho
- **Recorrido Postorden**
 - Visitar el nodo izquierdo
 - Visitar el nodo derecho
 - Visitar el nodo raíz

Ejemplo de Árbol Binario

Inorder
izquierdo - raíz - derecho

Preorder
raíz - izquierdo - derecho



Postorder
izquierdo - derecho - raíz

Listado en PreOrden de un Árbol Binario

- raíz (a)** \longrightarrow Toma el valor - **a**
- izquierdo (b)** \longrightarrow Tiene un subárbol, guarda
valor para después
- raíz (b)** \longrightarrow Toma el valor - **b**
- izquierdo (d)** \longrightarrow No es un subárbol, toma
el valor - **d**
- derecho (e)** \longrightarrow Tiene un subárbol, guarda
valor para después
- raíz (e)** \longrightarrow Toma valor - **e**
- izquierdo (f)** \longrightarrow No es un subárbol, toma
el valor - **f**
- derecho (c)** \longrightarrow No es un subárbol, toma
el valor - **c**

Listado en Preorden: a b d e f c

Listado en InOrden de un Árbol Binario

- raíz (a)** \longrightarrow Tiene subárboles,
guarda valor para después
- izquierdo (b)** \longrightarrow Tiene subárboles,
guarda valor para después
- izquierdo (d)** \longrightarrow No tiene subárboles,
toma el valor – **d**
- raíz (b)** \longrightarrow Se mueve a su raíz (b),
toma el valor – **b**
- derecho (e)** \longrightarrow Tiene un subárbol,
guarda valor para después

Listado en InOrden de un Árbol Binario

izquierdo (f) → No tiene subárboles,
toma el valor - **f**

raíz (e) → Se mueve a su raíz (e),
sin subárbol derecho

Toma el valor - **e**

Se mueve a su raíz (b),
valor ya tomado

raíz (a) → Se mueve a su raíz
(a), toma el valor - **a**

derecho (c) → No tiene subárboles,
toma el valor- **c**

Listado Inorden: d b f e a c

Listado en PostOrden de un Árbol Binario

raíz (a) → Tiene subárboles, guardar valor para después

izquierdo (b) → Tiene subárboles, guardar valor para después

izquierdo (d) → No tiene subárboles,
tomar el valor - **d**
No puede tomarse valor de la raíz, pues el subárbol derecho no se ha recorrido aún

derecho (e) → Tiene un subárbol,
guardar valor para después

Listado en PostOrden de un Árbol Binario

izquierdo (f) → No tiene subárboles, toma el valor - **f**

Se mueve a su raíz (e), sin subárbol derecho

Toma el valor- **e**

Se mueve a su raíz (b), toma el valor - **b**

Se mueve a su raíz (a), tiene subárbol derecho

derecho (c) → No tiene subárboles, tomar el valor - **c**

Se mueve a su raíz (a), toma el valor - **a**

Listado en Postorden: d f e b c a

Código C para Listado en PreOrden

```
void preOrder(Tipo_ArbolBinario *nodo) {  
    if (nodo != NULL) {  
        printf("%d\n", nodo->elemento);  
        preOrder(nodo->izquierdo);  
        preOrder(nodo->derecho);  
    }  
}
```

Código C para Listado en InOrden

```
void inOrder(Tipo_ArbolBinario *nodo) {  
    if (nodo != NULL) {  
        inOrder(nodo->izquierdo);  
        printf("%d\n", nodo->elemento);  
        inOrder(nodo->derecho);  
    }  
}
```

Código C para Listado en PostOrden

```
void postOrder(Tipo_ArbolBinario *nodo) {  
    if (node != NULL) {  
        postOrder(nodo->izquierdo);  
        postOrder(nodo->derecho);  
        printf("%d\n", nodo->elemento);  
    }  
}
```

Árbol de Búsqueda Binaria

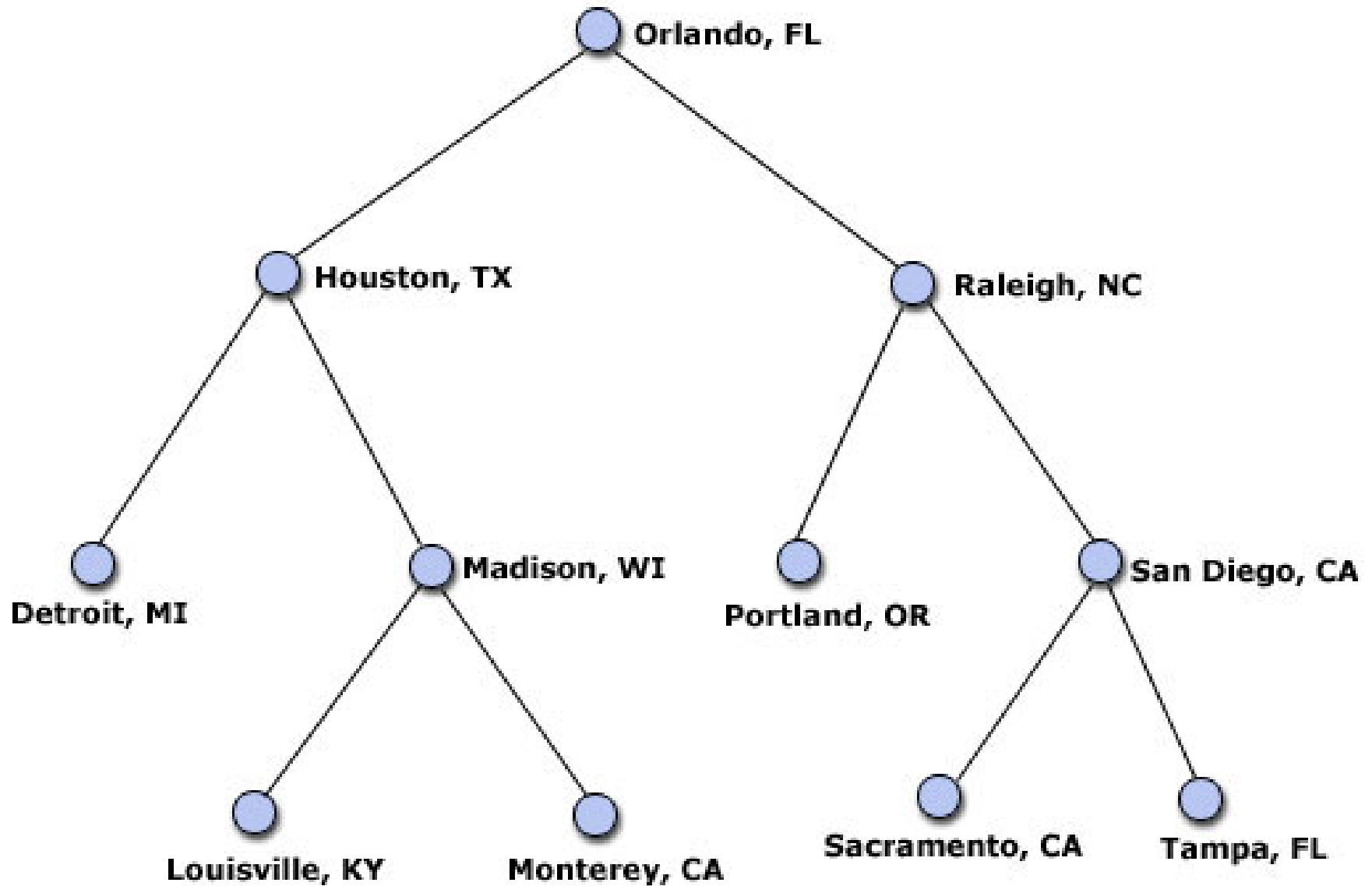
- Un **árbol de búsqueda binaria** es un árbol binario construido con el propósito de búsqueda, basada en el algoritmo de búsqueda binaria.
- Un árbol de búsqueda binaria es una forma especial de árbol binario .

Árbol de Búsqueda Binaria...1

➤ Características importantes de un árbol de búsqueda binaria:

- Si tiene un valor nulo para su raíz, entonces es un árbol de búsqueda binaria vacío.
- Todos los elementos que ocurren antes (menores que) del elemento en el nodo raíz están en el subárbol izquierdo.
- Todos los elementos que ocurren después (mayores que) el elemento en el nodo raíz están en el subárbol derecho
- El subárbol izquierdo también es un árbol de búsqueda binaria
- El subárbol derecho también es un árbol de búsqueda binaria

Árbol de Búsqueda Binaria - Ilustración



Búsqueda Usando Árbol de Búsqueda Binaria

Elemento de Búsqueda: 'Sacramento, CA'

- Se verifica el elemento buscado con el valor en la raíz, 'Orlando, FL'
- No es igual al valor en la raíz, se verifica si es menor, mayor o igual que 'Orlando, FL'
- El elemento buscado es mayor que el valor en el nodo raíz
- Nos movemos hacia el subárbol derecho de la raíz
- Ésta se convierte en la nueva raíz
- El elemento buscado se compara nuevamente con el valor en esta nueva raíz, 'Raleigh, NC'

Búsqueda usando Árbol de Búsqueda Binaria

- El elemento buscado no es igual a 'Raleigh, NC'
- También es mayor que 'Raleigh, NC'
- Por lo tanto, nos movemos a su subárbol derecho, que es 'San Diego, CA'.
- El elemento buscado no es igual al valor en la nueva raíz.
- Encontramos que es menor que 'San Diego'.
- Ahora nos movemos al subárbol izquierdo.
- El elemento buscado es igual al valor en esta nueva raíz, 'Sacramento, CA'.
- La búsqueda es un éxito

Árbol de Búsqueda Binaria – Lista Ordenada

Lista Inicial 1

Orlando, FL
Houston, TX
Raleigh, NC
Portland, OR
Madison, WI
Louisville, KY
San Diego, CA
Sacramento, CA
Monterey, CA
Detroit, MI
Tampa, FL



