

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	1/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Manual de prácticas del Laboratorio de Estructuras de Datos y Algoritmos II

Elaborado por:	Revisado por:	Autorizado por:	Vigente desde:
Elba Karen Sáenz García	Laura Sandoval Montaño	Alejandro Velázquez Mena	20 de enero de 2017

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	2/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Índice de prácticas

No	Nombre	Página
1	Algoritmos de Ordenamiento. Parte I.	3
2	Algoritmos de Ordenamiento. Parte II.	14
3	Algoritmos de Ordenamiento. Parte III.	25
4	Algoritmos de búsqueda parte 1	35
5	Algoritmos de búsqueda parte 2	47
6	Algoritmos de grafos. Parte 1	61
7	Algoritmos de grafos. Parte 2	80
8	Árboles. Parte 1	90
9	Árboles. Parte 2	106
10	Archivos	125
11	Introducción a OpenMP	137
12	Algoritmos paralelos. Parte 1	156
13	Algoritmos paralelos. Parte 2	172

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	3/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 1

Algoritmos de ordenamiento parte 1

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	4/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 1

Estructura de datos y Algoritmos II

Algoritmos de Ordenamiento. Parte 1.

Objetivo: El estudiante identificará la estructura de los algoritmos de ordenamiento *Bubble Sort* y *Merge Sort*.

Actividades

- Implementar el algoritmo *Bubble Sort* en algún lenguaje de programación para ordenar una secuencia de datos.
- Implementar el algoritmo *Merge Sort* en algún lenguaje de programación para ordenar una secuencia de datos.

Antecedentes

- Análisis previo de los algoritmos en clase teórica.
- Manejo de arreglos o listas, estructuras de control y funciones en Python 3.

Introducción

Debido a que las estructuras de datos son utilizadas para almacenar información, para poder recuperar o buscar esa información de manera eficiente es deseable que ésta esté ordenada.

Ordenar un grupo de datos es arreglarlos en algún orden secuencial ya sea en forma ascendente o descendente.

Hay dos tipos de ordenamiento que se pueden realizar: el interno y el externo. El interno se lleva a cabo completamente en la memoria de acceso aleatorio de alta velocidad de la computadora es decir todos los elementos que se ordenan caben en la memoria principal de la computadora. Cuando no cabe toda la información en memoria principal es necesario ocupar memoria secundaria y se dice que se realiza un ordenamiento externo [2].

Los algoritmos de ordenamiento de información se pueden clasificar en cuatro grupos:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	5/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Algoritmos de inserción: Se considera un elemento a la vez y cada elemento es insertado en la posición apropiada con respecto a los demás elementos que ya han sido ordenados. Ejemplo de algoritmos que trabajan de esta manera es Inserción directa, *shell sort*, inserción binaria y *hashing*.

Algoritmos de Intercambio: En estos algoritmos se trabaja con parejas de elementos que se van comparando y se intercambian si no están en el orden adecuado. El proceso se realiza hasta que se han revisado todos los elementos del conjunto a ordenar. Los algoritmos de *Bubble Sort* y *Quicksort* son ejemplos de algoritmos que trabajan de la forma mencionada.

Algoritmos de selección: En estos algoritmos se selecciona el elemento mínimo o el máximo de todo el conjunto a ordenar y se coloca en la posición apropiada. Esta selección se realiza con todos los elementos restantes del conjunto.

Algoritmos de enumeración: Se compara cada elemento con todos los demás y se determina cuántos son menores que él. La información del conteo para cada elemento indicará su posición de ordenamiento.

Algoritmo BubbleSort

El método de ordenación por burbuja (*BubbleSort* en inglés) es uno de los más básicos, simples y fáciles de comprender, pero también es poco eficiente.

La técnica utilizada se denomina ordenación por burbuja u ordenación por hundimiento y es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor a otro entonces se intercambia de posición. Los valores más pequeños suben o burbujan hacia la cima o parte superior de la lista, mientras que los valores mayores se hunden en la parte inferior.

Descripción Algoritmo

Para una lista a de n elementos numerados de 0 a $n - 1$, el algoritmo requiere $n - 1$ pasadas. Por cada pasada se comparan elementos adyacentes (parejas sucesivas) y se intercambian sus valores cuando el primer elemento es mayor que el segundo. Al final de cada pasada el elemento mayor se dice que ha burbujado hasta la cima de la sub-lista actual.

Sea la lista $a_0, a_1, a_2, \dots, a_{n-1}$, entonces:

En la pasada 0 se comparan elementos adyacentes $(a_0, a_1), (a_1, a_2), (a_2, a_3), \dots, (a_{n-2}, a_{n-1})$

Se realizan $n - 1$ comparaciones por cada pareja (a_i, a_{i+1}) y se intercambian si $a_{i+1} < a_i$. Al final de la pasada el elemento mayor de la lista estará situado en la posición $n - 1$.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	6/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el segundo elemento de mayor valor el cual estará situado en la posición $n - 2$.

Se realizan las siguientes pasadas de forma similar y el proceso termina con la pasada $n - 1$ donde se tendrá al elemento más pequeño en la posición 0.

A continuación, se muestra un ejemplo del proceso con la lista {9,21,4,40,10,35}.

Primera Pasada

{9,21,4,40,10,35} --> {9,21,4,40,10,35} No se realiza intercambio
 {9,21,4,40,10,35} --> {9,4,21,40,10,35} Intercambio entre el 21 y el 4
 {9,4,21,40,10,35} --> {9,4,21,40,10,35} No se realiza intercambio
 {9,4,21,40,10,35} --> {9,4,21,10,40,35} Intercambio entre el 40 y el 10
 {9,4,21,10,40,35} --> {9,4,21,10,35,40} Intercambio entre el 40 y el 35

Segunda Pasada

{9,4,21,10,35,40} --> {4,9,21,10,35,40} Intercambio entre el 9 y el 4
 {4,9,21,10,35,40} --> {4,9,21,10,35,40} No se realiza intercambio
 {4,9,21,10,35,40} --> {4,9,10,21,35,40} Intercambio entre el 21 y el 10
 {4,9,10,21,35,40} --> {4,9,10,21,35,40} No se realiza intercambio
 {4,9,10,21,35,40} --> {4,9,10,21,35,40} No se realiza intercambio

Aunque la lista ya está ordenada se harían otras 3 pasadas.

Si la lista se representa con arreglos lineales un algoritmo en pseudocódigo es:

```

bubbleSort(a, n)
  Para i=1 hasta n -1
    Para j=0 hasta i < n -2
      Si (a[j]>a[j+1]) entonces
        tmp=a[j]
        a[j]=a[j+1]
        a[j+1]=tmp
      Fin Si
    Fin Para
  Fin Para
Fin bubbleSort

```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	7/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

En este pseudocódigo se considera una lista con n elementos, las dos estructuras de repetición se realizan $n - 1$ veces es decir se hacen $n - 1$ pasadas y $n - 1$ comparaciones en cada pasada. Por consiguiente, el número de comparaciones es $(n - 1) * (n - 1) = n^2 - 2n + 1$, es decir la complejidad es $O(n^2)$.

Una posible mejora consiste en añadir una bandera que indique si se ha producido algún intercambio durante el recorrido del arreglo y en caso de que no se haya producido ninguno, el arreglo se encuentra ordenado y se puede abandonar el método.

bubbleSort2 (a, n)

```

Inicio
bandera = 1;
pasada=0
Mientras pasada < n-1 y bandera es igual a 1
  bandera = 0
  Para j = 0 hasta j < n-pasada-1
    Si a[j] > a[j+1] entonces
      bandera = 1
      tmp= a[j];
      a[j] = a[j+1];
      a[j+1] = tmp;
  Fin Si
  Fin Para
  pasada=pasada+1
Fin Mientras
Fin

```

Con la mejora se tendrá que en el mejor de los casos la ordenación se hace en una sola pasada y su complejidad es $O(n)$. En el peor de los casos se requieren $(n - i - 1)$ comparaciones y $(n - i - 1)$ intercambios y la ordenación completa requiere de $\frac{n(n-1)}{2}$ comparaciones. Así la complejidad sigue siendo $O(n^2)$.

Merge Sort

Merge Sort es un algoritmo de ordenamiento basado en el paradigma divide y vencerás o también conocido como divide y conquista. Esta es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo, pero de menor tamaño. Consta de tres pasos: Dividir el problema, resolver cada sub-problema y combinar las soluciones obtenidas para la solución al problema original

Las tres fases en este algoritmo son:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	8/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Divide: Se divide una secuencia A de n elementos a ordenar en dos sub-secuencias de $n/2$ elementos. Si la secuencia se representa por un arreglo lineal, para dividirlo en 2 se encuentra un número q entre p y r que son los extremos del arreglo o sub-arreglo. Figura 1.1. El cálculo se hace $q = \lfloor \frac{p+r}{2} \rfloor$.

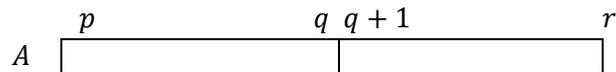


Figura 1.1.

Conquista: Ordena las dos sub-secuencias de forma recursiva utilizando el algoritmo **MergeSort()**. El caso base o término de la recursión ocurre cuando la secuencia a ser ordenada tiene solo un elemento y éste se supone ordenado.

Combina: Mezcla las dos sub-secuencias ordenadas para obtener la solución.

El algoritmo en pseudocódigo queda [1]:

```

MergeSort(A,p,r)
Inicio
  Si p<r entonces
    Si p<r entonces } // Divide
    q = ⌊(p+r)/2⌋ }
    MergeSort(A, p, q) } //Conquista
    MergeSort(A, q+1, r) }
    Merge(A,p,q,r) // Mezcla o combina
  Fin Si
Fin

```

La parte primordial de este algoritmo es mezclar las dos sub-secuencias de forma ordenada en la fase de **combina** y en el pseudocódigo se utiliza una función auxiliar $Merge(A, p, q, r)$ que realiza esta función. Para entender mejor el algoritmo y ver cómo se realiza la mezcla se considera el arreglo A en la Figura 1.2.

1	2	3	4	5	6	7	8	9	10
12	9	3	7	14	11	6	2	10	5

Figura 1.2.

En un inicio se hace la llamada a la función $MergeSort(A, 1, 10)$ dado que el arreglo A tiene 10 elementos y los índices de inicio y fin son 1 y 10 respectivamente. Si los índices de inicio y fin fueran 0 y 9, entonces la llamada se haría $Merge(A, 0, 9)$.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	9/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Se calcula el punto medio q que es 5 y se llama de forma recursiva a la función $MergeSort(A, 1, 5)$ y $MergeSort(A, 6, 10)$ sobre los subarreglos. Figura 1.3.



Figura 1.3.

Después de algunas llamadas recursivas, donde en cada una de ellas se calcula el punto medio q para obtener las siguientes sub-secuencias hasta llegar al caso base en el que están formadas por un elemento y donde se suponen ya ordenadas, se mezclan para ir obteniendo sub-secuencias ordenadas. Ejemplo Figura 1.4.

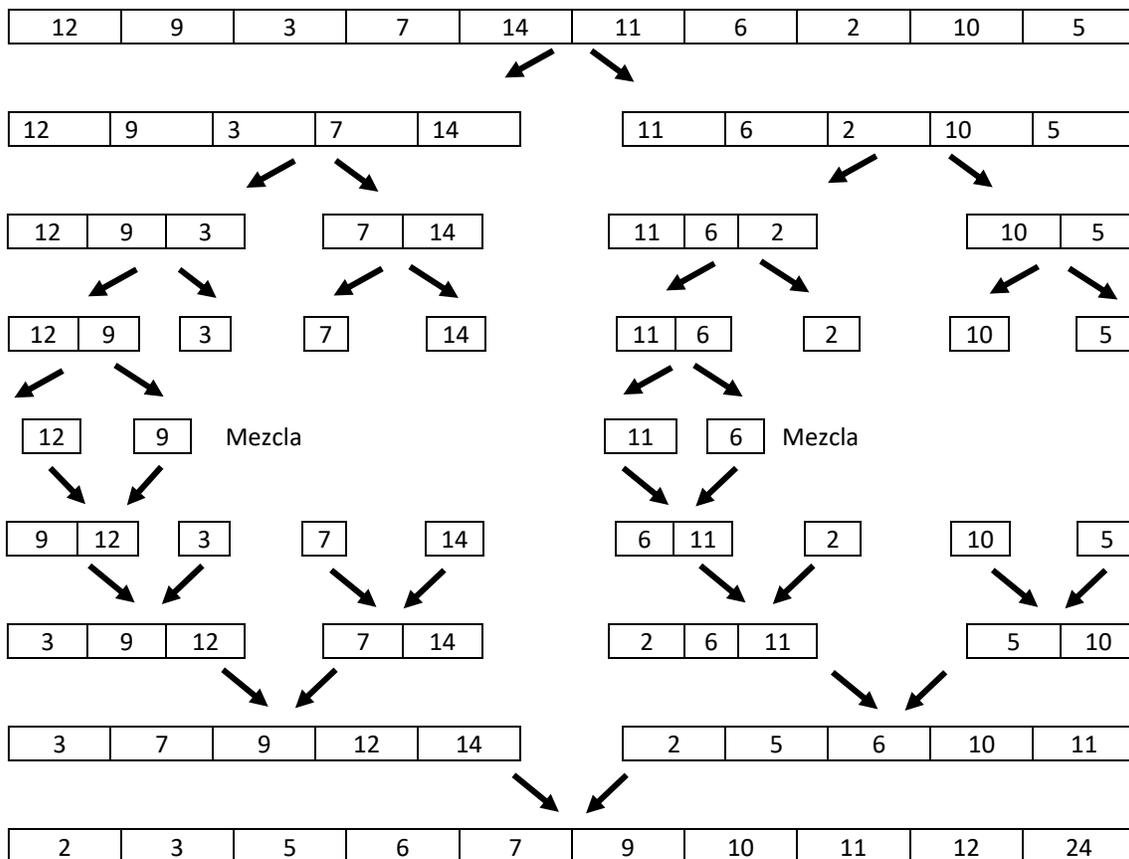


Figura 1.4.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	10/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Para comprender cómo se realiza la mezcla, se observa en la figura 1.4 que después de varias llamadas recursivas se obtienen dos sub arreglos ordenados (Figura 1.5)

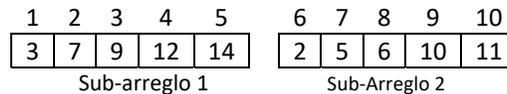


Figura 1.5.

Estos sub-arreglos se mezclan con la función $Merge(A, 1, 5, 10)$, tomando los valores en orden y quedando el arreglo ordenado. Así primero se escoge el 2 del sub-arreglo de la derecha luego el 3 de la izquierda, 5 y 6 de la derecha, 7 y 9 de la izquierda, 10 y 11 de la derecha y finalmente 12 y 14 de la izquierda. Figura 1.6.

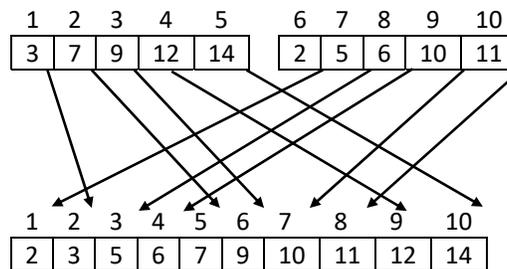


Figura 1.6

El siguiente es un pseudocódigo de la función $Merge()$ que realiza la mezcla, donde se considera que el índice de inicio de las secuencias es cero.

Merge(A,p,q,r)

Inicio

Formar sub-arreglo Izq[0,..., q-p] y sub-arreglo Der[0,...,r-q]

Copiar contenido de A[p...q] a Izq[0,..., q-p] y A[q+1...r] a Der[0,...,r-q -1]

i=0

j=0

Para k=p hasta r

Si (j >= r-q) ó (i < q-p+1 y Izq[i] < Der[j]) entonces

A[k]=Izq[k]

i=i+1

En otro caso

A[k]=Der[j]

j=j+1

Fin Si

Fin Para

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	11/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Desarrollo

Actividad 1

Abajo se muestra la implementación en Python de los pseudocódigos de las funciones *bubbleSort()* y *bubbleSort2()* mencionados en la guía y que permiten realizar el ordenamiento de una lista por **BubbleSort**. Se pide realizar un programa que ordene una lista de n elementos (la cual es proporcionada por el profesor) utilizando ambas funciones.

```
def bubbleSort(A):
    for i in range(1, len(A)+1):
        for j in range(len(A)-1):
            if A[j]>A[j+1]:
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp

def bubbleSort2(A):
    bandera= True
    pasada=0
    while pasada < len(A)-1 and bandera:
        bandera=False
        for j in range(len(A)-1-pasada):
            if(A[j] > A[j+1] ):
                bandera=True
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp
        pasada = pasada+1
```

Agregar al código la impresión para conocer el número de pasadas que realiza cada función e indicarlo. _____

¿Qué se tiene que hacer para ordenar la lista en orden inverso? Describirlo y modificar el código.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	12/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Actividad 2

A continuación, se proporciona la implementación en Python de los pseudocódigos de las funciones mencionadas en la guía para el algoritmo **MergeSort**. Se requiere utilizarlas para elaborar un programa que ordene una lista proporcionada por el profesor.

Agregar en el lugar correspondiente la impresión, para visualizar las sub-secuencias obtenidas en la recursión.

```

#MergeSort
|
def CrearSubArreglo(A, indIzq, indDer):
    return A[indIzq:indDer+1]

def Merge(A,p,q,r):
    Izq = CrearSubArreglo(A,p,q)
    Der = CrearSubArreglo(A,q+1,r)
    i = 0
    j = 0
    for k in range(p,r+1):
        if (j >= len(Der)) or (i < len(Izq) and Izq[i] < Der[j]):
            A[k] = Izq[i]
            i = i + 1
        else:
            A[k] = Der[j]
            j = j + 1

def MergeSort(A,p,r):
    if r - p > 0:
        q = int((p+r)/2)
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
        Merge(A,p,q,r)

```

¿Qué cambio(s) se hace(n) en el algoritmo para ordenar la lista en orden inverso? . Describirlo y realizar cambios en el código. _____

Actividad 3

Ejercicios propuestos por el profesor.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	13/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Referencias

[1] CORMEN, Thomas, LEISERSON, Charles, et al.

Introduction to Algorithms

3rd edition MA, USA

The MIT Press, 2009

[2] KNUTH, Donald

The Art of Computer Programming

New Jersey

Addison-Wesley Professional, 2011

Volumen 3

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	14/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 2

Algoritmos de ordenamiento parte 2

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	15/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 2

Estructura de datos y Algoritmos II

Algoritmos de Ordenamiento. Parte 2.

Objetivo: El estudiante conocerá e identificará la estructura de los algoritmos de ordenamiento *Quick Sort* y *Heap Sort*.

Actividades

- Implementar el algoritmo *Quick Sort* en algún lenguaje de programación para ordenar una secuencia de datos.
- Implementar el algoritmo *Heap Sort* en algún lenguaje de programación para ordenar una secuencia de datos.

Antecedentes

- Análisis previo de los algoritmos en clase teórica.
- Manejo de arreglos o listas, estructuras de control y funciones en Python 3.

Introducción

Quick Sort

Este algoritmo de ordenamiento al igual que *Merge Sort* sigue el paradigma divide y conquista por lo que en este documento se explican los tres procesos involucrados para ordenar una secuencia de números.

Para su descripción, la secuencia a ordenar está representada por un arreglo lineal o unidimensional. Los tres procesos son:

Divide: Se divide un arreglo A en 2 sub-arreglos utilizando un elemento pivote x de manera que de un lado queden todos los elementos menores o iguales a él y del otro los mayores. Figura 2.1.

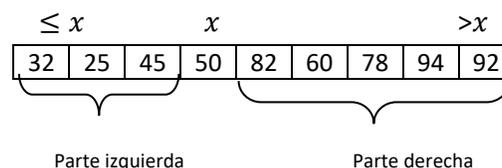


Figura 2.1

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	16/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Si el arreglo se representa como $A[p, \dots, r]$, donde el primer elemento está en la posición p y el último en r , al dividir la secuencia se tiene [1]:

- El elemento pivote es $x = A[q]$
- El subarreglo de izquierda es $A[p, \dots, q - 1]$
- El subarreglo de la derecha es $A[q + 1, \dots, r]$ y

Los valores de x cumplen que $A[p, \dots, q - 1] \leq x < A[q + 1, \dots, r]$, como se observa en la Figura 2.2



Figura 2.2

Conquista o resolver subproblemas: Ordenar los sub-arreglos de la izquierda y derecha con llamadas recursivas a la función *QuickSort()*.

Combina: En el caso de un arreglo, como los sub-arreglos son parte del mismo arreglo A y cada uno ya está ordenado no es necesario combinarlas, el arreglo $A[p, \dots, r]$ ya está ordenado.

Un algoritmo general para ordenar una lista se puede representar como:

QuickSort()

Inicio

Si lista tiene más de un elemento

Particionar la lista en dos sublistas (Sublista Izquierda y Sublista Derecha)

Aplicar el algoritmo QuickSort() a Sublist Izquierda

Aplicar Algoritmo QuickSort() a Sublista Derecha

Combinar las 2 listas ordenadas

Fin Si

FIN

Y de forma más específica considerando que A representa el arreglo o subarreglo a ordenar, p es el índice del primer elemento y r la del último, se tiene el siguiente pseudocódigo[1].

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	17/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

QuickSort(A,p,r)

Inicio

Si $p < r$ entonces // Si la lista tiene más de un elemento

$q = \text{Particionar}(A,p,r)$

QuickSort(A,p,q-1)

QuickSort(A,q+1,r)

Fin Si

Fin

Donde la función *Particionar()* es la clave del algoritmo, y lo que hace es reacomodar cada sub-arreglo para que de acuerdo a un elemento pivote seleccionado, los elementos queden acomodados de tal forma que a la izquierda queden todos los menores o iguales y del otro todos los mayores a él.

Hay variantes de esta función que dependen del elemento pivote a seleccionar (el último elemento, el primero, etc..), en ésta guía se explicará solo una alternativa.

En la función *Particionar()* en pseudocódigo [1] mostrada abajo, primero se define el elemento pivote x como el último elemento ($x = A[r]$) y después se va comparando cada elemento que se encuentran a la izquierda de él desde la posición $j = p$ hasta $j = r - 1$, de manera que si el elemento $A[j] \leq x$ entonces dicho elemento se intercambia para que quede del lado izquierdo y en otro caso vaya quedando del lado derecho. Al revisar todos los elementos el pivote se cambia para que se marque la división de los elementos menores y mayores a él. La función retorna la posición final del elemento pivote.

Particionar(A,p,r)

Inicio

$x = A[r]$

$i = p - 1$

para $j = p$ hasta $r - 1$

Si $A[j] \leq x$

$i = i + 1$

intercambiar $A[i]$ con $A[j]$

Fin Si

Fin para

intercambiar $A[i+1]$ con $A[r]$

retornar $i+1$

Fin

Un ejemplo de cómo se lleva a cabo la división de un sub-arreglo de 8 elementos se muestra en la figura 2.3 y ahí del inciso a a h se puede observar lo que sucede a los elementos en cada iteración y en el inciso i cómo queda el pivote marcando la división entre los menores o iguales y los mayores a él. En este ejemplo el pivote es 4.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	18/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

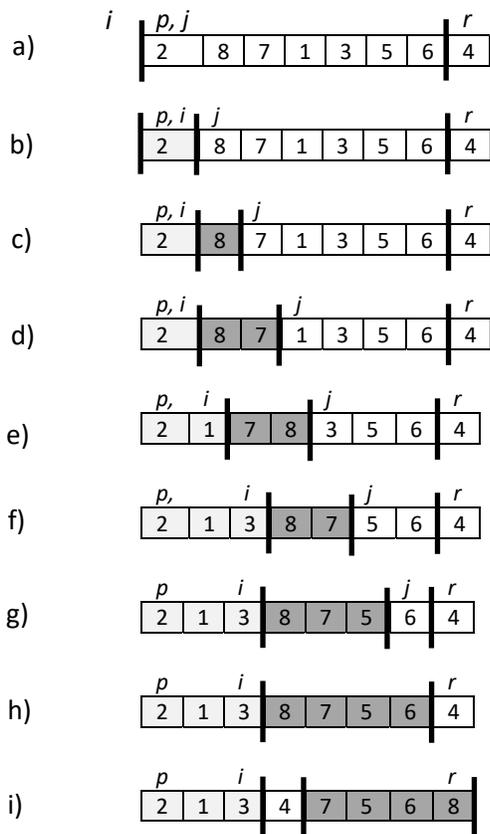


Figura 2.3 [1]

Para ordenar el arreglo completo se tiene que hacer la llamada a la función *QuickSort* () de la siguiente manera, **QuickSort(A,1,n)**, donde n es el tamaño de la lista o número de elementos del arreglo y 1 el índice del primer elemento del arreglo.

El tiempo de ejecución del algoritmo depende de los particionamientos que se realizan si están balanceados o no, lo cual depende del número de elementos involucrados en esta operación.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	19/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En la práctica, el algoritmo de ordenación *QuickSort* es el más rápido, su tiempo de ejecución promedio es $O(n \log(n))$, siendo en el peor de los casos $O(n^2)$ el cual es poco probable que suceda.

Cabe mencionar que

HeapSort

El método de ordenación *HeapSort* también es conocido con el nombre “montículo”. Un montículo es una estructura de datos que se puede manejar como un arreglo de objetos o también puede ser visto como un árbol binario con raíz cuyos nodos contienen información de un conjunto ordenado. Cada nodo del árbol corresponde a un elemento del arreglo. Figura 2.4.

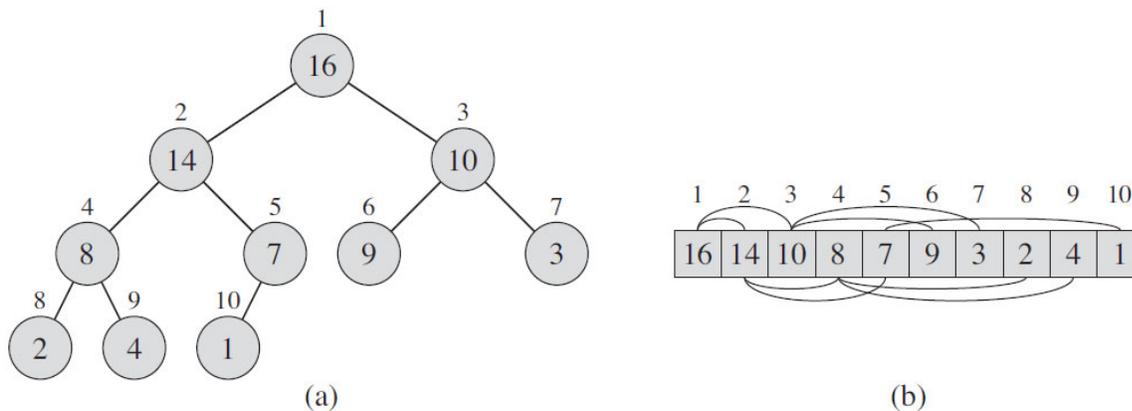


Figura 2.4 [1]

Un arreglo lineal A que representa un montículo es un objeto con 2 atributos:

longitudDeA: número de elementos en el arreglo.

TamañoHeapA: número de elementos en el montículo almacenados dentro de A y

$0 \leq \text{TamañoHeapA} \leq \text{longitudDeA}$.

La raíz del árbol será el primer elemento $A[1]$ y con el índice i de un nodo se pueden conocer los índices de sus padres, los nodos hijos a la izquierda y a la derecha, es decir:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	20/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Los índices pueden ser calculados utilizando las siguientes funciones en pseudocódigo[1]:

<i>Padre(i)</i>	<i>hIzq(i)</i>	<i>IDer(i)</i>
Inicio	Inicio	Inicio
retorna $\lfloor i/2 \rfloor$	retorna $2 * i$	retorna $2 * i + 1$
Fin	Fin	Fin

Para este tipo de estructuras **heap** hay dos tipos de árboles binarios, llamados **HeapMaximo** y **HeapMinimo**. Para un **HeapMaximo** la raíz de cada subárbol es mayor o igual que cualquiera de sus nodos restantes. Para un **HeapMinimo** la raíz de cada subárbol es menor o igual que cualquiera de sus hijos.

Para el algoritmo Heapsort se utiliza el **HeapMaximo** ya que un **HeapMinimo** se utiliza para las llamadas colas de prioridad.

Descripción del Algoritmo

El algoritmo consiste en forma general en construir un *heap*(montículo) y después ir extrayendo el nodo que queda como raíz del árbol en sucesivas iteraciones hasta obtener el conjunto ordenado.

De forma más detallada, si se construye un montículo *A* de tipo **HeapMaximo** representado en forma de un arreglo lineal donde sus elementos son $A[1], A[2], A[3], \dots, A[n]$. Primero se intercambian los valores $A[1]$ y $A[n]$ para tener siempre el máximo en el extremo $A[n]$ después se reconstruye el montículo con los elementos $A[1], A[2], A[3], \dots, A[n - 1]$ y se vuelven a intercambiar los valores $A[1]$ y $A[n - 1]$ para reconstruir nuevamente el montículo con los elementos $A[1], A[2], A[3], \dots, A[n - 2]$. El proceso se realiza de forma iterativa.

Para definir un algoritmo en pseudocódigo, se considera un arreglo lineal *A* para representar al **heap** de tipo **HeapMaximo**, el número de elementos en el arreglo *longitudDeA* y el número de elementos en el **heap** *TamañoHeapA*.

El algoritmo queda [1]:

OrdenacionHeapSort(A)

Inicio

construirHeapMaxIni(A)

Para $i = \text{longitudDeA}$ hasta 2 hacer

Intercambia($A[1], A[i]$)

$\text{TamañoHeapA} = \text{TamañoHeapA} - 1;$

MaxHeapify (A,1,TamañoHeap)

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	21/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Donde la función *construirHeapMaxini()* construye el *heap* inicial de forma que sea un **HeapMaximo**, *Intercambia()* es una función que intercambia de lugar los elementos $A[1]$ y $A[i]$; y *MaxHeapify()* permite que el *heap* modificado mantenga la propiedad de orden de un **HeapMaximo** es decir que el valor $A[i]$ que se encuentra ahora en raíz se acomode para que el árbol siga cumpliendo con que la raíz de cada subárbol es mayor o igual que cualquiera de sus nodos restantes.

La definición de la función *MaxHeapify()* es [1]:

MaxHeapify (A,i)

Inicio

L= hIzq(i)

R=hDer(i)

Si $L < \text{TamañoHeapA}$ y $A[L] > A[i]$

 posMax=L

En otro caso

 posMax = i

Fin Si

Si $R < \text{TamañoHeapA}$ y $A[R] > A[\text{posMax}]$ entonces

 posMax =R

Fin Si

Si posMax \neq i entonces

 Intercambia($A[i]$, $A[\text{posMax}]$)

 MaxHeapify(A ,posMax)

Fin Si

Fin

En esta función si $A[i]$ es mayor o igual que la información almacenada en la raíz de los subárboles izquierdo y derecho entonces el árbol con raíz en el nodo i es un **HeapMaximo** y la función termina. De lo contrario, la raíz de alguno de los subárboles tiene información mayor que la encontrada en $A[i]$ y es intercambiada con ésta, con lo cual se garantiza que el nodo i y sus hijos son **HeapMaximo**, pero, sin embargo el subárbol hijo con el cual se intercambié la información de $A[i]$ ahora puede no cumplir la propiedad de orden y por lo tanto, se debe llamar de forma recursiva a la función *MaxHeapify()* sobre el subárbol hijo con el cual se hizo el intercambio.

Construcción del Heap

Para la construcción del **heap** inicial se puede utilizar la función *MaxHeapify()* de abajo hacia arriba, para convertir el arreglo A de n elementos en un **HeapMaximo**; el pseudocódigo queda [1].

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	22/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

construirHeapMaxIni(A)

Inicio

TamañoHeapA=longitudDeA

Para $i=\lfloor \text{longitudDeA}/2 \rfloor$ hasta 1

MaxHeapify(A,i)

Fin Para

Fin

Desarrollo:

Actividad 1

Se proporcionan las funciones mencionadas en pseudocódigo para el algoritmo **Quick Sort** en Python. Se requiere utilizarlas para elaborar un programa que ordene una lista proporcionada por el profesor.

Las funciones en Python de los pseudocódigos descritos para el algoritmo Quick Sort son las siguientes:

```
def intercambia( A, x, y ):
    tmp = A[x]
    A[x] = A[y]
    A[y] = tmp

def Particionar(A,p,r):
    print (A)
    x=A[p]
    print (x)
    i=p
    for j in range(p+1,r+1):
        if (A[j]<=x):
            i=i+1
            intercambia(A,i,j)
    intercambia(A,i,p)
    return i

def QuickSort(A,p,r):
    if ( p < r ):
        q=Particionar(A,p,r)
        QuickSort(A,p,q-1)
        QuickSort(A,q+1,r)
```

Considerar que en esta implementación el índice de inicio de la lista es 0 y no 1.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	23/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Una vez elaborado el programa, responder a las siguientes preguntas.

¿Qué modificaciones se tienen que hacer para ordenar la lista en orden descendente? Describir y modificar el programa. _____

¿Qué se cambiaría en la función *Particionar()* para que en este proceso se tome al inicio el elemento pivote como el primer elemento del subarreglo? Describir y modificar el programa. _____

Actividad 2

Se proporcionan las funciones mencionadas en pseudocódigo para el algoritmo **Heap Sort** en Python. Se requiere utilizarlas para elaborar un programa que ordene la misma lista proporcionada por el profesor.

Las funciones en Python de los pseudocódigos descritos para el algoritmo Heap Sort son las siguientes:

```
import math
def hIzq(i):
    return 2*i+1

def hDer(i):
    return 2*i+2

def intercambia(A,x,y):
    tmp=A[x]
    A[x]=A[y]
    A[y]=tmp

def maxHeapify(A,i,tamanoHeap):
    L=hIzq(i)
    R=hDer(i)
    if(L<=(tamanoHeap-1) and A[L]>A[i]):
        posMax=L
    else:
        posMax=i
    if(R<=(tamanoHeap-1) and A[R]>A[posMax]):
        posMax=R
    if(posMax != i):
        intercambia(A,i,posMax)
        maxHeapify(A,posMax,tamanoHeap)
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	24/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

def construirHeapMaxIni(A,tamanoHeap):
    for i in range(math.ceil((tamanoHeap-1)/2), -1, -1):
        maxHeapify(A,i,tamanoHeap)

def ordenacionHeapSort(A,tamanoHeap):
    construirHeapMaxIni(A,tamanoHeap)
    for i in range(len(A)-1,0,-1):
        intercambia(A,0,i)
        tamanoHeap=tamanoHeap-1
        maxHeapify(A,0,tamanoHeap)

```

Hay que considerar que en esta implementación el índice de inicio de la lista es 0 y no 1 como en los pseudocódigos.

Una vez implementado el programa, ¿Qué cambios se harían para usar un **Heap Mínimo** en lugar de un **Heap Máximo**? Describir y modificar el programa para estos cambios. _____

Actividad 3

Ejercicios propuestos por el profesor.

Referencias

[1] CORMEN, Thomas, LEISERSON, Charles, et al.
Introduction to Algorithms
3rd edition
MA, USA
The MIT Press, 2009

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	25/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 3

Algoritmos de ordenamiento parte 3

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	26/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 3

Estructura de datos y Algoritmos II

Algoritmos de Ordenamiento. Parte 3.

Objetivo: El estudiante conocerá e identificará la estructura de los algoritmos de ordenamiento *Counting Sort* y *Radix Sort*.

Actividades

- Implementar el algoritmo *Counting Sort* en algún lenguaje de programación para ordenar una secuencia de datos.
- Implementar el algoritmo *Radix Sort* en algún lenguaje de programación para ordenar una secuencia de datos.

Antecedentes

- Análisis previo de los algoritmos en clase teórica.
- Manejo de arreglos o listas, diccionarios, estructuras de control y funciones en Python 3.

Introducción

Counting Sort

Es un algoritmo que no se basa en comparaciones y lo que hace es contar el número de elementos de cada clase en un rango de $0 - k$ para después ordenarlos determinando para cada elemento de entrada el número de elementos menores a él. Por lo tanto, la lista o arreglo a ordenar solo pueden utilizar elementos que sean contables (enteros).

Para la descripción del algoritmo se asumen 3 arreglos lineales:

- El arreglo de entrada A a ordenar con n elementos
- Un arreglo B de n elementos, para guardar la salida ya ordenada
- Un arreglo C para almacenamiento temporal de k elementos

El primer paso consiste en averiguar el rango de valores de los datos a ordenar ($0 - k$). Después se crea el arreglo C de números enteros con tantos elementos como valores haya en el intervalo $[0, k]$, y a cada elemento de C se le da el valor inicial de 0 (0 apariciones del elemento i donde toma valores de $i = 0, \dots, k$).

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	27/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En el arreglo A de la figura 3.1 se tiene que el rango de valores de los elementos es de 0 a 5, por lo tanto, el arreglo C tendrá 6 elementos con índices de 0 a 5 el cual se inicializa en 0.

	1	2	3	4	5	6	7	8		0	1	2	3	4	5
A	2	5	3	0	2	3	0	3	C	0	0	0	0	0	0

Figura 3.1

Después se recorren todos los elementos del arreglo A a ordenar y se cuenta el número de apariciones de cada elemento para almacenarlo en C . Figura 3.2

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	0	1	2	3	4	5		
C	2	0	2	3	0	1		

Figura 3.2 [1]

Posteriormente se determina para cada elemento $C[i]$ ($i = 0, \dots, k$) cuántos elementos son menores o iguales a él.

En la figura 3.3 se puede observar que 2 elementos son menores o iguales a 1, 4 elementos son menores o iguales a 2, 7 elementos son menores o iguales a 3, etc.

	0	1	2	3	4	5
C	2	2	4	7	7	8

Figura 3.3

Para terminar, se coloca cada elemento del arreglo A , ($A[j]$, $j = n, \dots, 1$) en la posición correcta en el arreglo de salida B , de tal forma que cada elemento de entrada se coloca en la posición del número de elementos menores o iguales a él ($B[C[A[j]]]$). Cada vez que se coloca un elemento en B , se decrementa el valor de $C[A[j]]$.