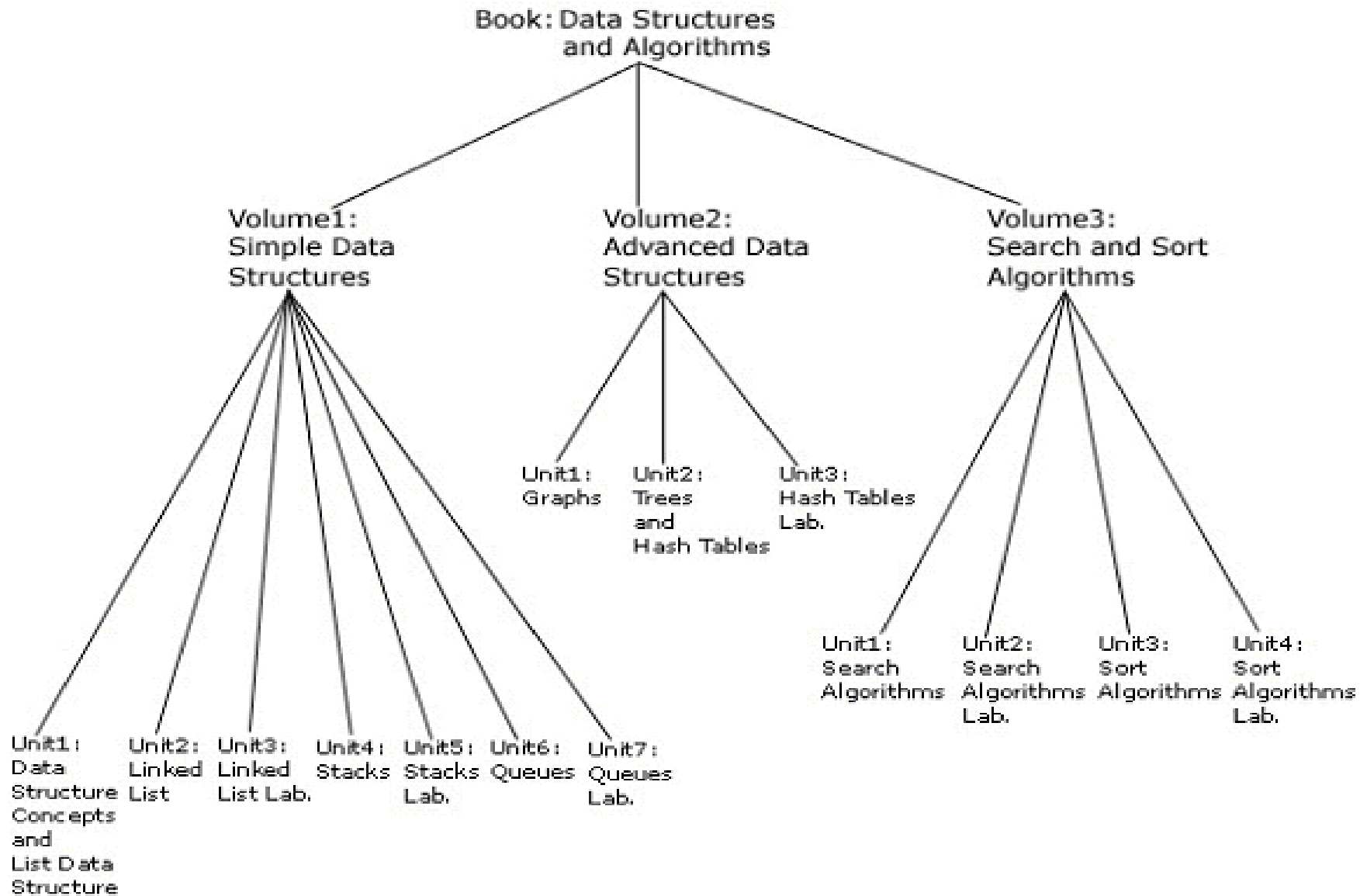
Lista Ordenada

Detroit, MI
Houston, TX
Louisville, KY
Madison, WI
Monterey, CA
Orlando, FL
Portland, OR
Raleigh, NC
Sacramento, CA
San Diego, CA
Tamp, FL

Árboles Generales

- Un árbol general es aquel en que un nodo puede tener más de dos subárboles
- Un ejemplo es el 'árbol genealógico'
- Las características de un árbol general son:
 - Puede tener sólo un nodo raíz
 - Es llamado un árbol nulo cuando no existen nodos
 - Puede tener cero, uno o más subárboles emanando desde cualquier nodo del árbol

Árboles Generales - Ejemplo



Aplicaciones de Árboles

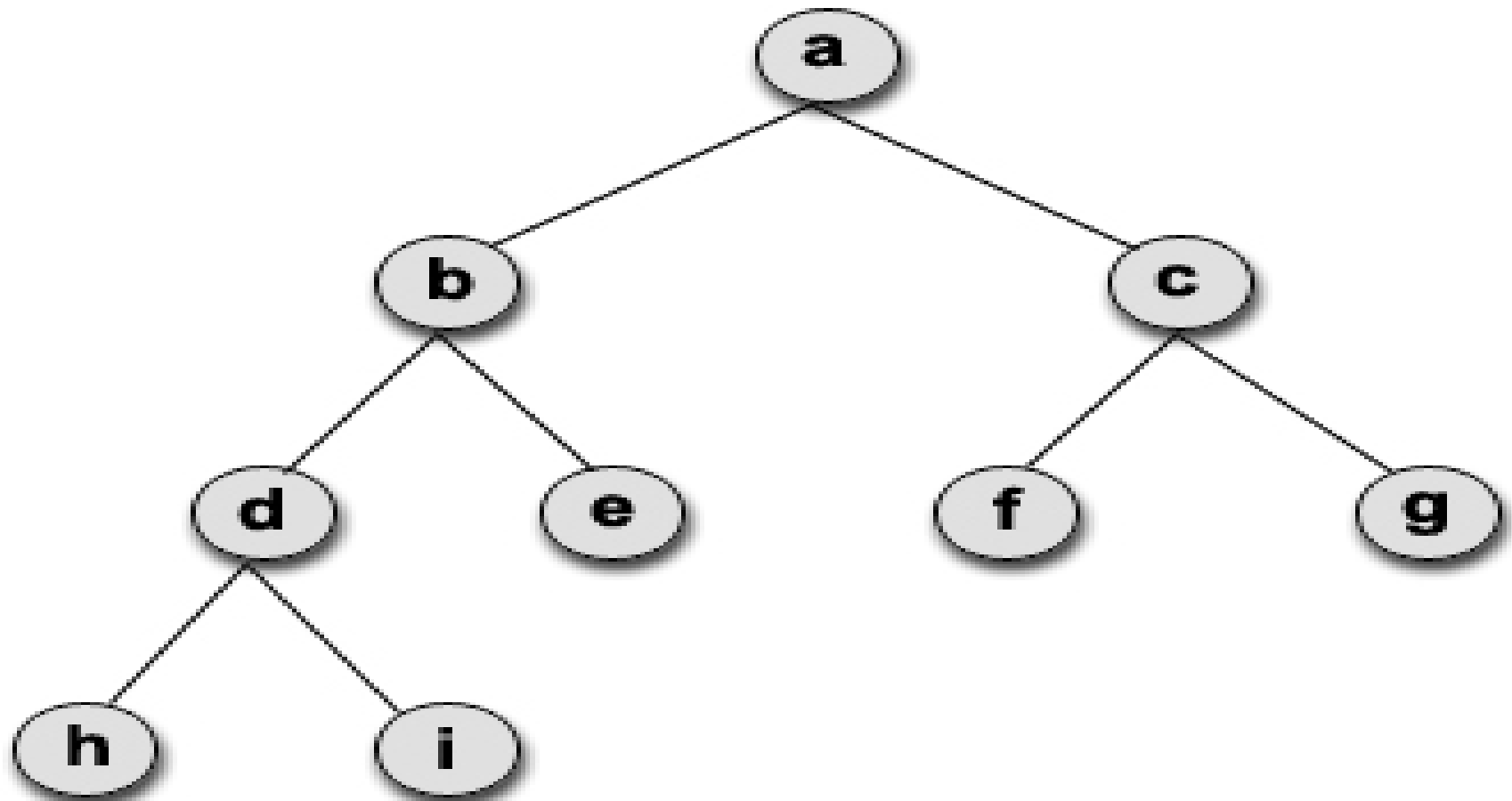
- Representación e Implementación de Expresiones Aritméticas
- Algoritmos de Búsqueda
- Representación e Implementación de Sistemas de Archivos
- Aplicaciones en Compiladores
- Procesamiento de Texto
- Compresión de Data
- Aplicaciones Genealógicas
- Árboles de Decisión para Juegos
- Representación de Relaciones Jerárquicas
- Aplicaciones en Bases de Datos
- Aplicaciones en Ciencias Biológicas y Bioinformática

Heaps

Un **heap** se define como un árbol binario que satisface las siguientes propiedades:

- Todas las hojas del árbol binario deben ocurrir en dos niveles adyacentes.
- Todas las hojas del menor nivel del árbol binario ocurren a la izquierda del árbol.
- Todos los niveles del árbol binario están completos, excepto en el caso del menor nivel del árbol, que puede estar sólo parcialmente completo.
- El elemento guardado en la raíz es mayor o igual que los elementos guardados en sus hijos. El árbol puede no tener hijos.
- Los subárboles izquierdo y derecho, que emanan de la raíz, son heaps en sí mismos.