Así el elemento $A[8] = 3$ se coloca en $B[C[A[8]] = B[C[3]] = B[7]$. Figura 3.4.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	28/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

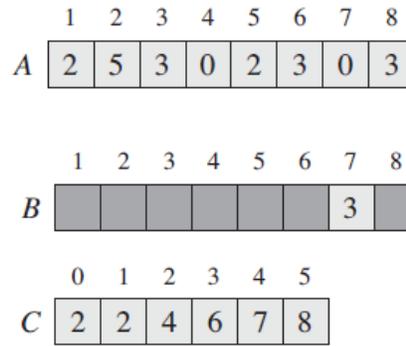


Figura 3.4

El elemento $A[7]$ se coloca en $B[C[A[7]]] = B[C[0]] = B[2]$. Figura 3.5.

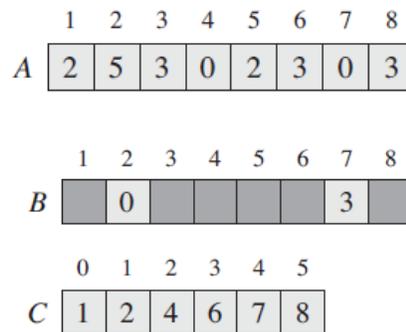


Figura 3.5

Una vez que se revisan todos los elementos de A tenemos que el arreglo ordenado en B queda como en la figura 3.6.



Figura 3.6

Un pseudocódigo del algoritmo es [1]:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	29/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

CountingSort(A,B,k)

Inicio

Formar C de k elementos

Para i=0 hasta i=k

C[i]=0

Fin Para

Para j=1 hasta número de elementos de A

C[A[j]]=C[A[j]]+1

Fin Para

Para i=1 hasta i=k

C[i]=C[i]+C[i-1]

Fin para

Para j= número de elementos de A hasta 1

B[C[A[j]]]=A[j]

C[A[j]]=C[A[j]]-1

Fin Para

Fin

El tiempo de ejecución para este algoritmo es $\theta(n + k)$ dado que para el primer y tercer ciclo se toma un tiempo de $\theta(k)$ y para el segundo y último un tiempo de $\theta(n)$.

Counting Sort es un algoritmo estable, es decir, si el ordenamiento se hace con base en una relación de orden y en esa relación dos elementos son equivalentes, entonces se preserva el orden original entre los elementos equivalentes.

Radix Sort

El método de ordenamiento *Radix Sort* también llamado ordenamiento por residuos puede utilizarse cuando los valores a ordenar están compuestos por secuencias de letras o dígitos que admiten un orden lexicográfico.

El algoritmo ordena utilizando un algoritmo de **ordenación estable**, las letras o dígitos de forma individual, partiendo desde el que está más a la derecha (menos significativo) y hasta el que se encuentra más a la izquierda (el más significativo). **Nota:** a cada letra o dígito se le asigna una llave o código representado por un número entero, el cual se utiliza para el ordenamiento de cada elemento que conforma el valor original.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	30/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Por ejemplo, en [1] se planea que una línea aérea proporciona números de confirmación diseñados con cadenas formadas con 2 caracteres donde cada carácter es un dígito o una letra que puede tomar 36 valores (26 letras y 10 dígitos) y así hay 36^2 posibles códigos.

Para cada carácter de los 36 se genera un código numérico entero de 0-36. Figura 3.7.

carácter	código
0	0
1	1
.	.
.	.
9	10
A	11
B	12
.	.
.	.
Z	35

Figura 3.7

Si se tienen los códigos de confirmación $\{F6, E5, R6, X6, X2, T5, F2, T3\}$ y se utiliza un algoritmo de ordenación estable en el carácter que se encuentra más a la derecha se obtiene la lista parcialmente ordenada de códigos $\{X2, F2, T3, E5, T5, F6, R6, X6\}$. Ahora si se ordena utilizando el mismo algoritmo de ordenamiento estable, pero sobre el carácter que se encuentra más a la izquierda se obtiene la lista $\{E5, F2, F6, R6, T3, T5, X2, X6\}$.

Procesa las letras o dígitos de forma individual partiendo desde el dígito menos significativo y hasta alcanzar el dígito más significativo.

Si los códigos de confirmación se forman con 6 caracteres y se tiene la lista de códigos $\{XI7FS6, PL4ZQ2, JI8FR9, XL8FQ6, PY2ZR5, KV7WS9, JL2ZV3, KI4WR2\}$, el ordenamiento del carácter más a la derecha hacia el de más a la izquierda se muestra en la Figura 3.2[1].

<i>i</i>	Resultado de las listas ordenadas con el <i>i</i> -ésimo carácter
1	$\langle PL4ZQ2, KI4WR2, JL2ZV3, PY2ZR5, XI7FS6, XL8FQ6, JI8FR9, KV7WS9 \rangle$
2	$\langle PL4ZQ2, XL8FQ6, KI4WR2, PY2ZR5, JI8FR9, XI7FS6, KV7WS9, JL2ZV3 \rangle$
3	$\langle XL8FQ6, JI8FR9, XI7FS6, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, JL2ZV3 \rangle$
4	$\langle PY2ZR5, JL2ZV3, KI4WR2, PL4ZQ2, XI7FS6, KV7WS9, XL8FQ6, JI8FR9 \rangle$
5	$\langle KI4WR2, XI7FS6, JI8FR9, JL2ZV3, PL4ZQ2, XL8FQ6, KV7WS9, PY2ZR5 \rangle$
6	$\langle JI8FR9, JL2ZV3, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, XI7FS6, XL8FQ6 \rangle$

Figura 3.8 [1]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	31/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

El algoritmo en pseudocódigo del *Radix Sort* [2] es:

RadixSort(A,d)

Inicio

Para $i=1$ hasta $i=d$

Ordenamiento de A en el dígito i

Fin

Donde A es una lista de n elementos, d es el número de dígitos o caracteres que tienen los elementos de A , si $i = 1$ se refiere al dígito o carácter colocado más a la derecha y cuando $i = d$ al que está más a la izquierda. El ordenamiento se realiza con algún algoritmo estable como por ejemplo *Counting Sort*.

Desarrollo:

Actividad 1

Se proporciona la función mencionada en pseudocódigo para el algoritmo *Counting Sort* en Python. Se requiere utilizarla para elaborar un programa que ordene una lista de enteros con valores comprendidos entre 0 y k . El valor de k es propuesto, pero debe ser mayor a 10 y menor a 30.

La función en Python del pseudocódigo descrito para *Counting Sort* es la siguiente, notar que aquí los índices de inicio de todas las listas es 0:

```
def countingSort(A,k):
    C=[0 for _ in range(k+1)]
    B=[0 for _ in range(len(A))]
    for j in range(0,len(A)):
        C[A[j]]=C[A[j]]+1

    for i in range(1,k+1):
        C[i]=C[i]+C[i-1]

    for j in range(len(A)-1,-1,-1):
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
    return B
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	32/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Una vez realizado el programa, indicar un procedimiento para obtener el valor de k , para saber el rango de valores de los elementos de la lista a ordenar. Además, contestar las siguientes preguntas

¿La lista se ordena en orden ascendente o descendente? ¿Por qué?

¿Es posible ordenar la lista en orden inverso?, ¿Cómo se haría? Realizar respectivos cambios en el programa.

Actividad 2

Para la explicación del algoritmo *Radix Sort* se utilizó el ejemplo de cómo ordenar claves de confirmación de una aerolínea, donde cada clave está formada con seis caracteres. Aquí se muestra una solución al problema en Python. Además, como algoritmo de ordenación estable se utiliza el *Counting Sort* visto al inicio del documento con unas pequeñas modificaciones.

Se pide probar el código proporcionado con la secuencia

{X17FS6, PL4ZQ2, J18FR9, XL8FQ6, PY2ZR5, KV7WS9, JL2ZV3, KI4WR2} y colocar la impresión correspondiente para ver las listas parcialmente ordenadas como en la Figura 3.8.

Una vez terminado lo anterior responder a la siguiente pregunta.

¿Qué cambios se harían para ordenar ahora del más significativo al menos significativo (izquierda a derecha)?
Describir y realizar las modificaciones correspondientes al programa. _____

Cabe mencionar que para la implementación en Python de la solución al problema de la ordenación de claves de la aerolínea mediante el algoritmo *RadixSort* y uso de *Counting Sort*, se requirió contar el número de caracteres que conforman la clave y formar una lista donde cada elemento contiene información de la clave y el código correspondiente al carácter en análisis (figura 3.7). El código de la figura 3.7 se implementó con un diccionario.

Las funciones que conforman la solución y la llamada a la función que da la solución se muestran abajo:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	33/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```
def formalistaConClaves(B,numCar):
    Btmp=[]
    D={}
    j=0
    for i in range(48,58):
        D[chr(i)]=j
        j+=1
    for i in range(65,91):
        D[chr(i)]=j
        j+=1
    for i in range(len(B)):
        Btmp.append([B[i]]*2)
        A3=list(B[i])
        Btmp[i][1]=D[A3[numCar-1]]
    return Btmp
```

```
def countingSort2(A,k):
    C=[0 for _ in range(k+1)]
    B=[list(0 for _ in range(2)) for _ in range(len(A))]
    for j in range(0,len(A)):
        C[A[j][1]]=C[A[j][1]]+1

    for i in range(1,k+1):
        C[i]=C[i]+C[i-1]

    for j in range(len(A)-1,-1,-1):
        B[C[A[j][1]]-1][1]=A[j][1]
        B[C[A[j][1]]-1][0]=A[j][0]
        C[A[j][1]]=C[A[j][1]]-1
    return B
```

```
def obtenerElemSinClaves(E):
    Elem=[]
    for i in range(0,len(E)):
        Elem.append(E[i][0])
    return Elem
```

```
def radixSort(A):
    numCar=len(A[1])
    for i in range(numCar,0,-1):
        cc=formalistaConClaves(A,i)
        ordenado=countingSort2(cc,36)
        A=obtenerElemSinClaves(ordenado)

    return A
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	34/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Para observar el funcionamiento se crea una lista con las claves y después se llama a la función *radixSort()* como se observa en la definición de la siguiente función main().

```
def main():
    lista=['XI7FS6', 'PL4ZQ2', 'JI8FR9', 'XL8FQ6', 'PY2ZR5', 'KV7WS9', 'JL2ZV3', 'KI4WR2' ]
    A=radixSort(lista)
    print('\n\nLista ordenada:\n',A)

main()
```

Actividad 3

Ejercicios propuestos por el profesor.

Referencias

[1] CORMEN, Thomas
Algorithms Unlocked
Cambridge MA, USA
The MIT Press, 2013

[2] CORMEN, Thomas, LEISERSON, Charles, et al.
Introduction to Algorithms
3rd edition
MA, USA
The MIT Press, 2009

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	35/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 4

Algoritmos de búsqueda parte 1

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	36/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 4

Estructura de datos y Algoritmos II

Algoritmos de Búsqueda. Parte 1.

Objetivo: El estudiante identificará el comportamiento y características de algunos algoritmos de búsqueda por comparación de llaves.

Actividades

- Implementar el algoritmo iterativo y recursivo de búsqueda lineal en algún lenguaje de programación para localizar alguna llave o valor en una secuencia de datos.
- Implementar el algoritmo iterativo y recursivo de búsqueda binaria en algún lenguaje de programación para localizar alguna llave o valor en una secuencia de datos.

Antecedentes

- Análisis previo de los algoritmos en clase teórica.
- Conocimiento de algún algoritmo de ordenamiento.
- Manejo de arreglos o listas, estructuras de control y funciones en Python 3.

Introducción

Recuperar información de una computadora es una de las actividades más útiles e importantes, muy comúnmente se tiene un nombre o llave que se quiere encontrar en una estructura de datos tales como una lista u arreglo.

Al proceso de encontrar un dato específico llamado llave en alguna estructura de datos, se denomina búsqueda. Este proceso termina exitosamente cuando se localiza el elemento que contienen la llave buscada, o termina sin éxito, cuando no aparece ningún elemento con esa llave.

Los algoritmos de búsqueda se pueden clasificar en:

- Búsqueda por comparación
- Búsqueda por transformación de llaves

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	37/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

En este documento se describen dos de los algoritmos de búsqueda por comparación: la búsqueda lineal o secuencial y la búsqueda binaria, que son dos de las técnicas más comunes; la primera por sencilla y la otra por eficiente.

Búsqueda Lineal

En este algoritmo se empieza a buscar desde la primera posición de la lista, comparando el elemento con la llave y si no se encuentra continúa verificando las siguientes posiciones hasta encontrarlo, entonces el algoritmo termina. Así el elemento a buscar puede estar en el primer elemento, el último o entre ellos.

A continuación, se muestra un algoritmo en pseudocódigo de una función que realiza la búsqueda secuencial del elemento x en una lista implementada como un arreglo lineal $A[0,1,2, \dots, n-1]$ de n elementos. El cual retorna la posición o índice del arreglo donde se encuentra el elemento x a buscar, y el valor de -1 si x no fue encontrado.

BúsquedaLineal (A,n,x)

Inicio

 encontrado=-1

 Para $k=0$ hasta $n-1$

 Si $x==A[k]$

 encontrado = k

 Fin Si

 Fin Para

 Retorna encontrado

Fin

Como se puede observar en la función BúsquedaLineal(), el número de iteraciones siempre es igual al tamaño del arreglo A , independientemente de dónde se encuentre el elemento a buscar y como todas las operaciones del interior de la estructura de repetición tienen una complejidad constante, la complejidad de este algoritmo de búsqueda secuencial es de $O(n)$.

Existen varias mejoras al algoritmo, una de ellas es no esperar a revisar todos los elementos del arreglo, si el buscado ya se encontró, entonces se tiene que terminar. Una forma de conseguirlo es revisar en cada iteración k , si el elemento a buscar x es ya igual al elemento $A[K]$, y si lo es, entonces retornar la posición del elemento encontrado y terminar. A continuación, se muestra una posible función en pseudocódigo donde se considera esta mejora [1].

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	38/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

BúsquedaLinealMejorado

Inicio

 encontrado=-1

 Para k=0 hasta n-1

 Si $A[k]==x$

 encontrado= k

 Salir de la estructura de repetición

 Fin Si

 Fin Para

 retorna encontrado

Fin

También se tiene la llamada búsqueda secuencial con centinela que asegura siempre encontrar el elemento a buscar. Lo que se hace es situar el valor que se está buscando al final del arreglo $A[n - 1]$ (centinela), guardando lo que se encuentra inicialmente ahí en algún lugar temporal. Así en la estructura de repetición que va revisando todos los elementos de A siempre se obtiene un valor del índice k donde se encuentra el elemento.

Al término de las iteraciones se regresa el valor inicial del elemento $A[n - 1]$ a su lugar y solo resta verificar si éste o el posicionado en $k < n - 1$ es el valor buscado, si no es ninguno de ellos, entonces, el elemento no está en la lista.

Lo anterior se puede representar en la siguiente función en pseudocódigo [1]

BusquedaLinealCentinela(A, n, x)

 tmp= $A[n-1]$

$A[n-1]=x$

 k=0

 Mientras $A[k]$ sea diferente de x

 k=k+1

 Fin Mientras

$A[n-1]=tmp$

 Si $k < n-1$ o $A[n-1]==x$

 retorna k

 En otro caso

 retorna -1

 Fin Si

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	39/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En estos dos últimos algoritmos el tiempo de ejecución en el mejor de los casos se da cuando el valor a buscar se encuentra en el primer elemento de la lista y es $\theta(1)$. En el peor de los casos cuando está en la última posición y es $\theta(n)$.

Existe una versión recursiva del algoritmo de búsqueda lineal, donde se tendrá un caso base de fracaso que se da cuando se rebasa el número de elementos de la lista y un caso base de éxito cuando después de algunas llamadas recursivas el elemento analizado de la lista es igual a la llave. Un algoritmo en pseudocódigo se puede plantear como se muestra en la siguiente función.

```

BusquedaLinealRecursiva (A,x,ini,fin)
Inicio
  Si ini > fin
    encontrado=-1
  Si no
    Si A[ini]==x
      encontrado=ini
    Si no
      encontrado = BusquedaLinealRecursiva(A, x,ini+1,fin)
  Fin Si no
Fin Si no
retorna encontrado
Fin

```

Búsqueda Binaria

El algoritmo de búsqueda binaria o también llamado dicotómica es eficiente para encontrar un elemento en una lista ya ordenada y es un ejemplo de la técnica divide y conquista. Es como buscar una palabra en un diccionario, donde dependiendo de la primera letra de la palabra a buscar se abre el libro cerca del principio, del centro o al final. Se puede tener suerte y haber abierto la página correcta o bien moverse a páginas posteriores o anteriores del libro.

Entonces en el algoritmo se divide repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una. La estrategia consiste en comparar una llave con el elemento de en medio de la lista, si es igual entonces se encontró el elemento, sino, cuando x es menor que el elemento del medio se aplica la misma estrategia a la lista de la izquierda y si x es mayor, a la lista de la derecha.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	40/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Para describir el algoritmo con más detalle, se supone una lista ordenada de forma ascendente que está almacenada en un arreglo unidimensional $A[0, \dots, n - 1]$ de n elementos y se quiere encontrar la llave x .

- Primero se calcula el índice del punto medio del arreglo utilizando los índices de inicio (izquierdo) y fin (derecho) de la lista o sublista de búsqueda, $medio = \left\lfloor \frac{indiceIzquierdo + indiceDerecho}{2} \right\rfloor$, para poder dividir la lista en dos sub-listas Figura 4.1.

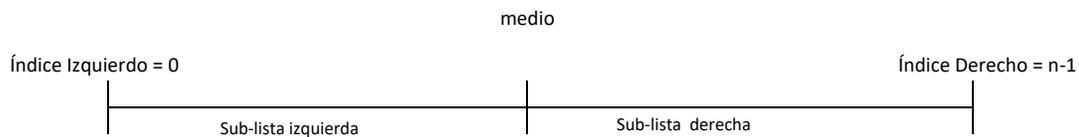


Figura 4.1

- Después se compara el punto *medio* con la llave a buscar x ,
 - Si son iguales, entonces la llave fue encontrada, y el algoritmo termina. Figura 4.2.

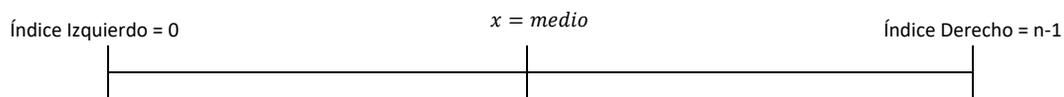


Figura 4.2

- Si el valor del arreglo A con índice *medio* es mayor a la llave x ($A[medio] > x$), se descarta la sub-lista de la derecha incluyendo el punto *medio* y se regresa al paso 1, donde la nueva sub-lista de búsqueda mantiene el índice izquierdo y el índice derecho cambia a $medio - 1$. Figura 4.3



Figura 4.3

- Si el valor del arreglo A con índice *medio* es menor a la llave x ($A[medio] < x$), se descarta la sub-lista de la izquierda incluyendo el punto *medio* y se regresa al paso 1, donde la nueva sub-lista de búsqueda mantiene el índice derecho y el índice izquierdo cambia a $medio + 1$. Figura 4.4.

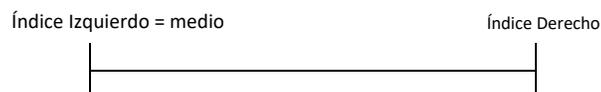


Figura 4.4

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	41/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Si en algún momento la sub-lista de búsqueda tiene longitud 0 o negativa significa que el valor buscado no se encuentra en la lista.

Ejemplo: Se quiere buscar el elemento $x = 18$ en un arreglo unidimensional de 13 elementos que contiene los siguientes datos, los cuales ya están ordenados:

0	1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	18	20	25	27	30	35	40	45	47

Donde *índice izquierdo* = 0 e *índice derecho* = 12.

$medio = \frac{12}{2} = 6$, entonces $A[medio] = 25$.

Se compara 18 con 25, como $18 < 25$, ahora se trabaja con la sub-lista izquierda:

0	1	2	3	4	5
10	12	13	14	18	20

Donde *índice izquierdo* = 0 e *índice derecho* = $6 - 1 = 5$.

$medio = \frac{5}{2} = 2$, entonces $A[medio] = 13$.

Se compara 18 con 13, como $18 > 13$, ahora se trabaja con la sub-lista derecha:

3	4	5
14	18	20

Donde *índice izquierdo* = $medio + 1 = 3$ e *índice derecho* = 5.

$medio = \frac{8}{2} = 4$, entonces $A[medio] = 18$.

Se observa que $A[medio]$ es igual $x = 18$, entonces el algoritmo termina.

Una función del algoritmo en pseudocódigo de la búsqueda binaria descrita es:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	42/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

BusquedaBinariaIterativa(A,x,indiceIzq,indiceDer)

Inicio

Encontrado= -1

Mientras indiceIzq <= indiceDer

Medio=[(indiceIzq + IndiceDer)/2]

Si x == A[Medio] entonces

Encontrado=Medio

Si no

Si A[medio] < x

indiceIzq=medio +1

Si no

indiceDer=medio-1

Fin Si no

Fin Si no

Fin Mientras

Retorna Encontrado

Fin

El algoritmo de búsqueda binaria también tiene un enfoque recursivo. Donde se tiene un caso base de fracaso, que sucede cuando se sobrepasa el número de elementos de la lista y un caso base de éxito, cuando después de algunas llamadas recursivas el elemento $A[\text{medio}]$ es igual a la llave x .

A continuación, se muestra una función en pseudocódigo que ilustra la forma recursiva:

BusquedaBinariaRecursiva(A,x,indiceIzq,indiceDer)

Inicio

Si indiceIzquierdo > indiceDerecho y x es diferente de A[indiceDerecho]

Retorna -1

Fin Si

medio = [(indiceIzq + IndiceDer)/2]

Si x==A[medio]

Retorna medio

En otro caso

Si x < A[medio]

Retorna BusquedaBinariaRecursiva(A,x,indiceIzquierdo,medio)

En otro caso

Retorna BusquedaBinariaRecursiva(A,x,medio, indiceDerecho)

Fin Si

Fin Si

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	43/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En ambos algoritmos en pseudocódigo (iterativo y recursivo) se retorna la posición donde se encuentra el elemento o bien -1 si no se encuentra. También en ambos el tiempo de ejecución es $\theta(\log n)$.

Desarrollo

Actividad 1.1

A continuación, se proporciona la implementación en Python de los pseudocódigos de las funciones para búsqueda lineal presentados en este documento. Se pide realizar un programa que utilice las tres funciones para buscar en una lista dada por el profesor, un valor o llave proporcionado en la entrada estándar. Se debe mostrar el número de iteraciones que realiza cada función, así como el índice donde se localizó la llave. Las funciones en Python son las siguientes:

```
#búsqueda Lineal o secuencial
def busquedaLineal(A, n, x):
    encontrado=-1
    for k in range (n):
        if A[k] == x:
            encontrado=k
    return encontrado
```

```
#Búsqueda Lineal Mejorada
def busquedaLinealMejorada(A, n, x):
    encontrado=-1
    for k in range (n):
        if A[k] == x:
            encontrado=k
            break
    return encontrado
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	44/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

#Búsqueda Lineal con Centinela
def busquedaLinealCentinela (A, n, x) :
    tmp=A[n]
    A[n]=x
    k=0
    while A[k] != x:
        k=k+1
    print (k)
    A[n]=tmp
    if k < n or A[n]==x:
        return k
    else:
        return -1
    return encontrado

```

Probar su programa con diferentes llaves, y describir el porqué de los resultados obtenidos.

Actividad 1.2

Implementar la función recursiva de la búsqueda lineal utilizando el pseudocódigo proporcionado en este documento.

Actividad 2

Abajo se muestran las funciones en Python de los pseudocódigos para búsqueda binaria (solución iterativa y recursiva) mencionados en esta guía. Se requiere implementar un programa utilizando estas funciones y alguna de las de ordenamiento vistas en las guías anteriores para buscar un valor en la lista proporcionada en la actividad 1.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	45/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```

#Búsqueda Binaria Iterativa
import math
def BusquedaBinIter(A, x, izquierda, derecha):

    while izquierda <= derecha:
        medio = math.floor((izquierda+derecha)/2)
        if A[medio] == x:
            return medio
        elif A[medio] < x:
            izquierda = medio+1
        else:
            derecha = medio-1
    return -1

```

```

#Búsqueda binaria Recursiva
import math
def BusquedaBinRecursiva(A, x, izquierda, derecha):
    if izquierda > derecha :
        return -1
    medio = math.floor((izquierda+derecha)/2)
    print (medio)

    if A[medio] == x:
        return medio

    elif A[medio] < x:

        return BusquedaBinRecursiva(A, x, medio+1, derecha)
    else:
        return BusquedaBinRecursiva(A, x, izquierda, medio-1)

```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	46/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Actividad 3

Ejercicios propuestos por el profesor.

Referencias

[1] CORMEN, Thomas
Algorithms Unlocked
Cambridge MA, USA
The MIT Press, 2013

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	47/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 5

Algoritmos de búsqueda parte 2

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	48/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 5

Estructura de datos y Algoritmos II

Algoritmos de Búsqueda. Parte 2.

Objetivo: El estudiante conocerá e identificará algunas de las características necesarias para realizar búsquedas por transformación de llaves.

Actividades

Implementar la búsqueda por transformación de llaves utilizando alguna técnica de resolución de colisiones en algún lenguaje de programación.

Antecedentes

- Análisis previo de los algoritmos en clase teórica.
- Manejo de arreglos o listas, estructuras de control y funciones en Python 3.

Introducción

Un diccionario es un conjunto de pares (llave, valor), siendo una abstracción que vincula un dato con otro. En un diccionario se pueden realizar operaciones de búsqueda, inserción y borrado de elementos dada una llave. Una tabla hash es una estructura de datos para implementar un tipo de dato abstracto (TDA) diccionario.

En el método de búsqueda por transformación de llaves, los datos son organizados con ayuda de una tabla hash, la cual permite que el acceso a los datos sea por una llave que sirve para obtener la posición donde están guardados los datos que se buscan. Se utiliza una función que transforma la llave o dato clave en una dirección o posición dentro de la estructura (tabla), dicha función de transformación se conoce como **función hash**.

Así, para determinar si un elemento con llave x está en la tabla, se aplica la función hash h a x ($h(x)$) y se obtiene la dirección del elemento en la tabla. Esto es si la función hash se expresa como $h: U \rightarrow \{0, 1, 2, \dots, m - 1\}$ se dice que h mapea el universo de llaves U en la posición de la tabla hash $T[0, 1, 2, \dots, m - 1]$ donde m es el tamaño de la tabla y es típicamente menor que U . Figura 5.1.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	49/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

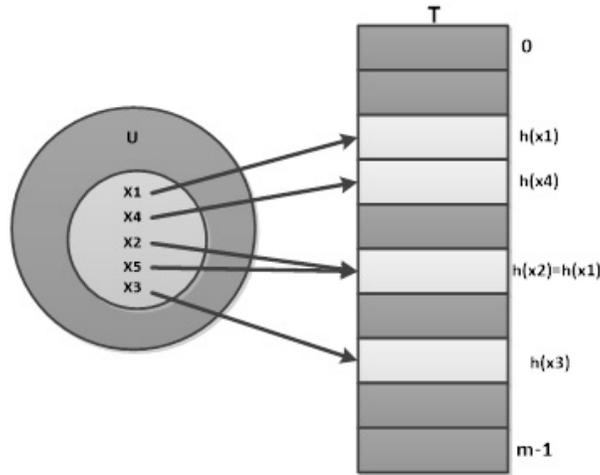


Figura 5.1 [1]

Si el universo de llaves es pequeño y todos los elementos tienen llave distinta, se realiza un direccionamiento o asignación directa, donde a cada posición en la tabla le corresponde una llave del universo U . Figura 5.2. La búsqueda y las otras funciones de diccionario se pueden escribir de forma simple como:

Búsqueda_DDirecto($T, llave$)
 Inicio
 Fin Retorna $T[llave]$

Agregar_DDirecto($T, valor$)
 Inicio
 Fin $T[llave.valor]=valor$

Borra_DDirecto($T, valor$)
 Inicio
 Fin $T[llave.valor]=NULL$

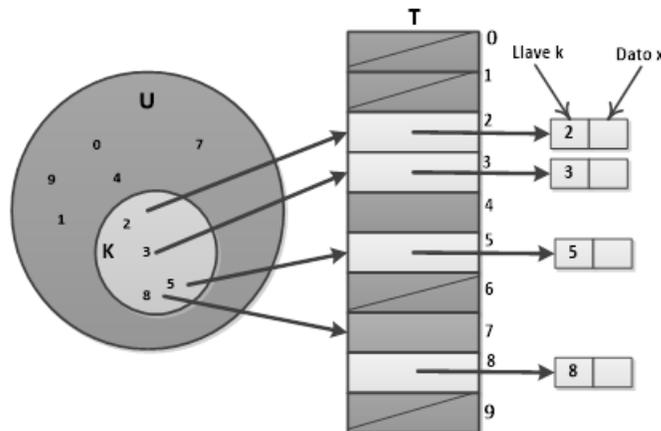


Figura 5.2

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	50/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En ocasiones se puede generar una **colisión**, que se define como una misma dirección para dos o más llaves distintas. Figura 5.1. Afortunadamente hay algunas técnicas para resolver el conflicto de las colisiones, aunque lo ideal sería no tener este problema.

Una función hash ideal debería ser biyectiva, es decir, que a cada elemento le corresponda un índice o dirección, y que a cada dirección le corresponda un elemento, pero no siempre es fácil encontrar esa función.

Entonces para trabajar con este método de búsqueda se necesitan principalmente:

- 1) Calcular el valor de la función de transformación (función hash), la cual transforma la llave de búsqueda en una dirección o entrada a la tabla.
- 2) Disponer de un método para tratar las colisiones. Dado que la función hash puede generar la misma dirección o posición en la tabla para diferentes llaves.

En lo siguiente, se mencionan algunas formas de cómo plantear una función hash y de cómo tratar las colisiones.

Universo de llaves

Para el diseño de una función hash se asume que el universo de llaves es el conjunto de números naturales o enteros positivos y si las llaves a usar no lo son, primero se busca la manera de expresarlos como tal.

Por ejemplo, si las llaves son letras de algún alfabeto, se puede asignar a cada letra el valor entero de su posición en el alfabeto o su valor correspondiente en algún código (lo que se puede expresar como $ord(x)$).

Si las llaves son cadenas de caracteres $c_0c_1c_2 \dots c_{n-1}$

El natural se puede obtener como $\sum_{i=0}^{n-1} ord(c_i)$ que es la suma de los valores numéricos asignados a cada carácter.

También se puede obtener la llave x en forma de entero positivo como:

$$K = \sum_{i=1}^n Clave[i](p[i])$$

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	51/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Donde n es el número de caracteres de la llave, $Clave[i]$ corresponde con la representación ASCII del i -ésimo carácter, y $p[i]$ es un entero procedente de un conjunto de pesos generados aleatoriamente para $1 \leq i \leq n$. La ventaja de utilizar pesos es que dos conjuntos de pesos diferentes $p_1[i]$ y $p_2[i]$ llevan a dos funciones de transformación diferente $h_1(x)$ y $h_2(x)$ [3].

Para explicar algunas de las funciones hash se asume que el universo de llaves es el conjunto de los naturales.

Funciones Hash

Una buena función hash realiza una distribución o asignación de direcciones uniforme, es decir, para cada llave de entrada cualquiera de las posibles salidas tiene la misma probabilidad de suceder.

En la literatura se han estudiado diferentes funciones de transformación, en esta guía se explican solo los tres esquemas planteados en [1] para el diseño de una buena función hash. En dos de los esquemas se utilizan las operaciones de multiplicación y división que son heurísticas mientras que en el tercer esquema se utiliza el llamado *hashing* universal que es un procedimiento aleatorio.

Método de división

En este método se mapea una llave x en una de las celdas de la tabla tomando el residuo de x dividido entre m .

$$h(x) = x \text{ mod } m$$

Se recomienda que m no sea potencia de dos. A esta operación se le llama módulo.

Ejemplo [2]. Suponer que se tienen ocho estudiantes en una escuela y sus números de cuenta son

197354864, 933185952, 132489973, 134152056, 216500306, 106500306, 216510306 y 197354865.

Se quiere almacenar la información de cada estudiante en una tabla hash en orden.

Entonces sean las llaves $x_1 = 197354864$, $x_2 = 933185952$, $x_3 = 132489973$, $x_4 = 134152056$, $x_5 = 216500306$, $x_6 = 106500306$, $x_7 = 216510306$, y $x_8 = 197354865$.

Si la tabla hash es de tamaño 13 y esta indexada del 0,1,2,3, ...,12 se define la función hash

$h: \{x_1, x_2, x_3, \dots, x_8\} \rightarrow \{0,1,2, \dots,13\}$ como $h(x_i) = x_i \% 13$ (% representa el operador módulo) y aplicando la función a las llaves se obtiene lo siguiente Tabla 5.1:

$h(x_1) = h(197354864) = 197354864 \% 13 = 5$	$h(x_5) = h(216500306) = 216500306 \% 13 = 9$
$h(x_2) = h(933185952) = 933185952 \% 13 = 10$	$h(x_6) = h(106500306) = 106500306 \% 13 = 3$
$h(x_3) = h(132489973) = 132489973 \% 13 = 5$	$h(x_7) = h(216510306) = 216510306 \% 13 = 12$
$h(x_4) = h(134152056) = 134152056 \% 13 = 12$	$h(x_8) = h(197354865) = 197354865 \% 12 = 6$

Tabla 5.1

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	52/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Donde se puede observar que el estudiante con número de cuenta 132489973 se va a almacenar en la posición 5 de la tabla y ésta ya está ocupada por el estudiante con número de cuenta 197354864, lo mismo sucede con la posición 12. Lo anterior ejemplifica lo que es una colisión.

Método de multiplicación

Este método opera en dos partes, primero multiplicamos la llave x por una constante A en un rango $0 < A < 1$ (xA) y se extrae la parte fraccional que se multiplica por m (m es una potencia de 2) es decir se calcula $m(xA \bmod 1)$. Por último, se obtiene el mayor número entero menor o igual al resultado obtenido.

$$h(x) = \lfloor m(xA \bmod 1) \rfloor$$

Hashing Universal

A veces para una sola función hash puede haber un conjunto de llaves que produzcan todas las mismas salidas o le sea asignado la misma dirección en la tabla. Una forma de minimizar esto es, en lugar de usar una sola función hash ya definida, se puede seleccionar una función *hash* de forma aleatoria de una familia de funciones H . A esto se le denomina *hashing universal*.

Definición: Sea U el universo de llaves, y sea H un conjunto finito de funciones hash que mapean U a $\{0,1,2,3,\dots,m-1\}$. Entonces el conjunto H es llamado universal si para toda x, y que pertenecen a U donde $x \neq y$

$$|\{h \in H: h(x) = h(y)\}| = \frac{|H|}{m}$$

En otras palabras, la probabilidad de una colisión para dos llaves diferentes x e y dada una función hash aleatoria seleccionada del conjunto H es $\frac{1}{m}$.

Construcción de un conjunto universal de funciones Hash.

Para formar un conjunto H de funciones hash se llevan a cabo los siguientes pasos:

Paso 1

Seleccionar el tamaño de la tabla m tal que sea primo.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	53/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Paso 2

Descomponer la llave x en $r + 1$ elementos esto es $x = \langle x_0, x_1, x_2, \dots, x_r \rangle$, donde $x_i \in \{0, 1, 2, \dots, m - 1\}$
Equivalente a escribir x en base m .

Paso 3

Sea $a = \langle a_0, a_1, a_2, \dots, a_r \rangle$ una secuencia de $r + 1$ elementos seleccionados aleatoriamente tal que $a_i \in \{0, 1, 2, \dots, m - 1\}$. Hay m^{r+1} posibles secuencias.

Paso 4

Definir a la función hash $h_a = \sum_{i=0}^r a_i x_i \text{ mod } m$.

Paso 5

El conjunto H de funciones hash es:

$$H = \bigcup_a h_a$$

Con m^{r+1} miembros, uno para cada posible secuencia a .

Ejemplo: Se tiene que el universo de claves es el conjunto de direcciones IP, y cada dirección IP es una 4-tupla de 32 bits $\langle x_1, x_2, x_3, x_4 \rangle$, donde $x_i \in \{0, 1, 2, \dots, 255\}$

Sea m un número primo, por ejemplo $m=997$ si se desean almacenar 500 IPs

Se define h_a para 4-tupla $a = \langle a_0, a_1, a_2, a_4 \rangle$ donde $a_i \in \{0, 1, 2, \dots, m - 1\}$ como:

$$h_a: \text{DireccionIP} \rightarrow \text{Posicion Tabla}$$

$$h_a(x_0, x_1, x_2, x_4) = (a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4) \text{ mod } m$$

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	54/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Resolución de Colisiones

Las técnicas de resolución de colisiones se clasifican en dos categorías

- 1) Direccionamiento abierto (*Open Addressing*) o también llamado *closed hashing*.
- 2) Enlazamiento (*Chainhing*) o también llamado *open hashing*.

Direccionamiento abierto

Cuando el número n de elementos en la tabla se puede estimar con anticipación, existen diferentes métodos para almacenar n registros en una tabla de tamaño m donde $m > n$, los cuales utilizan las entradas vacías de la tabla para resolver colisiones, esos métodos se denominan métodos de direccionamiento abierto.

Aquí cuando una llave x se direcciona a una entrada de la tabla que ya está ocupada, se elige una secuencia de localizaciones alternativas $h_1(x), h_2(x) \dots$ dentro de la tabla. Si ninguna de las $h_1(x), h_2(x) \dots$ posiciones se encuentra vacía, entonces la tabla está llena y x no se puede insertar.

Existen diferentes propuestas para elegir las localizaciones alternativas. Las más sencillas se denominan *hashing* lineal, en el que la posición $h_j(x)$ de la tabla viene dada por:

$$h_j = (h(x) + j) \bmod m \quad \text{para } 1 \leq j \leq m - 1$$

$m = \text{tamaño de la tabla}$

Ejemplo

Retomando el ejemplo de los ocho estudiantes donde en el cálculo de las direcciones en la tabla se obtuvieron colisiones Tabla 5.2:

$h(197354864) = 5 = h(132489973)$	$h(134152056) = 12 = h(216510306)$	$h(106500306) = 3$
$h(933185952) = 933185952 \% 13 = 10$	$h(216500306) = 9$	$h(197354865) = 6$

Tabla 5.2

Utilizando el *hashing* o prueba lineal se tiene: Tabla 5.3.

Número de Cuenta (NC)	$h(\text{NC})$	$(h(\text{NC})+1)\%13$	$(h(\text{NC})+2)\%13$
197354864	5		
933185952	10		
132489973	5	6	
134152056	12		
216500306	9		
106500306	3		
216500306	12	0	
197354865	6	7	

Tabla 5.3

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	55/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Operaciones de Búsqueda e Inserción con manejo de colisiones utilizando hashing lineal

Cuando se busca algún dato o registro en una tabla hash por lo general está ya contiene información, por lo que primero se muestra un algoritmo en pseudocódigo que permite realizar la inserción de un par $(x, valor)$ en una tabla T de tamaño m , donde x es la llave y $valor$ el dato o registro correspondiente a la llave x . Tabla 5.4

Índice	(Llave,valor)
0	$(x_4, valor)$
1	
2	
3	
4	
$m-1$	

Tabla 5.4

En el algoritmo, si la dirección en la tabla ya está ocupada, se vuelve a calcular otra utilizando hashing lineal.

```

Insertar(T, m, x, valor)
Inicio
  j=0
  h=h(x)
  Mientras j < m
    indice=(h+j)mod m
    par=(x,valor)
    Si T[indice] está vacía
      T[indice]=par
      retornar indice
    Si no
      j=j+1
  Fin sino
Fin Mientras
retornar -1
Fin

```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	56/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Para el algoritmo correspondiente a la búsqueda de un dato dada una llave x , si en la dirección o índice calculado de la tabla no se encuentra el valor buscado entonces se considera una colisión y se vuelve a calcular el índice utilizando hashing lineal hasta que exista coincidencia o se determine que no existe esa llave en la tabla.

Buscar(T, x, valor, m)

Inicio

$j=0$

$h=h(x)$

Mientras $j < m$

$\text{indice}=(h+j)\text{mod } m$

 Si $T[\text{indice}]$ no está vacía

 Si $T[\text{indice}][0]==x$

 retornar $T[\text{indice}][1]$

 Si no

$j=j+1$

 Fin si no

 Fin Si

 No existe esa llave

 retornar -1

Fin Mientras

retornar -1

Fin

Enlazamiento

Consiste en colocar los elementos mapeados a la misma dirección de la tabla hash en una lista ligada. Figura 5.3 La posición o dirección j contiene un apuntador al inicio de la una lista ligada que contiene todos los elementos que son mapeados a la posición j . Si no hay elementos entonces la localidad tendrá un NULL.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	57/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

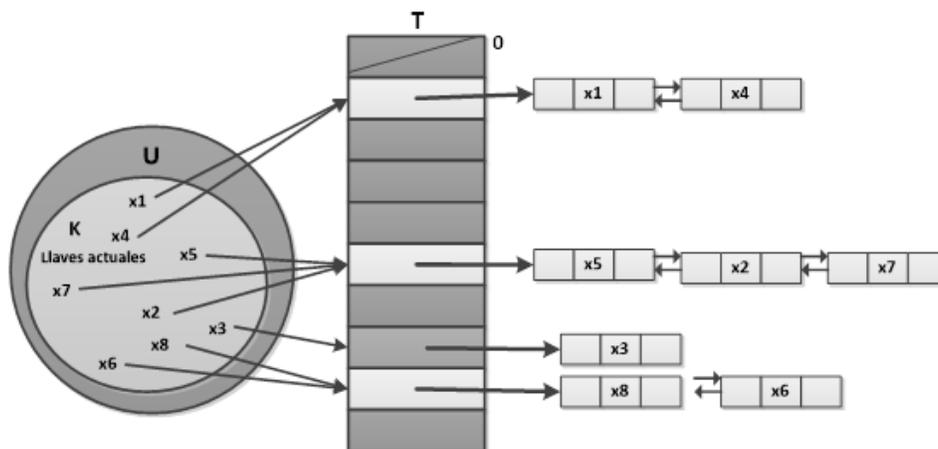


Figura 5.3 [1]

La búsqueda en la tabla cuando las colisiones se trabajan con enlazamiento se puede escribir como sigue:

Busqueda_Enlazamiento(T,k)

Inicio

Busca para un elemento con llave k en la lista T[h(k)]

Fin

Desarrollo

Actividad 1

Ir realizando un programa en Python donde dada una llave formada por una cadena de caracteres (letras y dígitos), se pueda insertar y buscar el valor o dato correspondiente a esa llave en la tabla hash. Para este desarrollo se tratarán las colisiones utilizando el direccionamiento abierto.

Aunque en Python ya se cuenta con una estructura de datos de diccionario, en esta guía se hará el mapeo a la tabla hash mediante listas para entender lo explicado antes.

Lo primero que se hará es crear la tabla de tamaño m utilizando una lista vacía de tamaño n , lo cual se puede hacer con la siguiente función

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	58/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```
def formarTabla(m):
    T=[None]*m
    return T
```

Ahora para representar la llave formada por una cadena de caracteres como un valor entero, se usa una función que suma el valor de cada carácter en ASCII y lo que retorna representará a la llave. La función en Python queda:

```
def convertirLlave(x):
    keyNum=0
    i=0
    for char in x:
        keyNum += ord(char)*i
        i+=1
    return keyNum
```

Como función hash se utiliza $h(x) = x \text{ mod } m$, la función en Python es:

```
def h(x,m):
    return x%m
```

Para agregar un elemento se tiene la siguiente función en Python que considera el manejo de colisiones utilizando el *hashing* lineal y se basa en el pseudocódigo mostrado anteriormente en esta guía. La función recibe información acerca tabla T, su tamaño m, la llave "x" y el valor a insertar.

```
def insertar(T,m,x,valor):
    j=0
    h1=h(convertirLlave(x),m)
    while (j<m):
        indice =(h1+j)%m
        par=[x,valor]
        if(T[indice]==None):
            T[indice] = par
            return indice
        else:
            j +=1
    print ("No hay lugar")
    return -1
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	59/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Una vez que se introduzcan elementos, será posible localizar datos en la tabla hash. La siguiente función en Python realiza una búsqueda, la cual recibe la misma información que la función insertar().

```
def buscar(T,m,x,valor):
    j=0
    h1=h(convertirLlave(x),m)
    while(j<m):
        indice = (h1+j)%m
        if(T[indice]!= None):
            if(T[indice][0]==x):
                return indice
            else:
                j+=1
        return -1
    return -1
```

Para visualizar qué pasa se define la función main() donde primero se forma la tabla de tamaño m=5, después se agregan 6 elementos (llave, valor) y por último se busca un elemento de los insertados.

```
def main():
    m= 5
    T=formarTabla(m)
    print(T)
    insertar(T,m,"Hola1","12213291")
    insertar(T,m,"Hola2","12213292")
    insertar(T,m,"Hola3","12213293")
    insertar(T,m,"Hola4","12213294")
    insertar(T,m,"Hola5","12213295")
    insertar(T,m,"Hola6","12213296")

    print(T)
    #retorna el indice de la tabla donde localizo el valor dada la llave
    print (buscar(T,m,"Hola5","12213295"))
main()
```

¿Qué sucede al agregar el sexto elemento? _____

¿Qué pasa si se busca un elemento que no esté en la tabla? _____

Actividad 2

Ejercicios propuestos por el profesor.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	60/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Referencias

[1] CORMEN, Thomas, LEISERSON, Charles, et al.

Introduction to Algorithms

3rd edition

MA, USA

The MIT Press, 2009

[2] D.S.Malik

Data Structure Using C++

Course Technology, Cengage Learning

Second Edition

[3] Ziviani, Nivio

Diseño de algoritmos con implementaciones en Pascal y C

Ediciones paraninfo /Thomson Learning

2007

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	61/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 6

Algoritmos de Grafos parte 1

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	62/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 6

Estructura de datos y Algoritmos II

Algoritmos de Grafos. Parte 1.

Objetivo: El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para entender el algoritmo de búsqueda por expansión.

Actividades

Implementar la búsqueda por expansión en un grafo representado por una lista de adyacencia en algún lenguaje de programación.

Antecedentes

- Análisis previo del concepto de grafo, su representación y algoritmo visto en clase teórica.
- Manejo de listas, diccionarios, estructuras de control, funciones y clases en Python 3.
- Conocimientos básicos de la programación orientada a objetos.

Introducción

Un grafo es una entidad matemática introducida por Leonhard Euler en 1736 con el trabajo de los 7 puentes de Königsberg, donde se formula el problema de cómo recorrer 7 puentes del centro de la ciudad de manera que se pasará solo una vez por ellos y se pudiera regresar al punto de partida.

Euler planteó el problema representando cada parte de tierra como un punto (llamado vértice) y cada puente como una línea (conocidas como aristas). Fig. 6.1.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	63/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

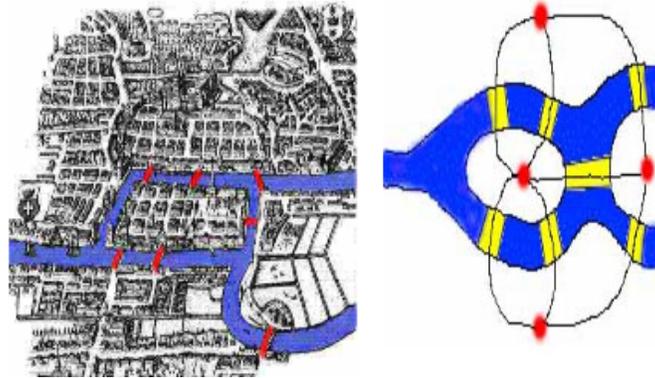


Figura 6.1.

Entonces un grafo permite representar entidades (vértices) y sus relaciones entre sí (aristas), por lo que permiten modelar problemas que se definen mediante las conexiones entre objetos o entidades; por ejemplo, en ingeniería para la representación de una red de cualquier tipo como de transporte (tren, carretera, avión), de servicios (eléctrica, gas, agua, comunicaciones), redes de internet, etc. Otros ejemplos pueden ser para representar estrategias de pase de un balón en un equipo de futbol, conexiones en circuitos lógicos entre otros.

Al modelar un problema mediante un grafo, las relaciones derivadas de las conexiones entre las entidades u objetos se pueden utilizar para responder a preguntas importantes para la solución de un problema como: ¿cuál es la menor distancia entre un objeto y otro?, ¿existe un camino para ir de un objeto a otro siguiendo las conexiones?, ¿cuántos objetos se pueden alcanzar a partir de uno determinado?

Definiciones

Grafo: Un grafo es un par ordenado $G = (V, A)$ donde V es el conjunto finito no vacío de elementos llamados vértices (o nodos) y A es el conjunto de pares no ordenados (i, j) donde i, j pertenecen a V , a este conjunto se le llama aristas(o arcos).

Un grafo $G = (V, A)$ contiene $|V|$ vértices y $|A|$ aristas, y el nombre dado a los vértices serán valores comprendidos entre 0 y $|V|$

Dependiendo de la importancia del orden de los vértices se tienen:

Grafos Dirigidos: Es un grafo donde A es un conjunto de aristas con una relación binaria en V , es decir el orden importa, la arista $(i, j) \neq (j, i)$. El que el vértice i esté conectado con el vértice j no implica que el vértice j esté conectado con el vértice i . En otras palabras, cuando las aristas tienen asignadas direcciones.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	64/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En la figura 6.2 se observan un grafo dirigido con un conjunto de vértices $V = \{0,1,2,3,4,5\}$ y un conjunto de aristas $A = \{(0,1), (0,3), (1,2), (1,3), (2,2), (2,3), (3,0), (5,4)\}$ [1]

En grafos dirigidos pueden existir aristas de un vértice a sí mismo, denominadas aristas cíclicas o self-loops.

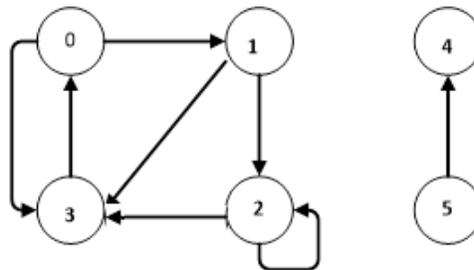


Figura 6.2.

Grafo no dirigido: Es un grafo donde el conjunto de las aristas es un conjunto de pares no ordenados, la arista (i, j) y (j, i) se consideran una única y misma arista. En un grafo no dirigido no se permiten las aristas cíclicas.

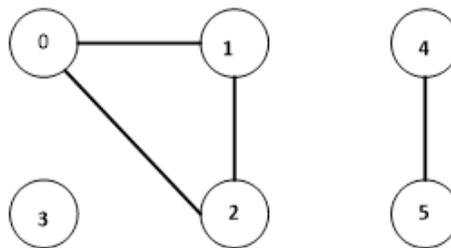


Figura 6.3

En la figura 6.3 se muestra un grafo no dirigido sobre un conjunto de vértices $V = \{0,1,2,3,4,5\}$ y el conjunto de aristas $A = \{(0,1), (0,2), (1,2), (4,5)\}$

Subgrafo: Un subgrafo de $G = (V, A)$ es un grafo $G' = (V', A')$ tal que $V' \subseteq V$ y $A' \subseteq A$

Orden: Es el número de vértices del grafo, el cardinal del conjunto V de vértices: $|V|$

Aristas incidentes a un vértice

En un grafo dirigido se dice que una arista a es **incidente** en un vértice v hacia el **exterior**, si v es el extremo inicial de a y a no es un ciclo. En la figura 6.2 las aristas que salen del vértice 1 son $(1,2)$ y $(1,3)$.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	65/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

La arista a es **incidente** a un vértice v hacia el **interior** si v es el extremo final de a y a no es un ciclo. En la figura 6.2 la arista que inciden sobre el vértice 1 es $(0,1)$.

En un grafo no dirigido se dice que una arista a es incidente a un vértice v , si v es uno de los extremos de a , y a no es un ciclo.

Adyacencia: Dos vértices son adyacentes si son distintos y existe una arista que va de uno a otro. En la figura 6.2 y 6.3 el vértice 1 es adyacente al vértice 0 ya que la arista $(0,1)$ pertenece a los dos grafos. Pero en la figura 6.2 el vértice 0 no es adyacente al 1 ya que la arista $(1,0)$ no pertenece al grafo.

Dos aristas son adyacentes, si siendo distintas, comparten un extremo común.

Grado: El grado de un vértice es el número de aristas que inciden en él. Si todos los vértices tienen el mismo grado, se conoce como grafo regular.

En un grafo dirigido se distingue entre semigrado interior o entrante de un vértice y es el número de aristas que llegan a él (inciden hacia el interior) y semigrado exterior o saliente de un vértice y es el número de aristas que salen de él (inciden hacia el exterior).

Grafo etiquetado. Un grafo se dice que está etiquetado, si cada arista tiene asociada una etiqueta o valor de cierto tipo.

Grafo ponderado o con pesos: Es un grafo etiquetado con valores numéricos.

Camino: En un grafo dirigido un camino es una secuencia finita de aristas tal que el extremo final de cada arista coincide con el extremo inicial del siguiente. Al camino se le llama simple cuando no utiliza dos veces la misma arista, en caso contrario se llama camino compuesto.

Longitud del camino: Es el número de aristas del camino y está dado por el número de vértices menos uno.

Circuito o ciclo: Es un camino en el que el vértice final coincide con el inicial.

Representación de los grafos

Un grafo se puede representar mediante una matriz cuadrada $B = [b_{ij}]$ con $|V| \times |V|$ elementos y es llamada matriz de adyacencia. En esta matriz cada renglón y cada columna representa un vértice del grafo y la posición (i, j) representa una arista o su ausencia, el extremo inicial de la arista se representa con el renglón i y el extremo final con la columna j . Se puede colocar un 1 en la posición (i, j) si existe un enlace que va de i a j y un 0 en caso contrario.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	66/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

$$b_{ij} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{en otro caso} \end{cases}$$

En el caso de un grafo no dirigido la matriz es simétrica lo que no ocurre en grafos dirigidos y con esta característica se puede almacenar solo la mitad de la matriz y ahorrar memoria. En la figura 6.4 se muestra la representación de un grafo no dirigido como una matriz de adyacencia.

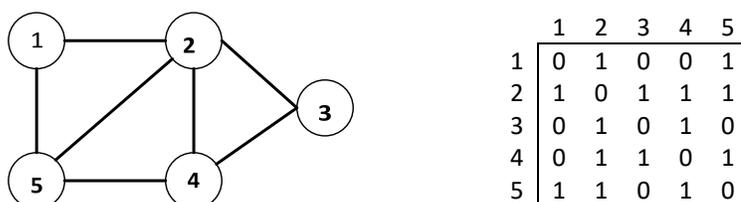


Figura 6.4 [1]

En la figura 6.5 se muestra un grafo dirigido representado en una matriz de adyacencia

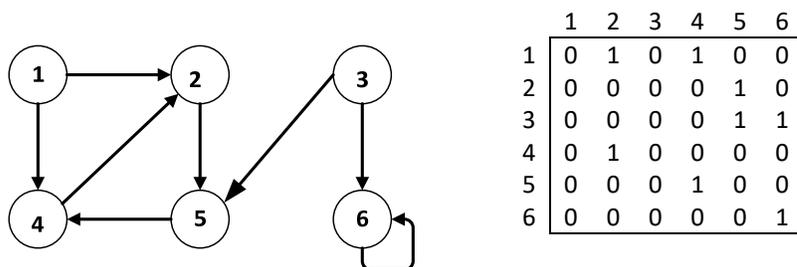


Figura 6.5[1]

Para grafos ponderados se puede colocar en lugar de 1, el valor del peso de la arista (i,j) en la misma posición y así hacer referencia a su existencia y un valor 0 o NULL para indicar la ausencia de la misma.

Es preferible usar una representación con Matriz de Adyacencia cuando el grafo es denso, es decir, el número de aristas $|A|$ es cercano al número de vértices al cuadrado $(|V|^2)$ (el grafo es pequeño) y cuando se quiere saber rápidamente si hay un arco o arista conectando dos vértices.

La otra forma de representar un grafo es mediante una lista de adyacencia, Figura 6.6. En este caso el Grafo $G = (V, A)$ consiste de un arreglo Adj que almacena $|V|$ listas, una para cada vértice o nodo en V . Para cada $u \in V$, la lista de adyacencia $Adj[u]$ contiene (la referencia a) todos los vértices v tal que hay una arista $(u, v) \in A$. Esto es $Adj[u]$ se conforma de todos vértices adyacentes a v en el grafo G [2].

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	67/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

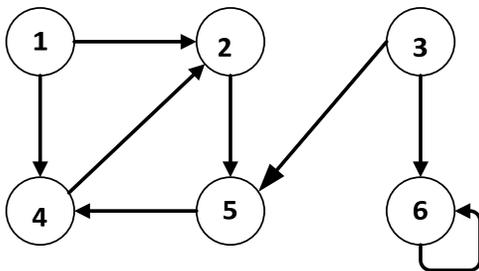
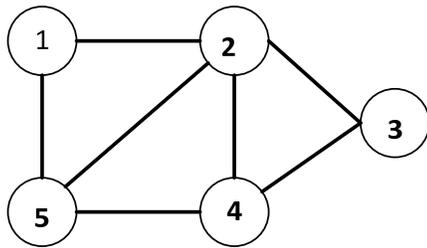


Figura 6.6 [1]

Si el grafo es dirigido, se cumple que la suma de los largos de las listas de adyacencia es $|A|$.

Si el grafo no es dirigido, se cumple que la suma de los largos de las listas de adyacencia es $2*|A|$. Dado que cada arco aparece dos veces.