Heap - Ejemplo

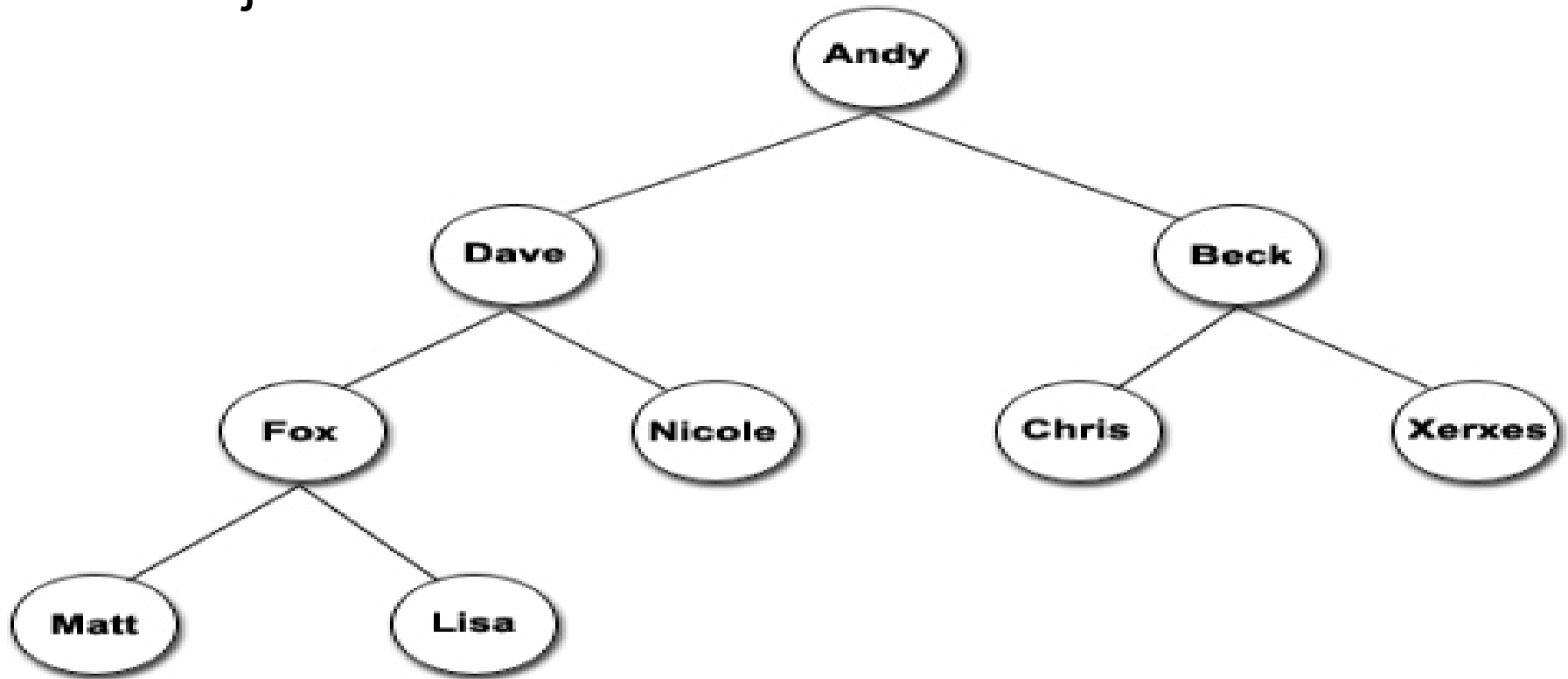


Heap

a	b	c	d	e	f	g	h	i
---	---	---	---	---	---	---	---	---

Heap Mínimo

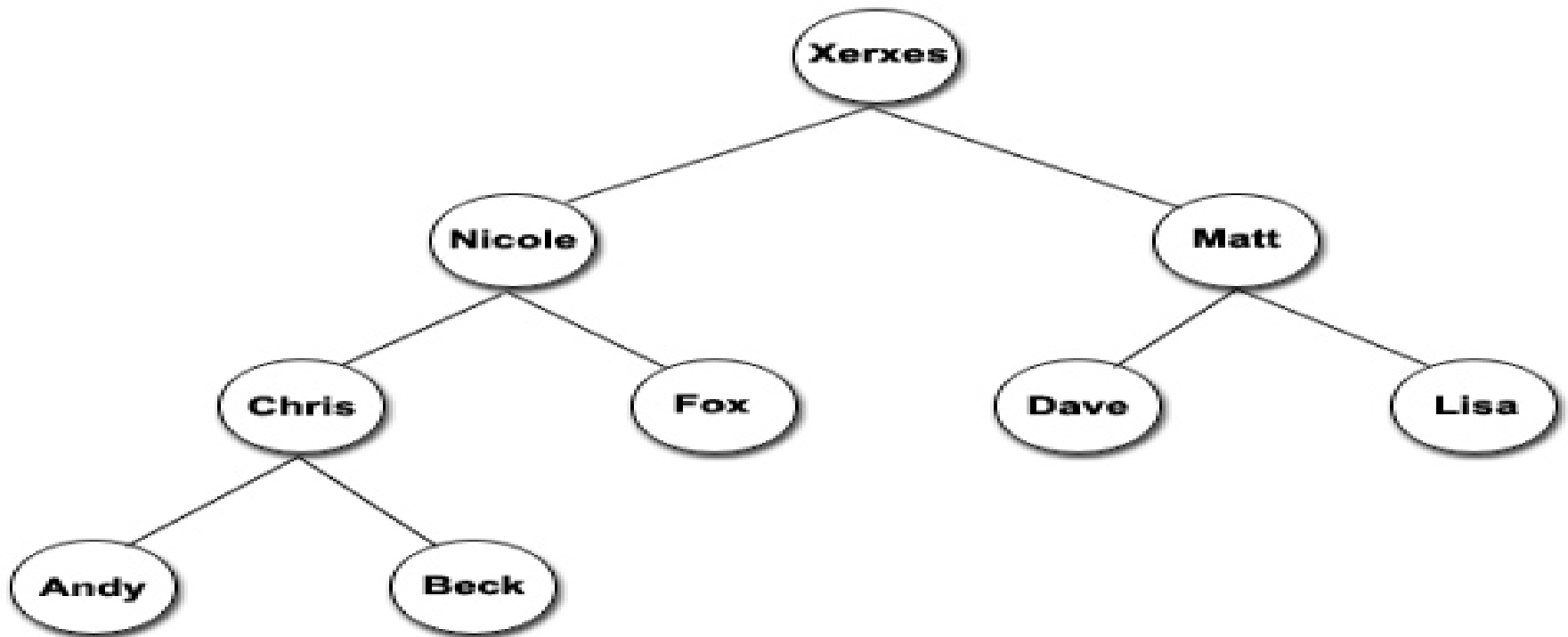
Un **Heap Mínimo** es un heap en el cual el elemento en la raíz es menor o igual que los elementos en sus hijos.



Andy	Dave	Beck	Fox	Nicole	Chris	Xerxes	Matt	Lisa
------	------	------	-----	--------	-------	--------	------	------

Heap Máximo

Un **Heap Máximo** es un heap en el cual el elemento en el nodo raíz es mayor o igual que el valor de los elementos en sus hijos



Xerxes	Nicole	Matt	Chris	Fox	Dave	Lisa	Andy	Beck
--------	--------	------	-------	-----	------	------	------	------

Cola de Prioridad

- Un heap se usa en la implementación de una cola de prioridad
- Una cola es llamada una cola de prioridad cuando la eliminación está basada ya sea en un elemento mínimo o máximo en la cola
- Una cola de prioridad tiene varias aplicaciones, que van desde sistemas operativos hasta aplicaciones de encolamiento en general
- Las colas de prioridad pueden usarse en planificadores de sistemas operativos donde las tareas esperando por el CPU son ordenados de acuerdo a prioridad
- El almacenamiento de eventos dependientes del tiempo emplea colas de prioridad

Resumen

- Se definió un árbol como una estructura de datos
- Se describieron los árboles binarios, árboles de búsqueda binaria y árboles generales
- Se explicaron los tres métodos de recorrido para un árbol binario
- Se definió el concepto de un heap
- Se diferenció entre un heap mínimo y un heap máximo

Unidad 3:

Técnicas Simples de Ordenamiento

Objetivos del Aprendizaje

- Proporcionar una visión general de las técnicas de ordenamiento
- Explicar el algoritmo de ordenamiento por inserción
- Explicar el algoritmo de ordenamiento por burbuja
- Explicar el algoritmo de ordenamiento por selección

Ordenar en la Vida Real - Ejemplos

- Ordenar libros y diarios en una biblioteca.
- Ordenar alfabéticamente nombres en un directorio telefónico.
- Listar palabras en un diccionario.
- Ordenar resultados de motores de búsqueda.

Categorías de Técnicas de Ordenamiento

- **Ordenamiento Interno**

Los elementos a ser ordenados están disponibles en la memoria primaria

- **Ordenamiento Externo**

Los elementos a ser ordenados son grandes y están disponibles sólo en dispositivos de almacenamiento secundario

Técnicas Simples de Ordenamiento

- Ordenamiento por Inserción
- Ordenamiento de la Burbuja
- Ordenamiento por Selección

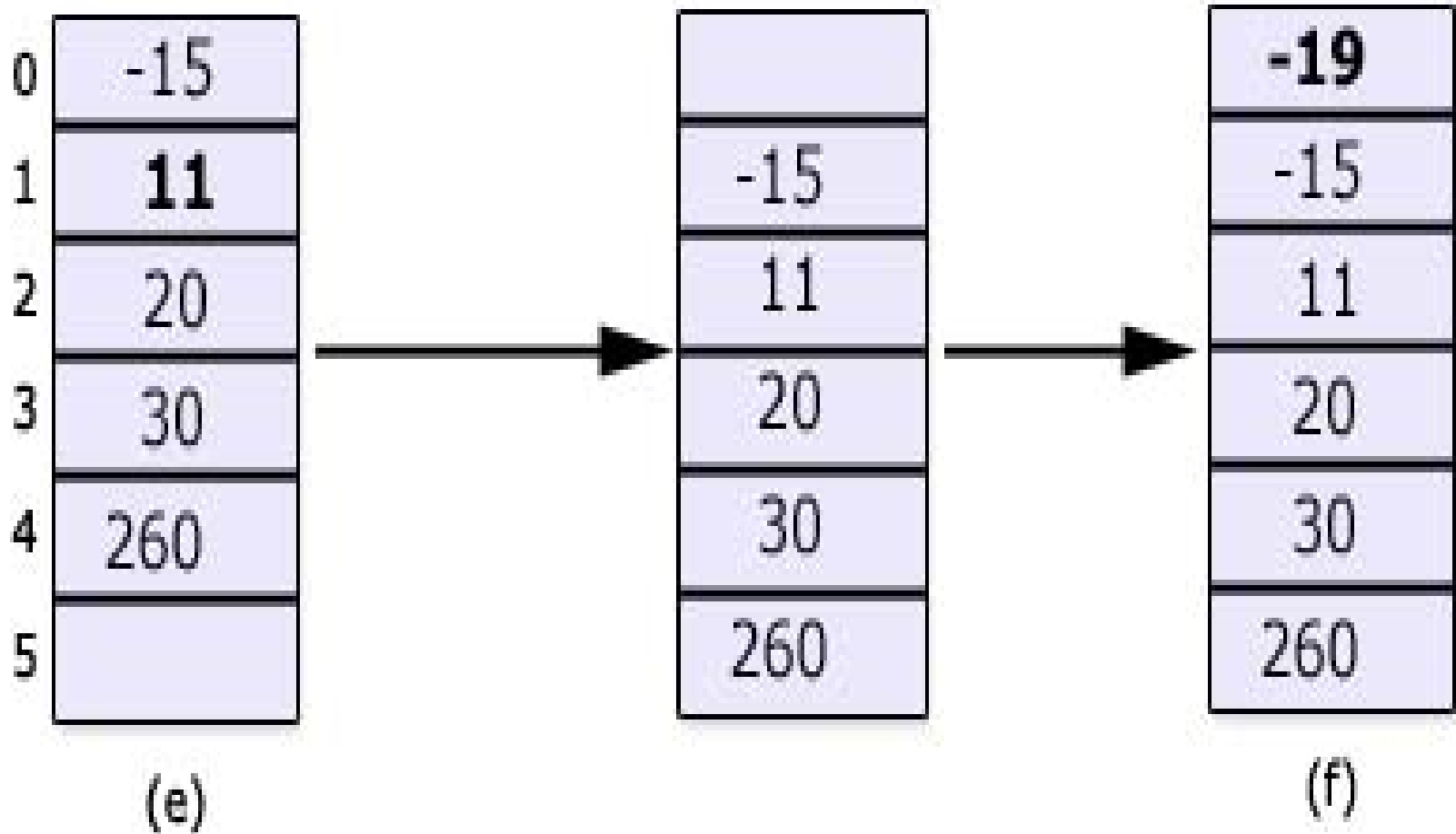
Ordenamiento por Inserción

- Entrada: Un arreglo de enteros en `InArray` a ser ordenados en orden ascendente
- Salida del arreglo ordenado: `OutArray`
- `InArray` permanece sin cambios durante el proceso de ordenamiento
- Escoger cada elemento de `InArray` de principio a fin
- Insertar cada elemento en un lugar apropiado en el `OutArray`
- La inserción se realiza de tal manera que en cualquier etapa `OutArray` está ordenado

Vista del Proceso de Ordenamiento por Inserción

0	260	260	-15	-15	-15	-15	-19
1	-15		260	20	20	11	-15
2	20		260	30	20	11	20
3	30			260	30	20	30
4	11				260	30	260
5	-19						
		(a)	(b)	(c)	(d)	(e)	(f)
	InArray	OutArray					

Vista de T durante un Paso de la Inserción



Programa para el Ordenamiento por Inserción

```
typedef int Tipo_elemento;
void insertionSort(Tipo_elemento *inArray, int
    n) {

    //Variable para guardar un valor temporalmente
    Tipo_elemento tempHolder;

    /* Declaración de contadores del ciclo */
    int i, j;

    /* Ciclo Externo: Recorrer el arreglo desde 1
       a n - 1 */
    for (i = 1; i < n; i++) {

        /* Guardar el elemento i-ésimo temporalmente */
        tempHolder = inArray[i];
```

Programa para el Ordenamiento por Inserción...1

```
/* Ciclo Interno: Elementos a ser desplazados
   hasta encontrar la posición del arreglo
   donde se puede insertar */
    j = i - 1;
    while(j >= 0 && tempHolder < inArray[j]) {
        inArray[j + 1] = inArray[j];
        j--;
    }

/* En este punto se puede insertar el valor
   guardado en tempHolder en la posición
   (j+1)ésimo del arreglo */
inArray[j + 1] = tempHolder;

/* Se tiene el arreglo ordenado */
}
```

Ordenamiento por Burbuja (Bubble Sort)

Los pasos son:

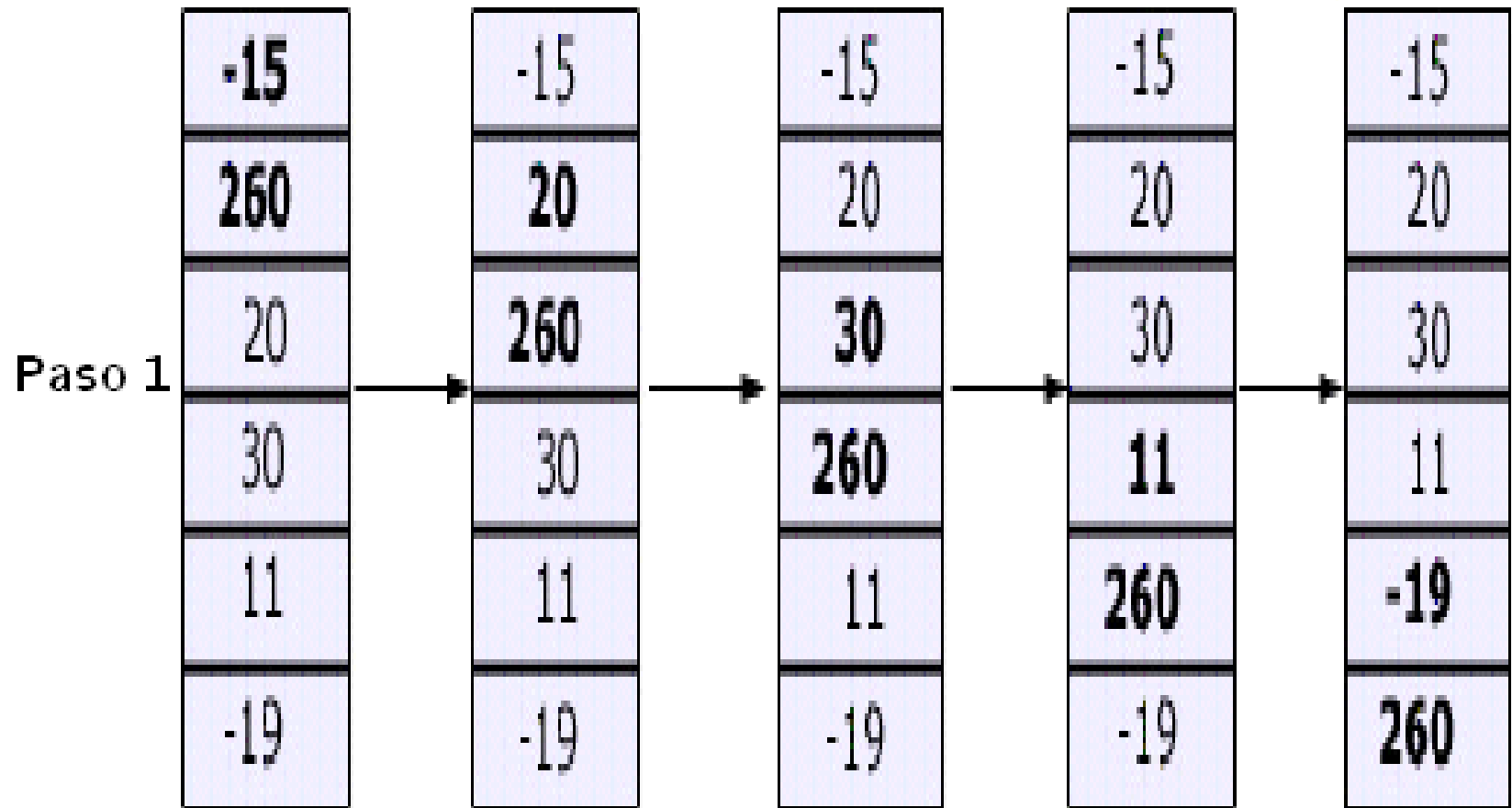
1. Entrada: Arreglo `InArray`
2. Empezar con el inicio del arreglo `InArray` con $k = 0$
3. Comparar dos elementos consecutivos de `InArray`, llamados, `InArray[k]` y `InArray[k+1]`
4. Si (`InArray[k] > InArray[k+1]`), intercambiar `InArray[k]` con `InArray[k+1]`
5. Avanzar al próximo elemento de `InArray` con $k++$.
Si $k < N-1$, entonces ir al paso 3
caso contrario ir al paso 6
6. Si ocurrió al menos un intercambio de elementos, volver al paso 2, caso contrario ir al paso 7
7. Fin del algoritmo