Las listas de adyacencia pueden ser fácilmente adaptadas para representar grafos con peso. En estos un peso es asociado a cada arista a través de una función de peso $w: A \rightarrow R$. Así el peso de la arista (u, v) es puesto en el nodo v de la lista u .

Otra representación frecuente con grafos no orientados es la llamada matriz de incidencia Figura 6.7, que es una matriz $B = [b_{ij}]$ de $|V| \times |A|$ tal que:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	68/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

$$b_{i,j} = \begin{cases} -1 & \text{si la arista } j \text{ sale del vertice } i \\ 1 & \text{si la arista } j \text{ incide en el vertice } i \\ 0 & \text{En otro caso} \end{cases}$$

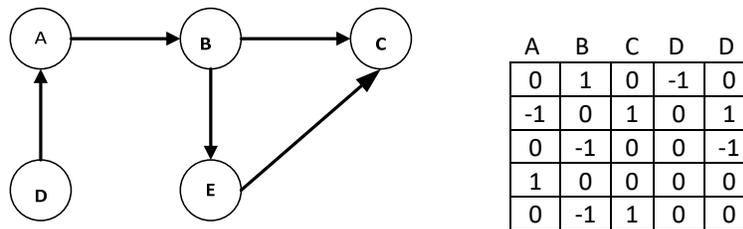


Figura 6.7

Algoritmos de exploración de grafos

Existen dos algoritmos importantes para la exploración de grafos que realizan un recorrido para visitar de forma eficiente cada vértice y arista, estos son:

- Búsqueda por expansión, también llamado búsqueda primero en amplitud o anchura
- Búsqueda por profundidad, también llamado búsqueda primero en profundidad.

Ambos algoritmos se ejecutan en un tiempo que crece linealmente al crecer el tamaño del grafo. [3]

Búsqueda primero en anchura (Breadth-first search- BFS)

Este algoritmo es uno de los más simples para explorar un grafo y es prototipo para otros algoritmos importantes como el algoritmo de Prim, que obtiene el árbol generador mínimo o árbol de recubrimiento mínimo y el algoritmo de Dijkstra que obtiene el camino más corto de un vértice a los otros vértices. [2]

Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto a través de la anchura de la frontera, es decir, llega a los nodos de distancia k , sólo luego de haber llegado a todos los nodos a distancia $k-1$.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	69/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

En el algoritmo BFS, dado un grafo $G = (V, A)$ y un vértice s , se exploran sistemáticamente las aristas de G para descubrir cada vértice que es alcanzable desde s , además calcula la menor distancia (menor número de arcos) de s a cada vértice alcanzable.

Una característica del algoritmo es que para cualquier vértice v alcanzable desde s , en la búsqueda, se va formando un árbol con raíz en s que contiene la ruta más corta y simple de s a v en G .

Para visualizar el progreso del algoritmo se colorea cada vértice de blanco, gris o negro. Todos los vértices inician en blanco y posteriormente se pueden colorear en gris y finalmente en negro, considerando lo siguiente [1]:

- Cuando se descubre un vértice por primera vez en el proceso de búsqueda, se colorea de gris.
- Si un vértice es gris o negro significa que ya ha sido descubierto, pero hay una distinción entre ellos para asegurar que la búsqueda se realice realmente en anchura.
- Si $(u, v) \in A$ y u es negro, entonces el vértice v es gris o negro, lo que significa que todos los vértices adyacentes a los vértices negros ya han sido descubiertos.
- Los vértices grises pueden tener algunos vértices adyacentes blancos que representan la frontera entre los vértices descubiertos y no descubiertos.

A continuación, se muestra el algoritmo en pseudocódigo de una función para realizar la búsqueda primero en anchura (BFS) en un grafo $G = (V, A)$ y partiendo de un vértice s [1]. Es importante mencionar que en el algoritmo que se describe se considera lo siguiente:

- El grafo está representado por una lista de adyacencia.
- A cada vértice se le agregan los atributos de color, distancia y predecesor, donde; el atributo color de un vértice u ($u.color$) puede ser blanco, gris o negro; el atributo distancia ($u.d$) mantiene la distancia de s al vértice u y el atributo predecesor ($u.p$) contiene información del predecesor del vértice u . Si no tiene se le asigna NULL.
- Se utiliza una cola Q (First in- First out), para la gestión de los vértices grises, o los que han sido descubiertos.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	70/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

BFS(G,s)

Inicio

```

Para cada vértice  $u \in V - \{s\}$ 
    u.color=White
    u.d =  $\infty$ 
    u.p=NULL
Fin Para
s.color=Gris
s.d=0
s.p=NULL
 $Q = \emptyset$ 
Encolar( $Q,s$ )
Mientras  $Q \neq \emptyset$ 
    u=Desencolar( $Q$ )
    Para cada vértice  $v$  que es adyacente al vértice  $u$ 
        Si v.color== Blanco
            v.color=Gris
            v.d=u.d+1
            v.p=u
            Encolar ( $Q,v$ )
        Fin Si
    Fin Para
    u.color=Negro
Fin Mientras

```

Fin

En la función BFS() primero se inician todos los vértices, a excepción de s , en blanco, se les coloca el atributo distancia en $u.d = \infty$ y el predecesor $u.p = NULL$.

Después se dan valores iniciales al vértice s para después encolarlo en la cola Q .

El ciclo *Mientras* funciona si se tienen vértices grises en la cola, los cuales forman el conjunto de vértices descubiertos cuyos vértices adyacentes (listas de adyacentes) todavía no se han revisado por completo. Antes de iniciar el ciclo *Mientras*, el único vértice que se encuentra en la cola es el vértice origen s .

La primera instrucción del ciclo *Mientras* es sacar de la cola al vértice u que se encuentre al inicio, para después con la estructura de repetición *Para* revisar todos los vértices v adyacentes a u . Si algún vértice v de los revisados, está coloreados en blanco, significa que apenas se descubrió y el algoritmo lo descubre colocando en sus atributos $v.color = Gris$, $v.d = u.d + 1$, $v.p = u$, una vez que se descubre el vértice v , se encola en Q ,

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	71/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

donde están los vértices descubiertos. Cuando ya se han revisado todos los vértices v adyacentes a u , u se coloca en negro.

El algoritmo trabaja con grafos dirigidos y no dirigidos.

Como ejemplo se va a realizar el recorrido en el siguiente grafo, Figura 6.8. Donde el vértice origen es s , se asignan valores iniciales a los demás y en la cola se agrega a s .

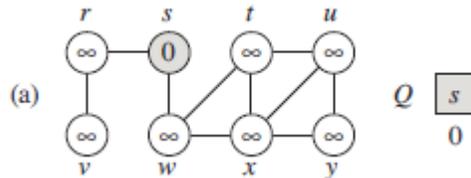


Figura 6.8[1]

Se desencola s para revisar sus vértices adyacentes (w y r), que apenas se descubrieron y están en blanco. Al descubrirse se colocan en gris y se agregan a la cola.

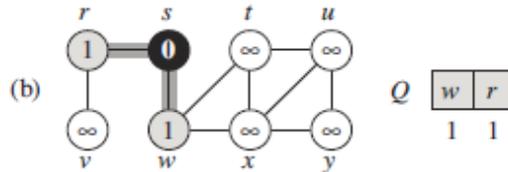


Figura 6.9[1]

Se desencola w y se revisan sus adyacentes, que son s, t, x , y los que están en blanco (t y x) se descubren colocándose en gris, se les aumenta el atributo distancia y se le añaden a la cola.

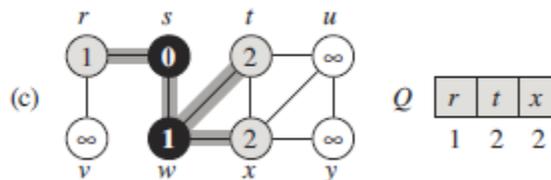


Figura 6.10[1]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	72/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ahora se desencola r y se revisan sus adyacentes que son s y v pero solo v se descubre, se colorea de gris, se le modifica la distancia, y se agrega a la cola.

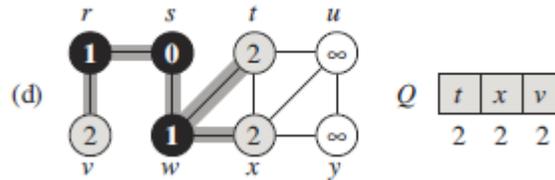


Figura 6.11[1]

El procedimiento continúa como se muestra en las siguientes secuencias, Figura 6.12. Hasta que todos los vértices ya están en negro y se obtiene la distancia d del vértice s a cada vértice alcanzable. Notar que la distancia d se observa dentro del nodo o vértice.

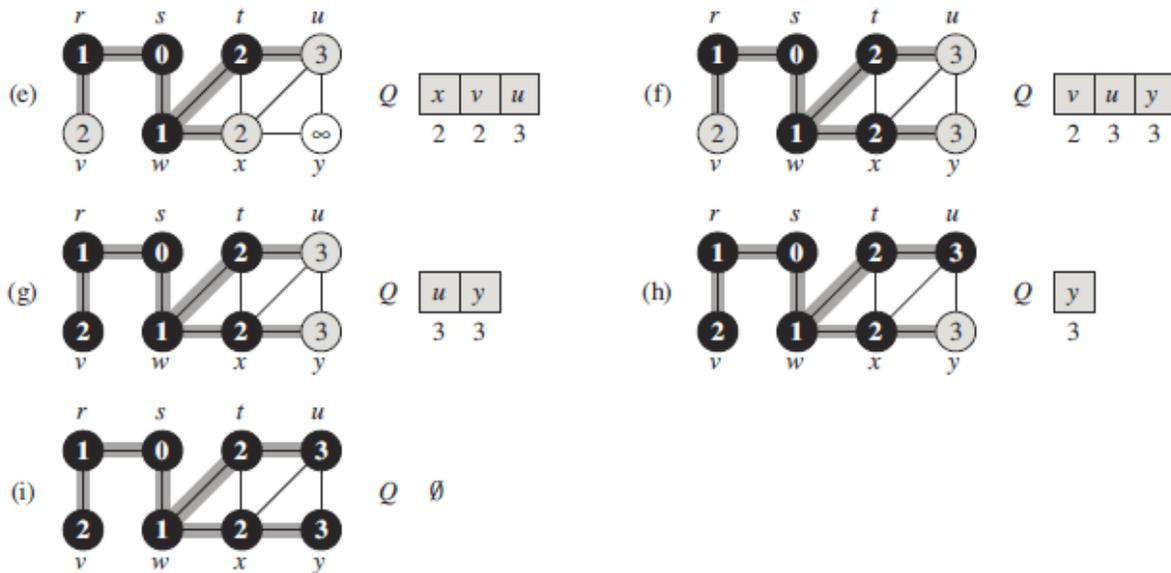


Figura 6.12[1]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	73/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Desarrollo

En esta práctica se involucra el uso de la programación orientada a objetos en Python, se introduce el uso de clases con sus atributos y métodos; la creación de un objeto, y su inicialización con el uso del método constructor al instanciar el objeto.

Antes de explicar la implementación del algoritmo BSF en Python se dan las definiciones básicas para recordar conceptos que se han visto en la asignatura de POO y como se usa en Python.

Clases: Las clases son los modelos o plantillas sobre los cuales se construirán los objetos, están formadas por atributos y métodos. En Python, una clase se define con la instrucción *class* seguida de su nombre y dos puntos, por ejemplo:

```
class nomClase:
```

La clase más elemental, es una clase vacía

```
class Clase:
    pass
```

donde la declaración *pass* indica que no se ejecutará ningún código, pero si se pueden instanciar objetos de la clase, por ejemplo:

```
objeto1 = Clase() # Crea objeto1 de la clase Clase
objeto2 = Clase() # Crea objeto2 de la clase Clase
```

Atributos. Las propiedades o atributos, son las características propias del objeto y modifican su estado. Estas se representan a modo de variables, los dos tipos de atributos o de variables existentes son variables de clase y variables de instancia (objetos). Ejemplo:

```
class Llanta():
    tipo = "variable de clase"
    longitud = ""

class Automovil():
    color = ""
    tamaño = ""
    modelo = ""
    llantas = Llanta() # atributo compuesto por un objeto de la clase Llanta
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	74/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Métodos. Los métodos son *funciones*, y representan acciones propias que puede realizar el objeto:

```
class Automovil():
    color = "azul"           #atributos
    tamaño = "grande"
    modelo = "VW"
    llantas = Llanta()
    def frenar(self): #metodo      #métodos
        pass
```

Importante: Notar que el primer parámetro de un método, siempre debe ser *self*

Método Constructor

El método `__init__` es el encargado de fungir como método constructor, es decir que este va a inicializar una serie de atributos y ejecutar el código que le definamos al momento de que se cree un objeto de la clase.

```
class Clase:
    def __init__(self):
        self.variable=42
```

Al crear el objeto *obj* de la forma, `obj = Clase()`, éste tiene su atributo *variable* inicializado con un valor de 42.

Si se quiere que los atributos se inicialicen de forma dinámica, se puede hacer lo siguiente:

```
class Clase:
    def __init__(self, valor=42):
        self.variable = valor
```

Actividad 1

Realizar un programa que represente un grafo mediante una lista de adyacencia y el paradigma orientado a objetos. A continuación, se describe brevemente un diseño e implementación en Python que ayudará al desarrollo del programa.

Considerar dos clases, la clase Vértice y la clase Grafo con sus atributos y métodos, como se muestra en el diagrama de clases de la figura 6.13.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	75/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

La clase Vértice, tiene como atributos un nombre y una lista de vértices adyacentes o vecinos y como método, agregar un vértice adyacente o vecino a su lista.

La clase grafo tiene como atributo un conjunto de vértices (representados con un diccionario) y como métodos, agregar vértices, agregar arista e imprimir grafo.

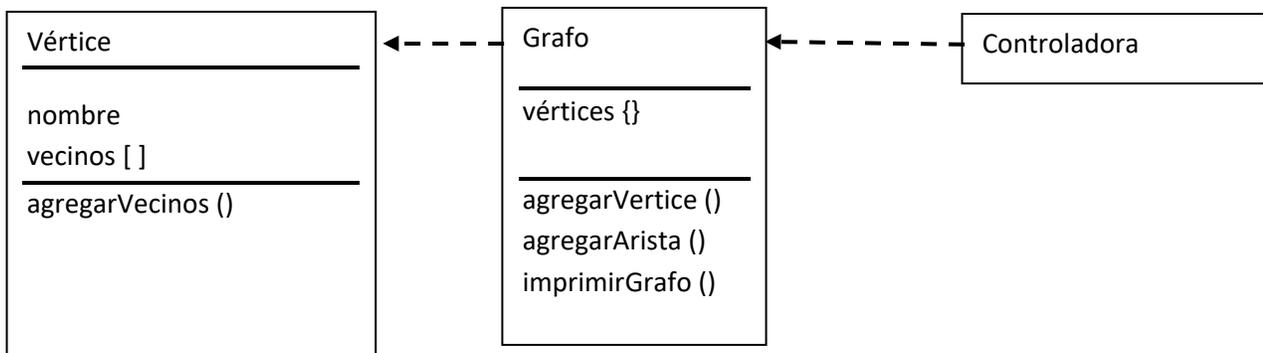


Figura 6.13

Abajo se muestra la implementación en Python de la clase Vértice y Grafo:

```

class Vertice:
    def __init__(self, n):
        self.nombre = n
        self.vecinos = list()

    def agregarVecino(self, v):
        if v not in self.vecinos:
            self.vecinos.append(v)
            self.vecinos.sort()
  
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	76/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```

class Grafo:
    vertices = {}

    def agregarVertice(self, vertice):
        if isinstance(vertice, Vertice) and vertice.nombre not in self.vertices:
            self.vertices[vertice.nombre] = vertice
            return True
        else:
            return False

    def agregarArista(self, u, v):
        if u in self.vertices and v in self.vertices:
            for key, value in self.vertices.items():
                if key == u:
                    value.agregarVecino(v)
                if key == v:
                    value.agregarVecino(u)
            return True
        else:
            return False

    def imprimeGrafo(self):
        for key in sorted(list(self.vertices.keys())):
            print("Vertice "+key +" Sus vecinos son"+ str(self.vertices[key].vecinos))

```

La siguiente clase, es la controladora, donde se coloca la secuencia de código que permite formar un grafo utilizando las clase *Vértice* y *Grafo* y cuyos vértices se representan con letras mayúsculas del alfabeto. Primero se crea un objeto 'g' de la clase *Grafo* y después se crean vértices que se van agregando al grafo.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	77/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

#Clase controladora
class Controladora:
    def main(self):
        #Se crea un objeto 'g' de la clase Grafo, el grafo
        g = Grafo()
        #Se crea un objeto 'a' de la clase Vertice, un vertice
        a = Vertice('A')
        # se agrega el vertice a al grafo
        g.agregarVertice(a)

        # Esta estructura de repetición es para agragar
        # todos los vertices, y no hacerlo uno a uno
        for i in range(ord('A'), ord('K')):
            g.agregarVertice(Vertice(chr(i)))

        # Se declara una lista que contiene las aristas del grafo
        edges = ['AB', 'AE', 'BF', 'CG', 'DE', 'DH', 'EH', 'FG', 'FI', 'FJ', 'GJ']

        # Se agregan las aristas al grafo
        for edge in edges:
            g.agregarArista(edge[:1], edge[1:])
        # Se imprime el grafo, como lista de adyacencia
        g.imprimeGrafo()

```

Para la ejecución del programa, se crea un objeto de la clase controladora y después se llama a la función main():

```

obj = Controladora()
obj.main()

```

Una vez terminado el programa probarlo con tres grafos no dirigidos propuestos.

Actividad 2

Realizar las modificaciones que se describen abajo a las clases *Vértice* y *Grafo* para que en el programa realizado se implemente el algoritmo en pseudocódigo de BFS explicado anteriormente.

Lo primero es agregar los atributos de color, distancia y predecesor a la clase *Vértice* como sigue:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	78/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```

class Vertice:
    def __init__(self, n):
        self.nombre = n
        self.vecinos = list()
        self.distancia = 9999
        self.color = 'white'
        self.pred = -1

    def agregarVecino(self, v):
        if v not in self.vecinos:
            self.vecinos.append(v)
            self.vecinos.sort()

```

Ahora adicionar el método *bsf()* a la clase Grafo. El método realiza la **Búsqueda primero en anchura** y es el siguiente:

```

def bsf(self, vert):
    vert.distancia = 0
    vert.color = 'gris'
    vert.pred=-1
    q=list()

    q.append(vert.nombre)

    while len(q) > 0:

        u = q.pop()
        node_u = self.vertices[u]
        for v in node_u.vecinos:
            node_v = self.vertices[v]
            if node_v.color == 'white':
                node_v.color='gris'
                node_v.distancia=node_u.distancia + 1
                node_v.pred=node_u.nombre
                q.append(v)
        self.vertices[u].color='black'

```

Además también se modifica el método *imprimeGrafo()* para mostrar las distancias obtenidas con la búsqueda del vértice *A* a cualquier otro vértice.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	79/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```
def imprimeGrafo(self):
    for key in sorted(list(self.vertices.keys())):
        print("Vertice "+key+" Sus vecinos son "+ str(self.vertices[key].vecinos))
        print("La Distancia de A a " + key + " es: "+ str(self.vertices[key].distancia))
```

Una vez terminadas las modificaciones, probar el programa con diferentes grafos propuestos y verificar resultados

Actividad 3

Ejercicios propuestos por el profesor.