Vista del Proceso de Ordenamiento por Burbuja

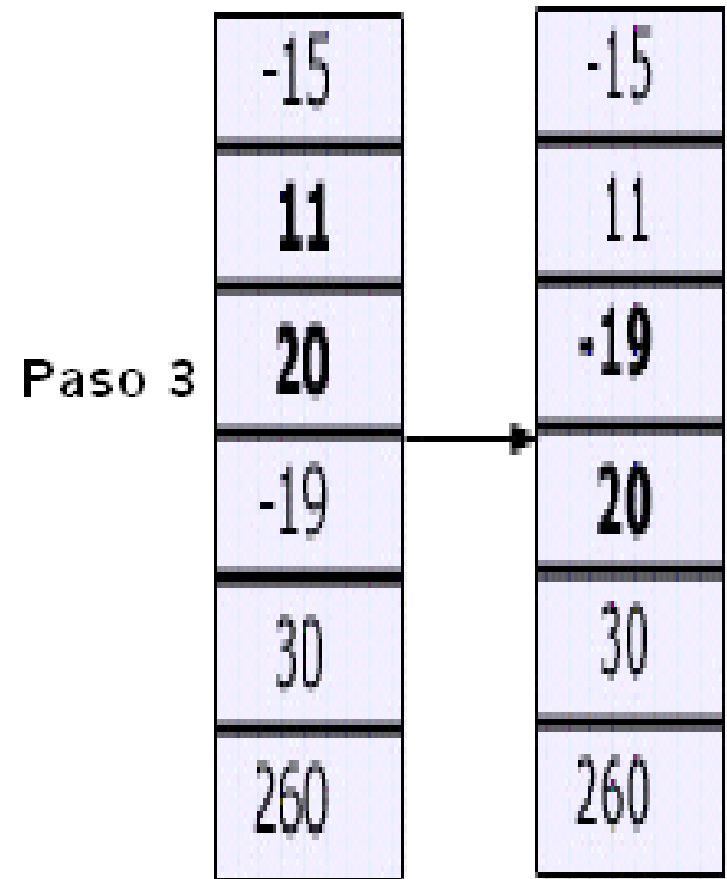
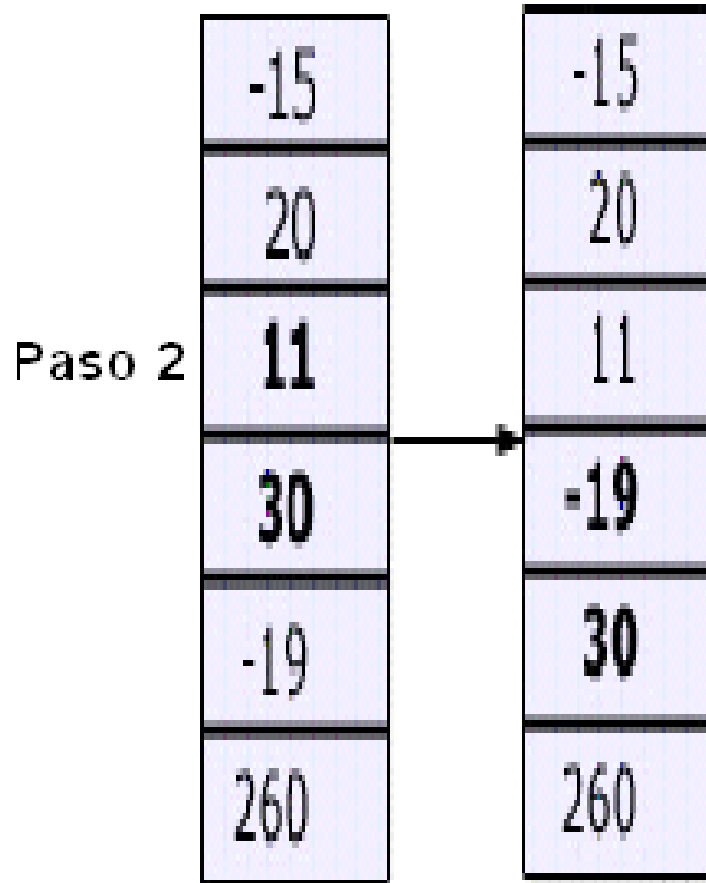
InArray

0	260
1	-15
2	20
3	30
4	11
5	-19

Vista del Proceso de Ordenamiento por Burbuja



Vista del Proceso de Ordenamiento por Burbuja



Vista del Proceso de Ordenamiento por Burbuja

Paso 4

-15
-19
11
20
30
260

Paso 5

-19
-15
11
20
30
260

Programa para el Ordenamiento por Burbuja

```
typedef int Tipo_elemento;
void bubbleSort(Tipo_elemento *inArray,
    Tipo_elemento n) {
    Tipo_elemento i, temp, interchange, j;

    interchange = 1;
    j = 1;
    while(interchange) {
        interchange = 0;
```

Programa para el Ordenamiento por Burbuja

```
for(i=0;i < n-j; i++) {  
    if(inArray[i] > inArray[i+1]) {  
  
        // Intercambiar los elementos  
        temp = inArray[i];  
        inArray[i] = inArray[i+1];  
        inArray[i+1] = temp;  
        interchange = 1;  
    }  
}  
j++;  
}
```

Ordenamiento por Selección

- Entrada: El arreglo `InArray` a ser ordenado
- Seleccione el elemento más pequeño en el arreglo
- Colóquelo en la primera posición, `InArray[0]`, intercambiando posiciones
- Busque el elemento siguiente más pequeño en el `InArray`, desde la posición 1 hacia adelante
- Asígnele a `pos` esa posición e intercambie `InArray[1]` con `InArray[pos]`
- Repita los pasos anteriores para $n-1$ pasos

Vista del Proceso de Ordenamiento por Selección

0	260	-19	-19	-19	-19	-19
1	-15	-15	-15	-15	-15	-15
2	20	20	20	11	11	11
3	30	30	30	30	20	20
4	11	11	11	20	30	30
5	-19	260	260	260	260	260
	InArray	(a)	(b)	(c)	(d)	(e)

Programa para el Ordenamiento por Selección

```
typedef int Tipo_elemento;
void selectionSort(Tipo_elemento *inArray,
    Tipo_elemento n) {

    Tipo_elemento i, pos, j, min, temp;

    for (i = 0; i <= n - 2; i++) {
        // Encontrar el más pequeño desde i hasta n-1
        min = inArray[i];
        pos = i;
```


Programa para el Ordenamiento por Selección

```
for (j = i + 1; j <= n - 1; j++)  
    if (inArray[j] < min) {  
        min = inArray[j];  
        pos = j;  
    }
```

// Mínimo encontrado en pos intercambiar

```
temp = inArray[i];  
inArray[i] = inArray[pos];  
inArray[pos] = temp;  
}  
}
```

Resumen

- Se presentó una visión general de las técnicas de ordenamiento
- Se explicó el algoritmo de ordenamiento por inserción
- Se explicó el algoritmo de ordenamiento por burbuja
- Se explicó el algoritmo de ordenamiento por selección

Unidad 4:

Laboratorio Técnicas Simples de Ordenamiento

Unidad 5:

Técnicas Avanzadas de Ordenamiento

Objetivos del Aprendizaje

- Explicar la técnica de merge sort (ordenamiento por fusión)
- Describir cómo escribir un algoritmo para desarrollar la técnica de ordenamiento de fusión
- Explicar la técnica de quicksort (ordenamiento rápido)
- Describir cómo escribir un algoritmo para desarrollar la técnica de ordenamiento rápido

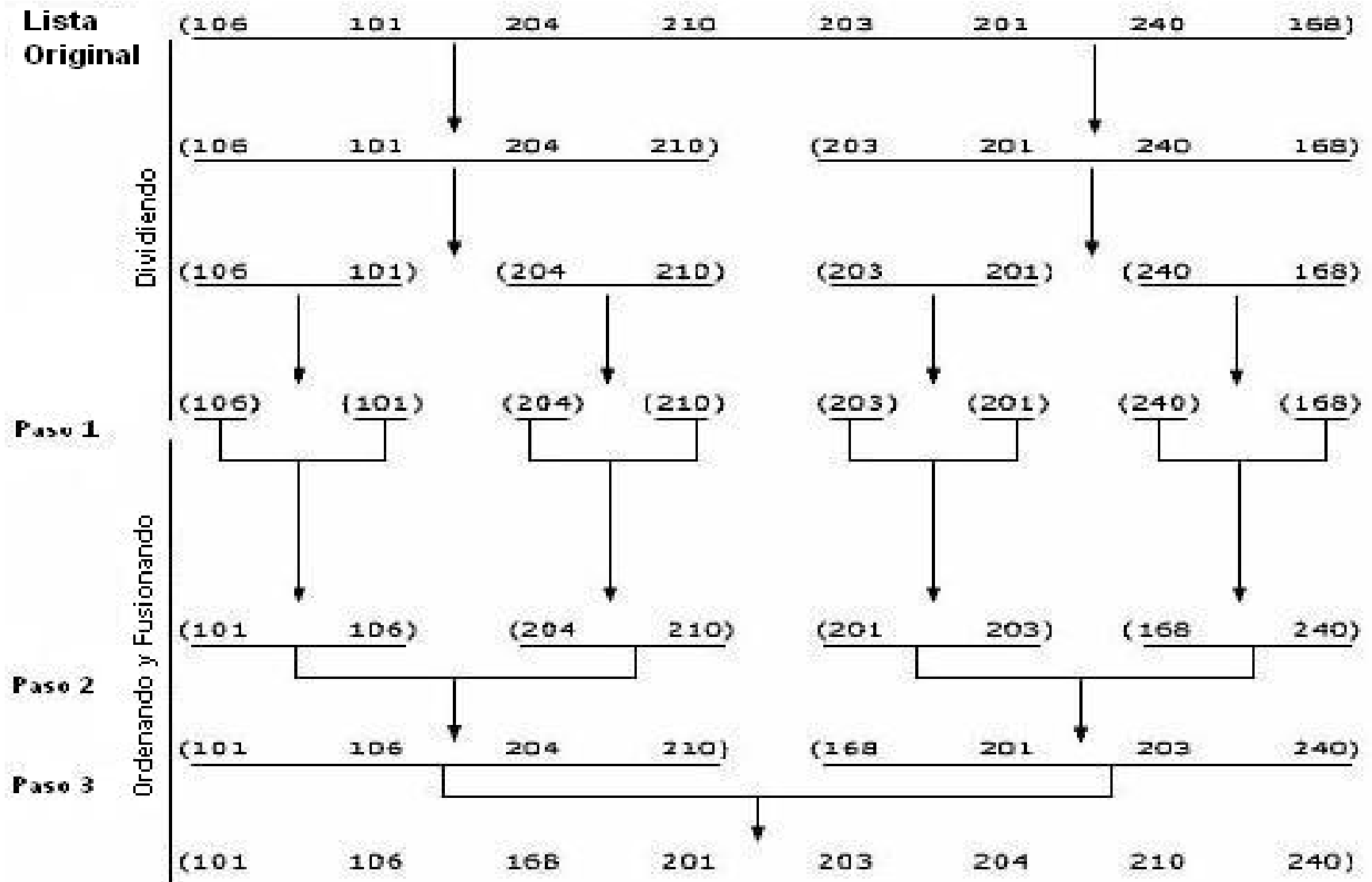
Introducción

- Un método de dividir un problema en porciones manejables se denomina **dividir y conquistar**.
- Para ordenar usando este enfoque hacer:
 - Ingresar el arreglo
 - Dividir el arreglo de entrada en dos arreglos, arreglo X y arreglo Y
 - Ordenar el arreglo X por separado
 - Ordenar el arreglo Y por separado
 - Fusionar los arreglos ordenados X e Y en un solo arreglo

Ordenamiento Merge Sort (por Fusión)

- La lista original se divide en dos arreglos de casi del mismo tamaño.
- Los dos arreglos se ordenan por separado.
- Los dos subarreglos ordenados después se fusionan en un sólo arreglo ordenado

Ordenamiento Merge Sort (por Fusión) - Ejemplo



Después del paso 1 (División):

- Los arreglos de un sólo elemento son fusionados en arreglos con dos elementos cada uno.
- Dentro de cada subarreglo los elementos están ordenados.
- El arreglo sólo tiene dos elementos a ordenar, sólo tomaría una pasada.

Para los pasos p (Ordenar y Fusionar):

- Después del paso p por el arreglo, los arreglos de $p/2$ -elementos serán fusionados en arreglos conteniendo 2^p elementos
- Dentro de cada subarreglo los elementos están ordenados.
- El arreglo tiene sólo 2^p elementos a ordenar y toma p pasadas.

Ordenamiento Merge Sort – Propiedades Especiales

Si

$$2^p = x$$

$$p = \log_2 x$$

De este modo

$$n = \log_2 n$$

Un arreglo con n elementos requiere $\log_2 n$ pasos

Ordenamiento Merge Sort – Funciones

- **mergeSort**

Toma como entrada el arreglo a ser ordenado y el número de elementos en el arreglo y controla el ordenamiento por fusión invocando a las funciones apropiadas.

- **calculateUpperBound**

Calcula el límite superior del subarreglo que es creado.

- **sortAndAssign**

La función ordena los subarreglos individuales y los asigna al arreglo auxiliar dummy.

Ordenamiento Merge Sort – Funciones

- **assignSub y assignRemaining**

Estas dos funciones realizan esencialmente la misma tarea, esto es, asignar los subarreglos ordenados al arreglo auxiliar dummy

- **assignSub**

Hace la asignación de los subarreglos apropiadamente divididos mediante el procedimiento de ordenamiento por fusión

- **assignRemaining**

Trabaja con la parte restante del subarreglo

Función calculateUpperBound

```
#define MAX_SIZE 100
int calculateUpperBound(int lowerBound2, \
    int size, int listSize) {

    if(lowerBound2 + size - 1 < listSize)
        return lowerBound2 + size - 1;
    else
        return listSize - 1;
}
```

Función sortAndAssign

```
void sortAndAssign(int lb1, int lb2, \
    int ub1, int ub2, int *i, int *j, \
    int *k, int x[], int dummy[]) {

    for(*i = lb1, *j = lb2; *i <= ub1 \
        && *j <= ub2; *k = *k+1){
        if(x[*i] <= x[*j]) {
            dummy[*k] = x[*i];
            *i = *i + 1;
        }else {
            dummy[*k] = x[*j];
            *j = *j + 1;
        }
    }
}
```

Función assignSub

```
void assignSub(int i, int ub, int *k, \
int x[], int dummy[]) {
    int numLoop = 0;

    while (i <= ub) {
        dummy[*k] = x[i++];
        *k = *k + 1;
    }
}
```

Función assignRemaining

```
void assignRemaining(int lb1, int k, \  
    int n, int x[], int dummy[]) {  
    int i;  
  
    for(i = lb1; k < n; i++) {  
        dummy[k++] = x[i];  
    }  
}
```


Función mergeSort

```
void mergeSort(int *myArray, int n) {  
    int dummy[MAX_SIZE], i=0, j=0;  
    int targetIndex = 0, loopCount = 0;  
    int lowerBound1, lowerBound2;  
    int upperBound1, upperBound2, size;  
    size = 1;  
    while(size < n) {  
        lowerBound1 = 0;  
        targetIndex = 0;
```

Función mergeSort...1

```
while (lowerBound1 + size < n) {  
    lowerBound2 = lowerBound1 + size;  
    upperBound1 = lowerBound2 - 1;  
    upperBound2 = calculateUpperBound \  
                    (lowerBound2, size, n);  
    sortAndAssign (lowerBound1, lowerBound2, \  
                    upperBound1, upperBound2, &i, &j, \  
                    &targetIndex, myArray, dummy);  
    assignSub (i, upperBound1, \  
               &targetIndex, myArray, dummy);  
}
```

Función mergeSort...2

```
    assignSub(j, upperBound2, \
        &targetIndex, myArray, dummy);
    lowerBound1 = upperBound2 + 1;
}
assignRemaining(lowerBound1, \
    targetIndex, n, myArray, dummy);
for(i = 0; i < n; i++)
    myArray[i] = dummy[i];
    size *= 2;
}
}
```

Ordenamiento Merge Sort - Ejecución

Lista Original:

106 101 204 210 203 201 240 168

Aplicando Merge Sort:

Pasada 1: *Ordenar y fusionar:* (101 106)
 Ordenar y fusionar: (204 210)
 Ordenar y fusionar: (201 203)
 Ordenar y fusionar: (168 240)

Pasada 2:
 Ordenar y fusionar: (101 106 204 210)
 Ordenar y fusionar: (168 201 203 240)

Pasada 3:
 Ordenar y fusionar: (101 106 168 201
 203 204 210 240)

Lista Ordenada:

101 106 168 201 203 204 210 240

Ordenamiento Quicksort (Rápido)

Características del Ordenamiento Quicksort:

- Se basa en principios similares al algoritmo de ordenamiento por fusión.
- La división se hace con mayor sofisticación e implicación.
- Un paso importante es el seleccionar un elemento clave, llamado un **pivote**.
- Los elementos se colocan de tal forma que los elementos con valores menores que el elemento pivote están a la izquierda del arreglo, mientras que los elementos mayores que el elemento pivote están a la derecha del arreglo.
- El arreglo se divide así en dos usando el elemento pivote.
- Estos dos subarreglos son ordenados por separado y fusionados, de manera que el arreglo combinado resultante esté ordenado.

Ordenamiento Quicksort (Rápido) - Ejemplo

Lista Original:

106 101 204 210 201 203 240 168

Aplicando QuickSort:

1era Pasada: (106 101 204 [210] 201 203 240 168)

2da Pasada: (168 101 [204] 106 201 203) 210 240

3era Pasada: (203 101 [168] 106 201) 204 210 240

4ta Pasada: ([106] 101) 168 203 201 204 210 240

5ta Pasada: 101 106 168 ([203] 201) 204 210 240

6ta Pasada: 101 106 168 (201 [203]) 204 210 240

Lista Ordenada:

101 106 168 201 203 204 210 240

Notación

[] Elemento pivote

() Lista a comparar

Escogee el Elemento Pivote

- La elección del elemento pivote puede ser en forma aleatoria entre los elementos del arreglo.
- Seleccionar el elemento más a la izquierda, el elemento más a la derecha y el elemento en el punto medio del arreglo.
- Escoger el elemento medio para que actúe como el pivote.
- Ejemplo:

106 101 204 210 201 203 240 168

Elemento más a la izquierda = **106**

Elemento más a la derecha = **168**

Elemento en el punto medio = **210**

Elemento medio 106, 168, 210 = **168**

Función quickSort

```
// Programa Quicksort
/* Esta función sólo lanza el proceso de
   quicksort, invocando la función Do_Quick_Sort
   */

void quickSort (int *myArray, int n) {
    doQuickSort(myArray, 0, n-1);
}
```


Función doQuickSort

// Algoritmo quicksort recursivo

```
void doQuickSort(int *myArray, int left_pos, \
                int right_pos, int len_array) {
    int pivotElementIndex, k, i;
    if (left_pos < right_pos) {
        pivotElementIndex = arrangeArray(myArray, \
                                         left_pos, right_pos, len_array);
        printf("left:%d,right:%d",left_pos,right_pos);
        printf("\nIndice del elemento pivote: \
                %d\n",pivotElementIndex);
        printf("-----\n");
        printf("\n");
        doQuickSort(myArray,left_pos, pivotElementIndex-1,
                    len_array);
        doQuickSort(myArray, pivotElementIndex+1,
                    right_pos, len_array);
    }
}
```

Función `arrangeArray`

```
int arrangeArray(int *myArray, int left_pos, \
int right_pos, int len_array) {
    int i, k, pivotElement, temp;

    // Obtiene el elemento pivote

    pivotElement = findPivotElement(myArray, \
        left_pos, right_pos, len_array);

    /* Guarda la ubicación de la posición más a la
    izquierda en una variable temporal */

    k = left_pos;
```

Función arrangeArray...1

```
/* Usando el elemento pivote y su ubicación
encuentra la ubicación correcta para el elemento
pivote en el arreglo */
i = left_pos;
while(i <= right_pos) {
    if (pivotElement > myArray[i]) {
        k++;
        if (k != i) {
            temp = myArray[k];
            myArray[k] = myArray[i];
            myArray[i] = temp;
            printf("temp: %d,k: %d,i: %d\n",temp,k,i);
        }
        printArray(myArray, len_array, k, left_pos, right_pos);
        printf("\n");
    }
    i++;
}
```

Función arrangeArray...2

```
temp = myArray[left_pos];
myArray[left_pos] = myArray[k];
myArray[k] = temp;
printf("\n temp: %d, k: %d, i: %d\n", temp, k, i);
printArray(myArray, len_array, k, left_pos, right_pos);
printf("\n----- El Arreglo Ordenado ----- \n");
printArray(myArray, len_array, k, left_pos, right_pos);
return k;
}
```

Función findPivoteElement

```
int findPivoteElement(int *myArray, int left_pos, \
int right_pos, int len_array) {
    int mid_point, temp, k;
    mid_point = (left_pos + right_pos)/2;
    printf("\n-- El Arreglo con Nuevo Pivote --\n");
    printArray(myArray, len_array, mid_point,
left_pos, right_pos);
    printf("-----\n");

/* Cambia el elemento más a la izquierda con el
elemento del Punto Medio. Retorna el elemento
más a la izquierda como el elemento pivote a la
función que lo llama */
    temp = myArray[left_pos];
    myArray[left_pos] = myArray[mid_point];
    myArray[mid_point] = temp;
    return (myArray[left_pos]);
}
```

Resumen

- Explicar la técnica de ordenamiento merge sort (por fusión)
- Describir cómo escribir un algoritmo para desarrollar la técnica de ordenamiento merge sort (por fusión)
- Explicar acerca de la técnica de ordenamiento quicksort (rápido)
- Describir cómo escribir un algoritmo para desarrollar la técnica de ordenamiento quicksort (rápido)

Unidad 6:

Laboratorio Técnicas Avanzadas de Ordenamiento

Unidad 7:

Técnicas de Búsqueda

Objetivos del Aprendizaje

- Explicar las diversas técnicas de búsqueda
- Diferenciar entre búsqueda interna y externa
- Describir como desarrollar un algoritmo para la técnica de búsqueda lineal
- Describir como desarrollar un algoritmo para la técnica de búsqueda binaria
- Explicar el uso de hashing para insertar y localizar elementos en una tabla hash
- Estudiar los métodos de resolución de colisión hash

Ejemplos de Búsqueda

Algunos ejemplos de búsqueda

- Localizar los sinónimos de una palabra en un diccionario.
- Localizar la dirección de un alumno desde la base de datos universitaria.
- Localizar el número de cuenta del cliente de un banco.
- Localizar todos los agentes de comercio Ferrari en Europa.
- Localizar los empleados que han estado trabajando en una organización por más de 20 años y están ganando un salario anual de \$1M.