Referencias

[1] CORMEN, Thomas, LEISERSON, Charles, et al.

Introduction to Algorithms

3rd edition

MA, USA

The MIT Press, 2009

[2] Ziviani, Nivio

Diseño de algoritmos con implementaciones en Pascal y C

Ediciones paraninfo /Thomson Learning

2007

[3] Baase-Val Gelder

Algoritmos computacionales. Introducción al análisis y diseño

Tercera edición

Addison Wesley

[4]Hernández Figueroa, Rodríguez, del Pino, González Domínguez, Díaz Roca, Pérez Aguilar, Rodríguez Rodríguez

Fundamentos de estructuras de Datos, Soluciones en ADA, JAVA y C++

[5] http://librosweb.es/libro/python/capitulo_5/programacion_orientada_a_objetos.html

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	80/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 7

Algoritmos de Grafos parte 2

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	81/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 7

Estructura de datos y Algoritmos II

Algoritmos de Grafos. Parte 2.

Objetivo: El estudiante conocerá e identificará las características necesarias para entender el algoritmo de búsqueda por profundidad en un grafo.

Actividades

Implementar la búsqueda por profundidad en un grafo representado por una lista de adyacencia en algún lenguaje de programación.

Antecedentes

- Análisis previo del concepto de grafo, su representación y algoritmo visto en clase teórica.
- Desarrollo de la Guía práctica de estudio 6.
- Manejo de listas, diccionarios, estructuras de control, funciones y clases en Python 3.
- Conocimientos básicos de la programación orientada a objetos.

Introducción

Recorrer un grafo consiste en “visitar” cada uno de los vértices a través de las aristas del mismo.

La búsqueda en profundidad (del inglés Depth-First Search DFS) es un algoritmo para recorrer un grafo $G = (V, A)$ visitando primero los vértices más profundos en G siempre que sea posible.

La estrategia de recorrido en profundidad (DFS), explora sistemáticamente las aristas de G , de manera que primero se visitan los vértices adyacentes a los visitados más recientemente. Se parte de un vértice v y el recorrido comienza por alguno de sus vértices adyacentes, luego un adyacente de este, y así sucesivamente hasta llegar a un vértice que ya no tiene vértices adyacentes que visitar. Luego la búsqueda vuelve atrás (*backtrack*) para seguir otro camino por algún vértice adyacente del último vértice visitado que aún tiene vértices adyacentes sin visitar. Este proceso continúa hasta que todos los vértices alcanzables desde el vértice v de la llamada original han sido descubiertos.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	82/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

A igual que el algoritmo BFS visto en la guía 6, en el recorrido, cada vértice se colorea de blanco, gris o negro. Primero todos se inicializan en blanco, cuando el vértice se descubre por primera vez se colorea en gris y cuando toda su lista de adyacentes se ha examinado por completo se colorea en negro.

Durante el progreso del algoritmo también sucede lo siguiente:

- Cuando se descubre un vértice u durante la lectura de la lista de adyacencia de un vértice v ya descubierto, el algoritmo registra este evento asignando v al antecesor de u ($v.pred = u$).
- También se registran dos tiempos, el primero $u.d$ que es el instante o momento en que el vértice u se descubre (y se colorea de gris), y el tiempo en $u.f$, el instante de tiempo en el que se termina de examinar la lista de adyacentes a u (y se colorea de negro). Estos dos tiempos proveen información importante acerca de la estructura del grafo y ayudan a la inferencia del desarrollo del mismo.

A continuación, se presenta un algoritmo para DFS en pseudocódigo [1], donde los tiempos mencionados son enteros entre 1 y $2|V|$ y para cada vértice u , $u.d < u.f$.

DFS ($G = (V, E)$)

Inicio

Para cada vértice $u \in V$

Inicio

$u.color = \text{blanco}$

$u.pred = \text{ninguno}$

Fin Para

Para cada vértice $u \in V$

Inicio

Si $u.color == \text{blanco}$

DFS-VISITAR(G, u)

Fin Si

Fin Para

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	83/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

DFS-VISITAR(G, u)

Inicio

tiempo=tiempo +1

$u.d$ =tiempo

$u.color$ =gris

Para cada vértice v que pertenece a la lista de adyacencia de u

Inicio

 Si $v.color$ == blanco

$v.pred$ = u

 DFS-VISITAR(G, v)

 Fin Si

Fin Para

$u.color$ =negro

tiempo=tiempo+1

$u.f$ =tiempo

Fin

Analizando la función DFS (), primero se utiliza una estructura de repetición para colorear todos los vértices del grafo en blanco e inicializar su atributo del predecesor en “ninguno”. Una vez terminado se inicializa el tiempo en cero y con otra estructura de repetición se revisa cada vértice $u \in V$ en turno; cuando está en blanco realiza una visita en profundidad utilizando la función DFS-VISITAR () y al término de esta, al vértice u se le habrá asignado un tiempo de descubrimiento y el tiempo de término de examinar su lista de adyacencia.

Una característica importante que cabe señalar es que cada vez que se llama a la función DFS-VISITAR(G, u) el vértice u se convierte en raíz de un nuevo árbol de búsqueda en profundidad y el conjunto de árboles forma un bosque de árboles de búsqueda o subgrafo de predecesores G_{pred} , definido de la siguiente manera:

$$G_{pred} = (V, A_{pred}) \text{ donde } A_{pred} = \{(v.pred, v) : v \in V \text{ y } v.pred \neq \text{ninguno}\}$$

En cada llamada a la función DFS-VISITAR(G, u), para el vértice u visitado, se modifica el tiempo incrementándolo en uno, se registra en $u.d$ como tiempo de descubrimiento y se colorea u de gris. Después se examina con una estructura de repetición cada vértice v adyacente a u y si este está en blanco se le realiza una visita de forma recursiva, aquí se dice que cada arista (u, v) se exploró por búsqueda primero en profundidad.

Una vez terminada esta exploración se colorea u de negro se incrementa el tiempo y se registra en $u.f$ como de término de la exploración de u por DFS.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	84/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Una propiedad de este algoritmo es que se puede usar para la clasificación de aristas del grafo de entrada, información que es útil para implementar otros algoritmos como por ejemplo para verificar si un grafo es acíclico.

Se pueden definir cuatro tipos de aristas: **Aristas de árbol:** Son las aristas de un árbol de búsqueda en profundidad y por tanto del bosque *Gpred*. La arista (u, v) es una arista de árbol si v se alcanzó por primera vez al recorrer dicha arista [2].

Aristas de retorno: Son las aristas (u, v) que se conectan al vértice u con un antecesor v de un árbol de búsqueda en profundidad. Las aristas cíclicas se consideran aristas de retorno [2][1].

Aristas de avance: Son las aristas (u, v) que no pertenecen al árbol de búsqueda. Pero conectan un vértice u con un descendiente v que pertenece al árbol de búsqueda en profundidad [2].

Aristas de cruce: Son todas las otras aristas, que pueden conectar vértices en el mismo árbol de búsqueda en profundidad o en dos árboles de búsqueda en profundidad diferentes [2][1].

A continuación, se muestra la evolución de la búsqueda por profundidad para un grafo dirigido, figura 7.1-figura 7.8, considerar que dentro del vértice se coloca el tiempo de *descubrimiento/tiempo de finalización*. Las aristas sombreadas indican que forman parte de un árbol mientras si no lo son, se representan como líneas discontinuas y son etiquetadas con *B*, *C* o *F* para indicar si son aristas de retorno, de cruce o de avance respectivamente.

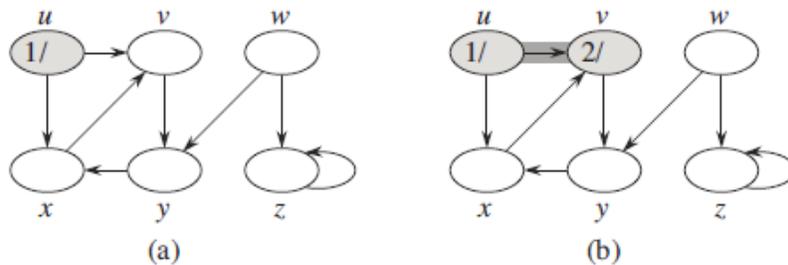


Figura 7.1 [1]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	85/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

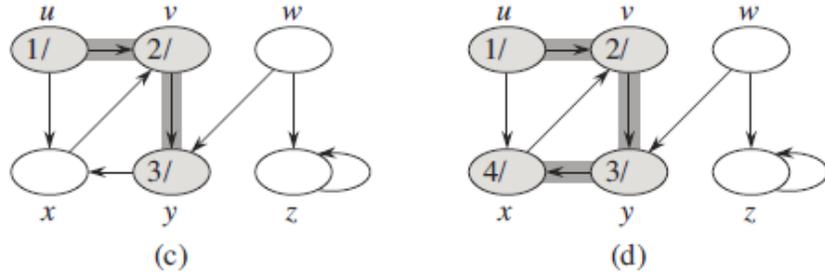


Figura 7.2 [1]

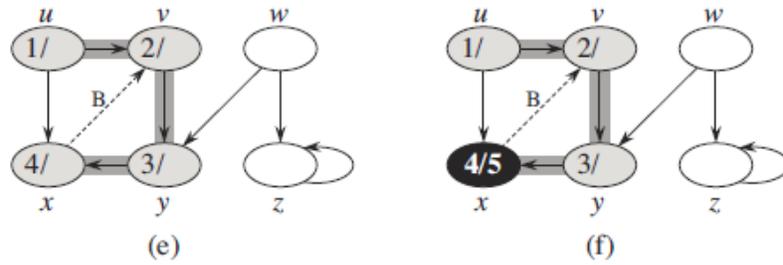


Figura 7.3 [1]

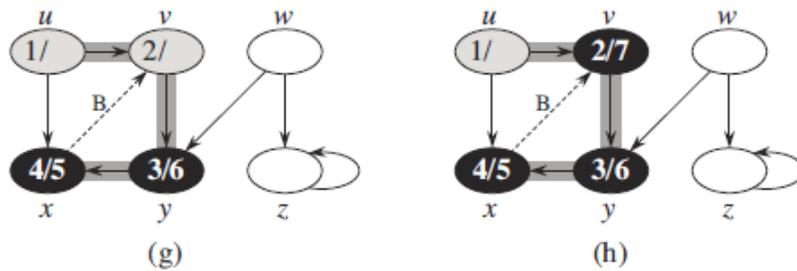


Figura 7.4 [1]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	86/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

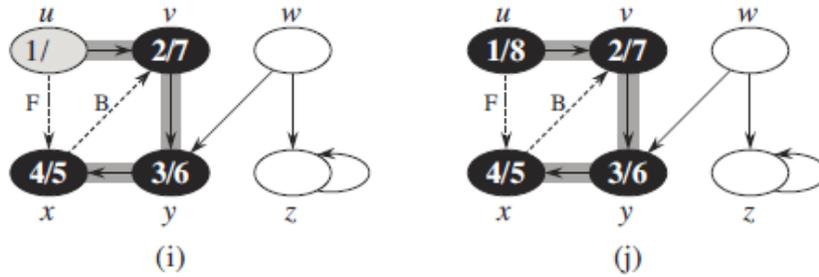


Figura 7.5 [1]

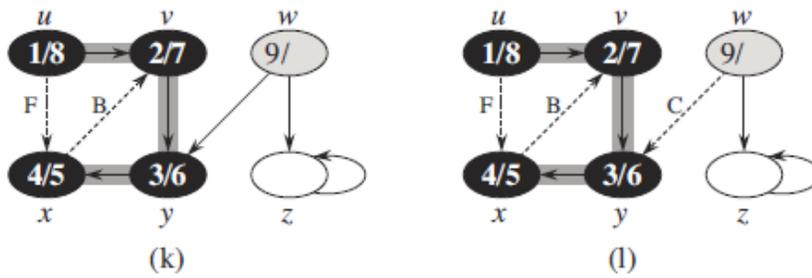


Figura 7.6 [1]

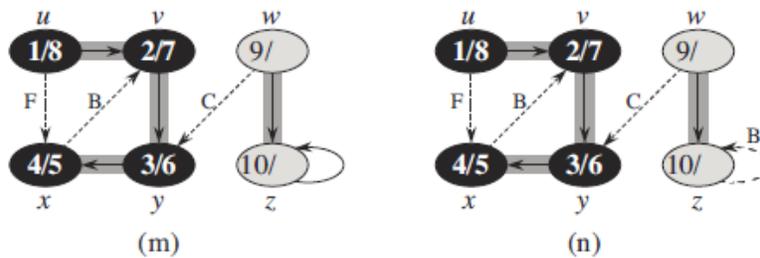


Figura 7.7 [1]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	87/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

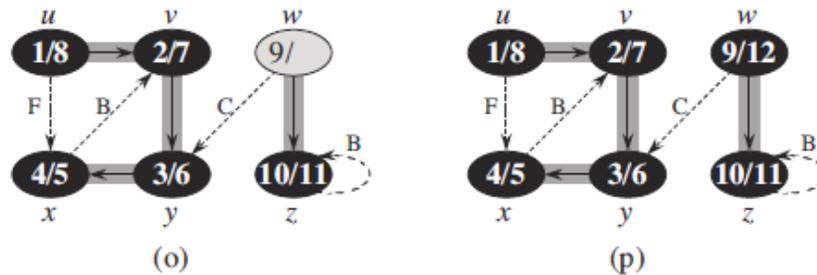


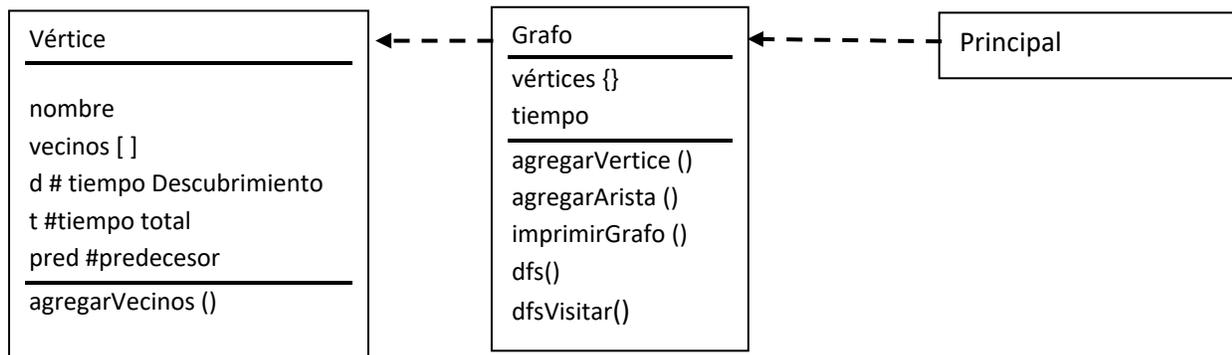
Figura 7.8 [1]

Desarrollo

Actividad 1

Realizar un programa donde se implemente el algoritmo DFS que se analizó en esta guía utilizando el pseudocódigo explicado del mismo, además probar que la salida coincide con los resultados del ejemplo mostrado en las figuras 7.1 - 7.8 .

Para ello se plantea el algoritmo en el siguiente diagrama de clases:



A continuación, se dan las implementaciones de las clases *Grafo* y *Vertice* en Python, y como parte de la actividad se debe realizar la controladora que contenga un método que dé la secuencia de solución para la búsqueda por profundidad del grafo dirigido de la figura 7.1.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	88/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

La clase *Vertice* es:

```
class Vertice:
    def __init__(self, n):
        self.nombre = n
        self.vecinos = list()

        self.d = 0 # tiempo de descubrimiento
        self.f=0 # tiempo de término
        self.color = 'white'
        self.pred = -1

    def agregarVecino(self, v):
        if v not in self.vecinos:
            self.vecinos.append(v)
            self.vecinos.sort()
```

Para la clase *Grafo* primero se muestran los atributos y después por separado cada uno de los métodos:

```
class Grafo:
    vertices = {}
    tiempo = 0
```

```
def agregarVertice(self, vertice):
    if isinstance(vertice, Vertice) and vertice.nombre not in self.vertices:
        self.vertices[vertice.nombre] = vertice
        return True
    else:
        return False
```

```
def agregarArista(self, u, v):
    if u in self.vertices and v in self.vertices:
        for key, value in self.vertices.items():
            if key == u:
                value.agregarVecino(v)
                #if key == v: #Se comenta porque es grafo dirigido
                #    value.agregarVecino(u)
        return True
    else:
        return False
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	89/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```
def imprimeGrafo(self):
    for key in sorted(list(self.vertices.keys())):
        print("Vertice: "+key)
        print("Descubierto/Termino:"+str(self.vertices[key].d)+"/"+str(self.vertices[key].f))
```

```
def dfs(self,vert):
    global tiempo
    tiempo=0
    for u in sorted(list(self.vertices.keys())):
        if self.vertices[u].color=='white':
            self.dfsVisitar(self.vertices[u])
```

```
def dfsVisitar(self,vert):
    global tiempo
    tiempo = tiempo + 1
    vert.d=tiempo
    vert.color='gris'

    for v in vert.vecinos:
        if self.vertices[v].color == 'white':
            self.vertices[v].pred=vert
            self.dfsVisitar(self.vertices[v])
    vert.color="black"
    tiempo=tiempo+1
    vert.f=tiempo
```

Actividad 2

Ejercicios sugeridos por el profesor

Referencias

[1] CORMEN, Thomas, LEISERSON, Charles, et al.

Introduction to Algorithms

3rd edition

MA, USA

The MIT Press, 2009

[2] Ziviani, Nivio

Diseño de algoritmos con implementaciones en Pascal y C

Ediciones paraninfo /Thomson Learning

2007

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	90/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 8

Árboles parte 1

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	91/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 8

Estructura de datos y Algoritmos II

Árboles. Parte 1.

Objetivo: El estudiante conocerá e identificará las características de la estructura no lineal árbol.

Actividades

Implementar una estructura de datos- árbol binario- así como su recorrido en algún lenguaje de programación.

Antecedentes

- Análisis previo del concepto de árbol y su representación visto en clase teórica.
- Manejo de listas, diccionarios, estructuras de control, funciones y clases en Python 3.
- Conocimientos básicos de la programación orientada a objetos.

Introducción

En las estructuras de datos lineales como las pilas o las colas, los datos se estructuran en forma secuencial es decir cada elemento puede ir enlazado al siguiente o al anterior. En las estructuras de datos no lineales o estructuras multi-enlazadas se pueden presentar relaciones más complejas entre los elementos; cada elemento puede ir enlazado a cualquier otro, es decir. puede tener varios sucesores y/o varios predecesores. Ejemplos de estructuras de datos no lineales son los grafos y árboles.

Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación(rama) que impone una estructura jerárquica entre los nodos. Los árboles genealógicos y los organigramas son ejemplos de árboles.

Un árbol puede definirse formalmente de forma recursiva como sigue:

- 1- Un solo nodo es, por sí mismo un árbol. Ese nodo es también la raíz de dicho árbol.
- 2- Suponer que n es un nodo y que A_1, A_2, \dots, A_k son árboles con raíces n_1, n_2, \dots, n_k , respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos n_1, n_2, \dots, n_k . En dicho árbol, n es la raíz y A_1, A_2, \dots, A_k son subárboles de la raíz. Los nodos n_1, n_2, \dots, n_k , reciben el nombre de hijos del nodo n .