Introducción a Búsqueda

- La búsqueda se lleva acabo en base a un elemento particular
- El elemento que es la base de la búsqueda, se denomina **elemento de búsqueda** o **elemento clave**

Introducción a Búsqueda...1

La búsqueda se divide en dos áreas:

- **Búsqueda Interna**

Cuando los registros se buscan en el área de memoria primaria.

- **Búsqueda Externa**

Cuando el número de registros es demasiado grande y no se puedan mantener juntos en la memoria primaria, la mayor parte de los registros son guardados en un dispositivo de almacenamiento secundario y se realiza la búsqueda usando uno de los métodos de búsqueda externa

Búsqueda Lineal o Secuencial

Paso 1: Declarar el arreglo que almacenará los elementos.

Paso 2: Leer los elementos hacia el arreglo.

Paso 3: Leer el elemento a buscar.

Paso 4: Fijar la variable contadora del ciclo k a 1.

Paso 5: Si el elemento k en el arreglo es igual al buscado, entonces el elemento es encontrado. Tomar la acción adecuada. Ir al paso 9.

Búsqueda Lineal o Secuencial

Paso 6: Si el elemento k no es igual al elemento buscado, entonces incrementar k en 1

Paso 7: Si k es mayor que el número de elementos en el arreglo, entonces el elemento no se encuentra. Tomar la acción adecuada. Ir al paso 9

Paso 8: Ir al paso 5

Paso 9: Fin del algoritmo

Programa para Búsqueda Lineal o Secuencial

```
typedef int Element_type;
int linearSearch(Element_type *elements, \
    int n, Element_type element) {
    int k, found = 0;

    for (k = 0; k < n && !found; k++)
        if (elements[k] == element)
            found = 1;
    return found;
}
```

Programa para Búsqueda Lineal o Secuencial

```
typedef int Element_type;
int linearSearch(Element_type *elements,\
                int n, Element_type element) {
    int k;

    for (k = 0; k < n; k++)
        if (elements[k] == element)
            return 1;
    return 0;
}
```


Programa para Búsqueda Lineal o Secuencial

```
typedef int Element_type;
int linearSearch(Element_type *elements, \
                int n, Element_type element) {
    int k;

    for (k = 0; k < n && elements[k] \
        != element; k++);
    if (k < n)
        return 1;
    else
        return 0;
}
```

Búsqueda Binaria

Paso 1: Ingresar el arreglo y ordenarlo en forma ascendente

Paso 2: Ingresar el elemento a buscar.

Paso 3: El índice 0 del arreglo se llamara **top**, y el índice **n-1** se llamará **bot**

Paso 4: Encontrar el elemento central, es decir, el punto medio en el arreglo. Se puede hacer usando la fórmula:

$$\text{mid} = (\text{top} + \text{bot}) / 2$$

Ejemplo:

Considere que el número de elementos **n** es 9.

El valor de **bot** es 8

El valor de **mid** es: $(0 + 8) / 2 = 4$

Si el número de elementos fuese 10 y **bot** 9, el valor de **mid** será fijado en 4, como resultado de la división entera

Búsqueda Binaria

Paso 5: Verificar si el elemento a buscar es igual al elemento en `mid`. Si es así, entonces tomar la acción apropiada. Ir al Paso 8.

Paso 6: Si el elemento a buscar **no es igual** al elemento en `mid`, verificar:

- Si este **es menor** que el elemento en `mid` entonces se hace la búsqueda en la mitad superior del arreglo.

Se fija un nuevo valor a **`bot = mid - 1`**

- Si el elemento a buscar **es mayor** que el elemento en `mid`, entonces se hace la búsqueda en la mitad inferior del arreglo.

Se fija un nuevo valor a **`top = mid + 1`**

- En cualquier caso, se ha reducido a la mitad el número de elementos a comparar

Búsqueda Binaria

Paso 7: Verificar si $\text{top} > \text{bot}$.

Si es cierto, entonces el elemento a buscar no se encontró. Tomar la acción apropiada.

Si es falso, entonces ir al paso 4.

Paso 8: Fin del algoritmo

Búsqueda Exitosa Usando Algoritmo de Búsqueda Binaria

top	→ 0	binary
	1	complexity
	2	dictionary
	3	explanation
	4	linear
	5	method
	6	node
mid	→ 7	paragraph
	8	probing
	9	rephrased
	10	sentence
	11	tree
	12	while
	13	word
bot	→ 14	wrong

(a)

top	→ 0	binary
	1	complexity
	2	dictionary
mid	→ 3	explanation
	4	linear
	5	method
bot	→ 6	node
	7	paragraph
	8	probing
	9	rephrased
	10	sentence
	11	tree
	12	while
	13	word
	14	wrong

(b)

	0	binary
	1	complexity
	2	dictionary
	3	explanation
top	→ 4	linear
mid	→ 5	method
bot	→ 6	node
	7	paragraph
	8	probing
	9	rephrased
	10	sentence
	11	tree
	12	while
	13	word
	14	wrong

(c)

Primer Ejemplo de una Búsqueda sin Éxito

top →	0	-100
	1	-4
	2	0
	3	30
	4	81
	5	100
	6	101
	7	200
mid →	8	467
	9	2001
	10	3213
	11	43555
	12	50000
	13	60000
	14	90000
	15	100001
bot →	16	100006

(a)


top →	0	-100
	1	-4
	2	0
mid →	3	30
	4	81
	5	100
	6	101
bot →	7	200
	8	467
	9	2001
	10	3213
	11	43555
	12	50000
	13	60000
	14	90000
	15	100001
	16	100006

(b)

top →	0	-100
mid →	1	-4
bot →	2	0
	3	30
	4	81
	5	100
	6	101
	7	200
	8	467
	9	2001
	10	3213
	11	43555
	12	50000
	13	60000
	14	90000
	15	100001
	16	100006


(c)

Primer Ejemplo de una Búsqueda sin Éxito...1



0	-100
1	-4
2	0
3	30
4	81
5	100
6	101
7	200
8	467
9	2001
10	3213
11	43555
12	50000
13	60000
14	90000
15	100001
16	100006

(d)



0	-100
1	-4
2	0
3	30
4	81
5	100
6	101
7	200
8	467
9	2001
10	3213
11	43555
12	50000
13	60000
14	90000
15	100001
16	100006

(e)

Segundo Ejemplo de una Búsqueda sin Éxito

top →	0	-100
	1	-4
	2	0
	3	30
	4	81
	5	100
	6	101
	7	200
mid →	8	467
	9	2001
	10	3213
	11	43555
	12	50000
	13	60000
	14	90000
	15	100001
bot →	16	100006

(a)

	0	-100
	1	-4
	2	0
	3	30
	4	81
	5	100
	6	101
	7	200
	8	467
top →	9	2001
	10	3213
	11	43555
mid →	12	50000
	13	60000
	14	90000
	15	100001
bot →	16	100006

(b)

	0	-100
	1	-4
	2	0
	3	30
	4	81
	5	100
	6	101
	7	200
	8	467
	9	2001
	10	3213
	11	43555
	12	50000
top →	13	60000
mid →	14	90000
	15	100001
bot →	16	100006

(c)

Segundo Ejemplo de una Búsqueda sin Éxito

0	-100
1	-4
2	0
3	30
4	81
5	100
6	101
7	200
8	467
9	2001
10	3213
11	43555
12	50000
13	60000
14	90000
top mid bot	15 15 16
	100001
	100006

(d)

0	-100
1	-4
2	0
3	30
4	81
5	100
6	101
7	200
8	467
9	2001
10	3213
11	43555
12	50000
13	60000
14	90000
top	15
mid bot	16 16
	100001
	100006

(e)

0	-100
1	-4
2	0
3	30
4	81
5	100
6	101
7	200
8	467
9	2001
10	3213
11	43555
12	50000
13	60000
14	90000
bot	15
mid top	16 16
	100001
	100006

(f)

Programa para la Búsqueda Binaria

```
typedef int Element_type;
int binarySearch(Element_type *elements, \
                int n, Element_type element) {
    int top, bot, mid, found;
    top = 0;
    bot = n-1;
    found = 0;
    while (top <= bot && !found) {
        mid = (top + bot)/2;
        if (elements[mid] == element)
            found = 1;
        else
            if (element > elements[mid])
                top = mid +1;
            else
                bot = mid -1;
    }
    return found;
}
```

Comparación entre Búsqueda Lineal y Binaria

➤ **Búsqueda Lineal**

- No es posible conocer de antemano el número de comparaciones que se realizarán antes que se encuentre el elemento de búsqueda
- El arreglo no necesita ser ordenado antes de la búsqueda

➤ **Búsqueda Binaria**

- Cada comparación reduce el tamaño del arreglo a la mitad, por lo que se tiene una idea general acerca del número máximo posible de comparaciones
- El arreglo necesita ser ordenado antes de la búsqueda

Tablas Hash

- Una tabla hash es una estructura de datos que proporciona un método rápido y más eficiente para buscar los elementos de un arreglo
- La tabla para contener los elementos se define normalmente muy grande
- Los elementos en una tabla son almacenados en base a la correspondencia de un elemento a un entero único
- El único entero cae en el rango de índices de la tabla hash
- Se llega al valor único usando la técnica llamada hashing

Hashing

- Una **función hash** se utiliza para ejecutar hashing
- El valor retornado por una función hash es llamada **valor hash**
- La función hash toma el elemento como una entrada y retorna un valor que corresponde al elemento en un lugar único en la tabla hash
- Las tablas hash usualmente se definen para ser muy grandes, de manera tal que un rango amplio de índices esté representado
- Para almacenar 1000 enteros en una tabla hash se puede definir la tabla hash de un tamaño de 10000 enteros

Hashing - Ejemplo

- Función hash :
Número de caracteres en la palabra * valor ASCII del primer carácter
- Lista de palabras con sus valores hash:

complexity	→	990
tree	→	464
node	→	440
while	→	595
method	→	654
sentence	→	920
rephrased	→	1026
wrong	→	595
dictionary	→	1000
word	→	476

Tabla Hash de palabras

1	
...	
439	
440	node
441	
...	
463	
464	tree
465	
...	
475	
476	word
477	
...	
587	
588	binary
589	
...	
594	
595	while
596	
...	

Tabla Hash – Aspectos Importantes

➤ Orden de Almacenamiento

- No es en ninguna manera similar al orden de entrada de las palabras
- La primera palabra de la tabla hash es almacenada en la posición 440, la cual es el tercer elemento de la lista de entrada

➤ El Mismo Valor Hash

- Cuando dos o más palabras corresponde a la misma posición después de aplicar la función hash, se conoce como una **colisión**.
- Dos palabras tienen el valor hash de 595.

Inserción en una Tabla Hash

- La función hash se aplica al elemento que será insertado
- El valor hash se hace corresponder (map) a uno de los valores índices del arreglo definido
- Si esa posición en la tabla está vacía, entonces el elemento es insertado en esa posición
- Si la posición no está vacía, implica que ha ocurrido una colisión
- Se debe resolver la colisión antes de ser insertado el elemento en la tabla hash
- Se puede usar una función rehash para resolver la colisión

Eliminación en una Tabla Hash

- Utilizando la función hash y el elemento que se necesita recuperar desde la tabla hash, se calcula el valor hash
- Este es uno de los valores del índice del arreglo definido para representar la tabla hash
- Se verifica la posición del arreglo indicado por el valor hash del elemento
- Si el elemento es encontrado en la posición, entonces el proceso de recuperación tiene éxito
- Si el elemento no es encontrado en la posición, se aplica la función rehash.

Diferentes Funciones Hash

➤ **Criterio para Escoger una Función Hash**

- Debe ser simple y fácil de calcular
- Las función hash, de preferencia, debe dar como resultado una distribución uniforme de los índices

➤ **Tipos de Funciones Hash**

- El método de división modular
- El método de multiplicación
- El método mid-square
- El método de plegado (folding)

Método de División Modular

$$\text{Hash}(\text{key}) = \text{key} \bmod m$$

- Asuma que m es 25
- Considere los valores clave 2546, 128, 250, y 32767
- El resultado de:

$$2546 \bmod 25 = 21$$

$$128 \bmod 25 = 3$$

$$250 \bmod 25 = 0$$

$$32767 \bmod 25 = 17$$

Método de Multiplicación

- Seleccionar un número real k como constante que esté en el intervalo 0 a 1, tal que no esté demasiado cercano a 0 ó a 1.
- Calcular el valor fraccional de $key * k$.
- Para hacer corresponder la clave en el intervalo 0 a m , seleccionar m y luego se obtiene la parte entera de la expresión:

$$m * \text{fractional_part.}$$

- Esto dará un valor entre 0 y m .
- Función Hash:

$$\text{Integer_part}(\text{Fraction_part}(k * \text{key})) * m)$$

Método de Multiplicación - Ejemplo

- Key 2546

$$k * \text{key} = 0.4 * 2546 = 1018.4$$

$$\text{Fractional_part} = 0.4$$

$$\text{Fractional_part} * m = 25 * 0.4 = 10.0$$

$$\text{Integer_part} = 10 \text{ el valor hash}$$

- Key 128

$$k * \text{key} = 0.4 * 128 = 51.2$$

$$\text{Fractional_part} = 0.2$$

$$\text{Fractional_part} * m = 25 * 0.2 = 5.0$$

$$\text{Integer_part} = 5 \text{ el valor hash}$$

Método de Mid-Square

Se calcula key^2

Se elimina un cierto número de dígitos desde el final del número hasta el resultado en el valor hash.

- **Función Hash:**

`delete_digits_from_ends(key2)`

- **Ejemplo:**

- **Key 2546**

$$\text{Key}^2 = 2546^2 = 6482116$$

Dígitos eliminados = 648 y 16

Valor Hash = 21

- **Key 128**

$$\text{Key}^2 = 128^2 = 16384$$

Dígitos eliminados = 1 y 84

Valor Hash = 63

Método de Plegado (Folding)

- Fraccionar la clave en múltiples partes, tal que cada parte tenga el mismo número de dígitos, excepto para alguna parte.
- Las partes fraccionadas individuales son sumadas y el resultado es el valor hash.
- Función Hash: $\text{key}_1 + \text{key}_2 + \dots + \text{key}_n$
- Ejemplo:
 - Key 2546

La clave se divide en 25 y 46
Partes sumadas = $25 + 46 = 71$
Valor hash = 71
 - Key 128

La clave se divide en 12 y 8
Partes sumadas = $12 + 8 = 20$
Valor hash = 20

Método de Resolución de Colisiones – Análisis Lineal

- El Análisis Lineal (Linear probing) ve más allá de la posición del valor hash para una ubicación libre para una clave

Tree	2
Node	7
While	11
Method	2
Sentence	6
Dictionary	1
Word	11
Binary	4
Linear	14
Probe	9

Análisis Lineal (Linear Probing) - Ejemplo

1	Dictionary
2	Tree
3	Method
4	Binary
5	-
6	Sentence
7	Node
8	-
9	Probe
10	-
11	While
12	Word
13	-
14	Linear
15	-

Rehashing

- El agrupamiento no es deseable, dado que esto indica que las claves no están distribuidas uniformemente.
- Un método para evitar el agrupamiento es **rehashing**.
- En el método de **rehashing** cuando ocurre una colisión se calcula una segunda función hash.
- La posición de la clave será dada por el valor calculado de la segunda función hash, por esto es rehashing.

- **Función Hash**

`key value % SIZEOFARRAY`

- **Función Rehash**

`(key value + 1) % SIZEOFARRAY`

Análisis Cuadrático (Quadratic Probing)

En el Análisis Cuadrático (Quadratic Probing) se prueban las posiciones $x+1$, $x+4$, $x+9$, $x+16$ y así sucesivamente.

Tree	2
Node	7
While	11
Method	2
Sentence	7
Dictionary	11
Word	11
Binary	4
Linear	14
Probe	9

Análisis Cuadrático (Quadratic Probing) - Ejemplo

1	-	
2	Tree	
3	Method	x+1
4	Binary	
5	-	
6	-	
7	Node	
8	Sentence	x+1
9	Probe	
10	-	
11	While	
12	Dictionary	x+1
13	-	
14	Linear	
15	Word	x+4

Tabla Hash – Definición Básica

```
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE 10000

typedef int Element_type;
typedef struct node {
    Element_type element;
    struct node *next;
} Node_type;
typedef Node_type *HashTable_type[MAXSIZE];
```

Tabla Hash – Inicialización

`/* Función para inicializar la tabla hash */`

```
void initHashTable (HashTable_type hashTable)
{
    int i;
    for (i = 0; i < MAXSIZE; i++)
        hashTable[i] = NULL;
}
```

Tabla Hash – Funciones Hash y Rehash

```
/* La función hash */
```

```
int hash(int key) {  
    return (key % MAXSIZE);  
}
```

```
/* La función rehash */
```

```
int rehash(int key) {  
    return ((key + 1) % MAXSIZE);  
}
```


Tabla Hash – Inserción

```
/* Función para insertar en la tabla hash */
```

```
void insert(HashTable_type hashTable,  
            Node_type *node) {  
    int hashValue, firstHashValue;  
    hashValue = hash(node->element);  
    firstHashValue = hashValue;  
    while (hashTable[hashValue] != NULL) {  
        hashValue = rehash(hashValue);  
    }
```

Tabla Hash – Inserción

```
/* Chequear si la tabla hash está llena */  
if (hashValue == firstHashValue) {  
    printf("La tabla está llena: No puede \  
        insertar\n");  
    return; } }  
  
printf("Valor Hash para el elemento %d \  
    es %d\n", node->element, hashValue);  
node->next = hashTable[hashValue];  
hashTable[hashValue] = node;  
}
```

Tabla Hash – Recuperación de un elemento

```
/* Función para recuperar elementos de la  
tabla hash */
```

```
Node_type* find(HashTable_type hashTable, \  
    Element_type element, int *index) {  
  
    int hashValue = hash(element);  
    int firstHashValue = hashValue;  
    if (hashTable[hashValue] == NULL)  
        return NULL;
```

Tabla Hash – Recuperación de un elemento

```
while (hashTable[hashValue]->element \
      != element) {
    hashValue = rehash(hashValue);

// Chequear si la tabla hash table está llena
    if (hashValue == firstHashValue)
        return NULL;
}
*index = hashValue;
return hashTable[hashValue];
}
```

Aplicaciones de Tablas Hash

➤ Compiladores y Ensambladores

- Los compiladores que compilan programas fuente en lenguajes de alto nivel, tales como C o Pascal, mantienen una estructura de datos llamada tabla de símbolo
- La tabla de símbolo contiene, entre otras cosas, el símbolo en sí mismo y algunos atributos del símbolo derivados del contexto de su utilización en el programa

➤ Aplicaciones que requieren buscar por el valor de la clave

- Existen numerosas aplicaciones que requieren ejecutar una búsqueda basada en un valor clave
- La búsqueda puede ser ejecutada en un diccionario, un directorio telefónico, un catalogo de productos o un índice de un sistema de archivos en un libro

Resumen

- Se explicaron las diversas técnicas de búsqueda disponibles para operaciones en arreglos y otras estructuras de datos
- Se describió cómo desarrollar un algoritmo para la técnica de búsqueda lineal
- Se describió cómo desarrollar un algoritmo para la técnica de búsqueda binaria
- Se explicó el uso de la técnica hashing para insertar y localizar elementos en una tabla hash
- Se presentaron los métodos de resolución de una colisión hash

Unidad 8:

Laboratorio Técnicas de Búsqueda

Unidad 9:

Laboratorio Tablas Hash