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	92/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

La definición implica que cada nodo del árbol es raíz de algún subárbol contenido en el árbol principal.

Un árbol vacío o nulo es aquel que no tiene ningún nodo.

Ejemplo: El índice general de un libro Figura 8.1 a), es un árbol y se puede dibujar como en 8.1 b) donde la relación padre hijo se representa con una línea que los une. Los árboles normalmente se dibujan de arriba hacia abajo y de izquierda a derecha como en el ejemplo.

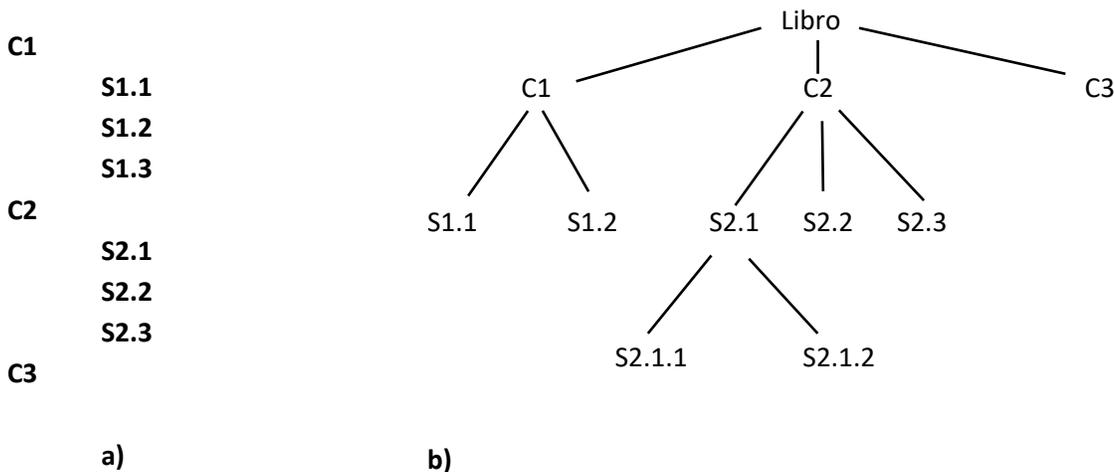


Figura 8.1

Se puede observar en la figura 8.2 b) que la raíz del árbol es libro y este tiene 3 subárboles C1, C2 y C3 que a su vez son hijos del nodo o padre libro.

Términos asociados con el concepto de árbol.

Grado de un nodo: Es el número de subárboles o hijos que tienen como raíz ese nodo. En la figura 8.2 b) la raíz del árbol es libro y este tiene 3 subárboles C1, C2 y C3, el grado de libro es tres.

Nodo terminal u hoja: Nodo con grado 0, no tiene subárboles, por ejemplo, el nodo C3.

Grado de un árbol: Grado máximo de los nodos de un árbol.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	93/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Hijos de un nodo: Nodos que dependen directamente de ese nodo, es decir, las raíces de sus subárboles. Si un nodo x es descendiente directo de un nodo y , se dice que x es hijo de y .

Padre de un nodo: Antecesor directo de un nodo, nodo del que depende directamente. Si un nodo x es antecesor directo de un nodo y , se dice que x es padre de y .

Nodos hermanos: Nodos hijos del mismo nodo padre.

Camino: Sucesión de nodos del árbol $n_1, n_2, n_3, \dots, n_k$, tal que n_i es el padre de n_{i+1} .

Antecesoros de un nodo: Todos los nodos en el camino desde la raíz del árbol hasta ese nodo.

Nivel de un nodo: Es el número de arcos que deben ser recorridos para llegar a un determinado nodo o la longitud del camino desde la raíz hasta el nodo.

Altura: La altura de un nodo en un árbol es la longitud del mayor de los caminos del nodo a cada hoja. La altura de un árbol es la altura de la raíz.

Longitud de camino de un árbol: Suma de las longitudes de los caminos a todos sus componentes.

Bosque: Conjunto de uno o más árboles.

Recorridos

El recorrido de una estructura de datos es el acceso sistemático a cada uno de sus miembros. Hay dos formas básicas de recorrer un árbol; en profundidad y en amplitud

Recorrido en profundidad: Siempre empieza accediendo a la raíz. Cuando es en profundidad se trata de alejarse en todo lo posible de la raíz hasta alcanzar un nodo hoja. Una vez alcanzado se da un paso atrás para intentar alejarse por un camino alternativo. Este esquema implica que por nodo se pasa varias veces: cuando se accede por primera vez y cuando se regresa de cada uno de sus hijos para acceder al siguiente o volver al padre. El tratamiento del nodo se puede hacer en cualquiera de esas ocasiones, lo que da lugar, según el momento que se elija a tres variantes: los recorridos en preorden, inorden(simétrico) y postorden[1]. El nombre dado a los recorridos es de acuerdo a la posición de la raíz.

Preorden: Se trata primero la raíz del árbol y , a continuación, se recorren en preorden sus subárboles de izquierda a derecha (raíz - subárbol izq. - subárbol der). Figura 8.2.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	94/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

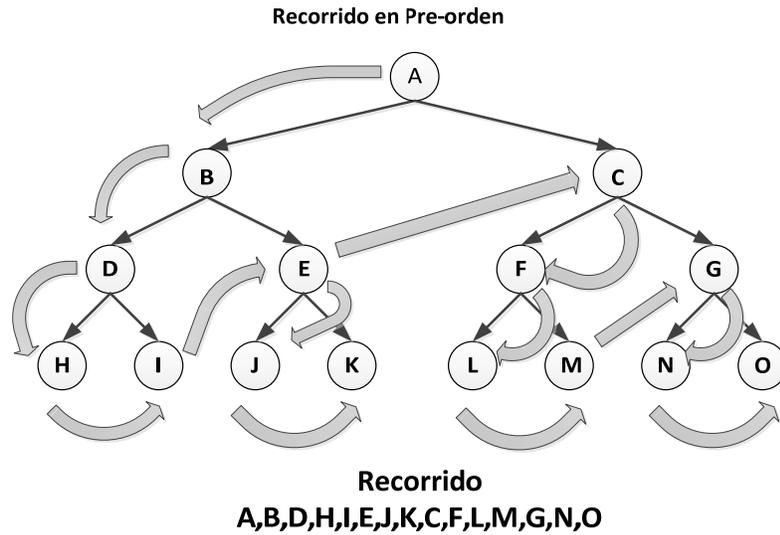


Figura 8.2

Inorden : Se recorre en inorden el primer subárbol , luego se trata de la raíz y, a continuación se recorre en inorden el resto de los subárboles (subárbol izq. - raíz - subárbol der). Figura 8.3.

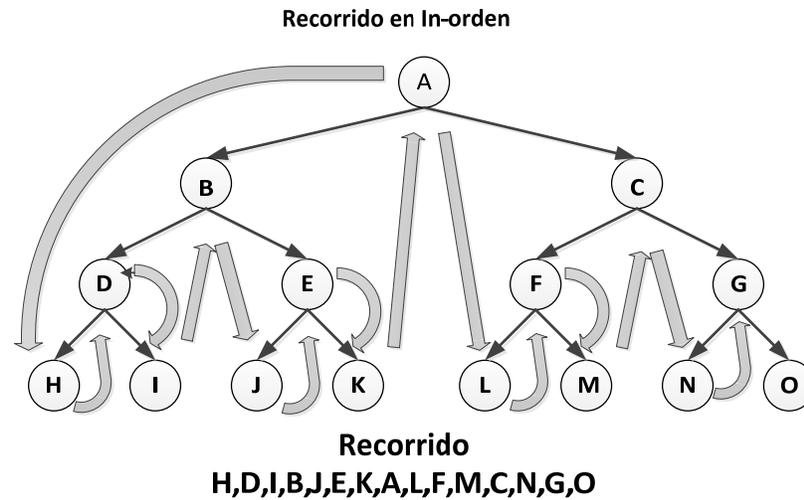
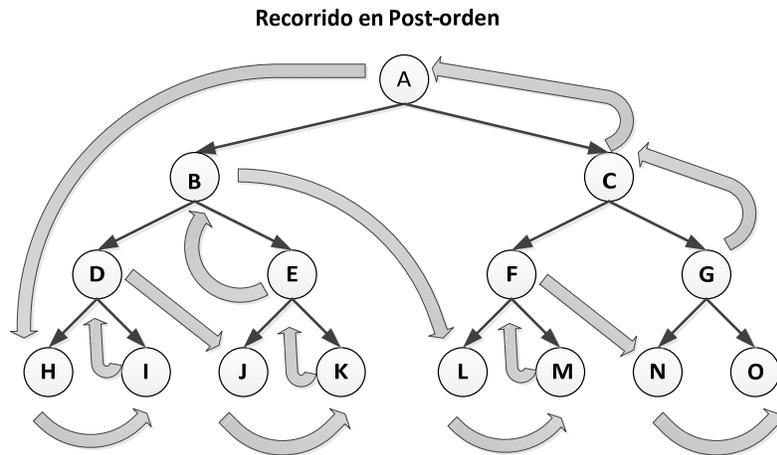


Figura 8.3

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	95/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

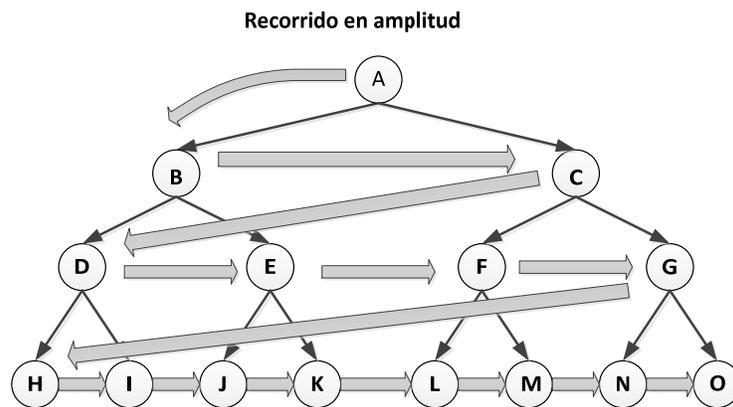
Postorden: Se recorren primero en postorden todos los subárboles, de izquierda a derecha, y finalmente se trata la raíz (subárbol izq. - subárbol der. -raíz). Figura 8.4.



Recorrido
H,I,D,J,K,E,B,L,M,F,N,O,G,C,A

Figura 8.4

Recorrido en anchura: También conocida como recorrido por niveles o en amplitud. Se explora el árbol nivel a nivel, de izquierda a derecha y del primero al último. Figura 8.5.



Recorrido
A,B,C,D,E,F,G,H,I,J,K,L,M,N,O

Figura 8.5

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	96/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Ejemplo. Para el árbol de la figura 8.6, sus recorridos son:

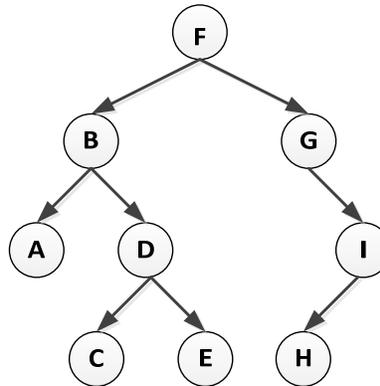


Figura 8.6

Recorrido en Profundidad:

Secuencia de recorrido de preorden: F, B, A, D, C, E, G, I, H (raíz, izquierda, derecha)

Secuencia de recorrido de inorden: A, B, C, D, E, F, G, H, I (izquierda, raíz, derecha).

Secuencia de recorrido de postorden: A, C, E, D, B, H, I, G, F (izquierda, derecha, raíz)

Recorrido en Anchura:

Secuencia de recorrido de orden por nivel: F, B, G, A, D, I, C, E, H

Árboles Binarios

Un árbol binario es un árbol de grado dos, esto es, cada nodo puede tener dos, uno o ningún hijo. En los árboles binarios se distingue entre el subárbol izquierdo y el subárbol derecho de cada nodo. De forma que, por ejemplo, los dos siguientes árboles (Figura 8.7), a pesar de contener la misma información son distintos por la disposición de los subárboles:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	97/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

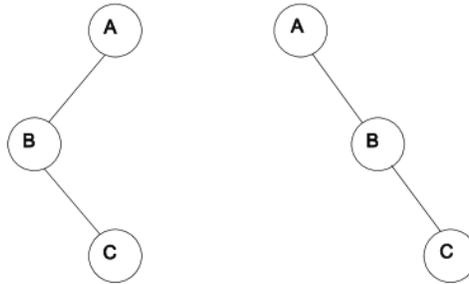


Figura 8.7

Se puede definir al árbol binario como un conjunto finito de m nodos ($m \geq 0$), tal que:

- 1- Si $m = 0$, el árbol está vacío
- 2- Si $m > 0$
 - a. Existe un nodo raíz
 - b. El resto de los nodos se reparte entre dos árboles binarios, que se conocen como subárbol izquierdo y subárbol derecho de la raíz.

A continuación, se describen algunas características de los árboles binarios.

Árbol binario lleno

Es un árbol binario lleno si cada nodo es de grado cero o dos. O bien, si es un árbol binario de profundidad k que tiene $2^k - 1$ nodos (es el número máximo de nodos).

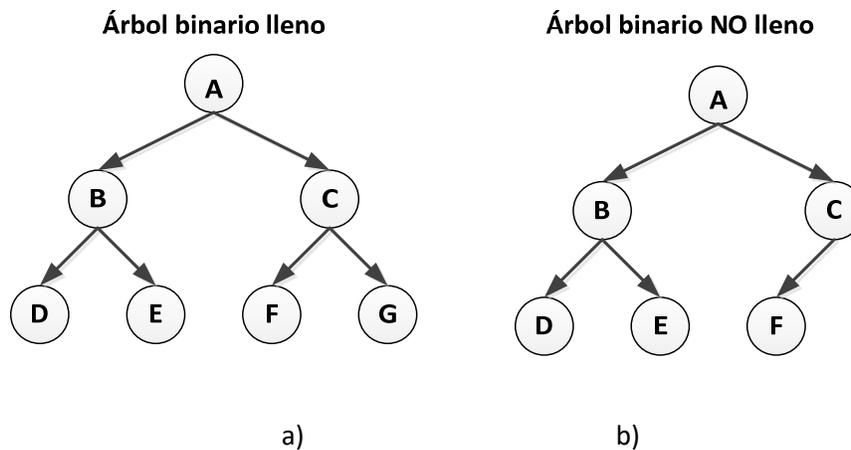


Figura 8.8[4]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	98/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Árbol binario completo

Un árbol binario es completo si todos los nodos de grado cero o uno están en los dos últimos niveles, de forma que las hojas del último nivel ocupan las posiciones más a la izquierda de dicho nivel. En un árbol binario completo de altura h , todos los niveles, excepto posiblemente el nivel h están completamente llenos.

Si un árbol está lleno también está completo.

Árboles equilibrados

Un árbol es equilibrado cuando la diferencia de altura entre los subárboles de cualquier nodo es máxima 1. Un árbol binario es equilibrado totalmente cuando los subárboles izquierdo y derecho de cada nodo tienen las mismas alturas, es decir, es un árbol lleno.

Se dice que un árbol completo es equilibrado y un árbol lleno es totalmente equilibrado.

Operaciones en árboles binarios: Un árbol binario se puede ver como un tipo de dato abstracto y algunas de las operaciones que se pueden realizar sobre él son:

- Crear y eliminar el árbol
- Verificar el estado del árbol, si está vacío o no.
- Crear y remover nodos
- Saber si el nodo es hoja o no
- Búsqueda por contenido
- Recorrer el árbol
- Obtener el padre, hijo izquierdo o hijo derecho, dado un nodo.

Una de las operaciones básicas es la creación del árbol, una vez que ya se tiene, se pueden realizar las operaciones sobre sus elementos como recorrerlo, insertar un nuevo nodo, eliminar uno existente o buscar un valor determinado. Es importante mencionar que el proceso de generación de un árbol dependerá de las reglas impuestas por una aplicación particular y el procedimiento de generación deberá, por tanto, reproducir de la manera más eficiente posible esas reglas de generación.

Representación de los árboles binarios

Si el árbol es lleno o árbol completo, es posible encontrar una buena representación secuencial del mismo. En esos casos, los nodos pueden ser almacenados en un arreglo unidimensional, A , de manera que el nodo

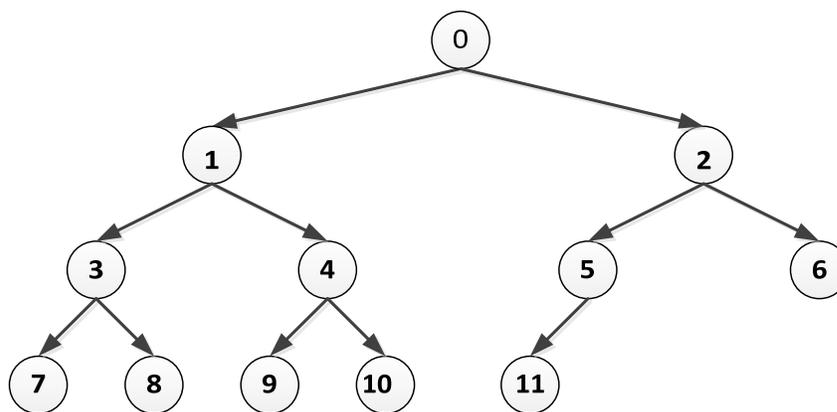
	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página:	99/180
		Sección ISO	8.3
		Fecha de emisión:	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

numerado como i se almacena en $A[i]$. Esto permite localizar fácilmente las posiciones de los nodos padre, hijo izquierdo e hijo derecho de cualquier nodo i en un árbol binario arbitrario que cumpla tales condiciones.

Entonces, si un árbol binario completo con n nodos se representa secuencialmente, se cumple que, para cualquier nodo con índice i , $0 \leq i \leq (n - 1)$, y se tiene que:

- El padre del nodo i estará localizado en la posición $(i - 1)/2$ si $i > 0$. Si $i = 0$, se trata del nodo raíz y no tiene padre.
- El hijo izquierdo del nodo i estará localizado en la posición $2 * (i + 1) - 1$ si $2 * (i + 1) - 1 < n$. Si $2 * (i + 1) - 1 \geq n$, el nodo no tiene hijo izquierdo.
- El hijo derecho del nodo i estará localizado en la posición $2 * (i + 1)$ si $2 * (i + 1) < n$. Si $2 * (i + 1) \geq n$, el nodo no tiene hijo derecho.

Ejemplo en la figura 8.9 se tienen tres arreglos, padre, hijo izquierdo e hijo derecho, el índice de cada arreglo representa al nodo y el contenido de cada arreglo la posición de su padre, hijo izquierdo e hijo derecho.



Posición	0	1	2	3	4	5	6	7	8	9	10	11
Padre	--	0	0	1	1	2	2	3	3	4	4	5
Hijo Izquierdo	1	3	5	7	9	11	--	--	--	--	--	--
Hijo Derecho	2	4	6	8	10	--	--	--	--	--	--	--

Figura 8.9

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	100/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

También es posible representar cada nodo de un árbol como un objeto o estructura, donde cada nodo contiene como atributos el valor, y las referencias a los nodos hijos su padre. En árboles binarios se pueden tener los atributos, padre, hijo izquierdo e hijo derecho los cuales pueden hacer referencia al nodo raíz, nodo izquierdo o nodo derecho respectivamente.

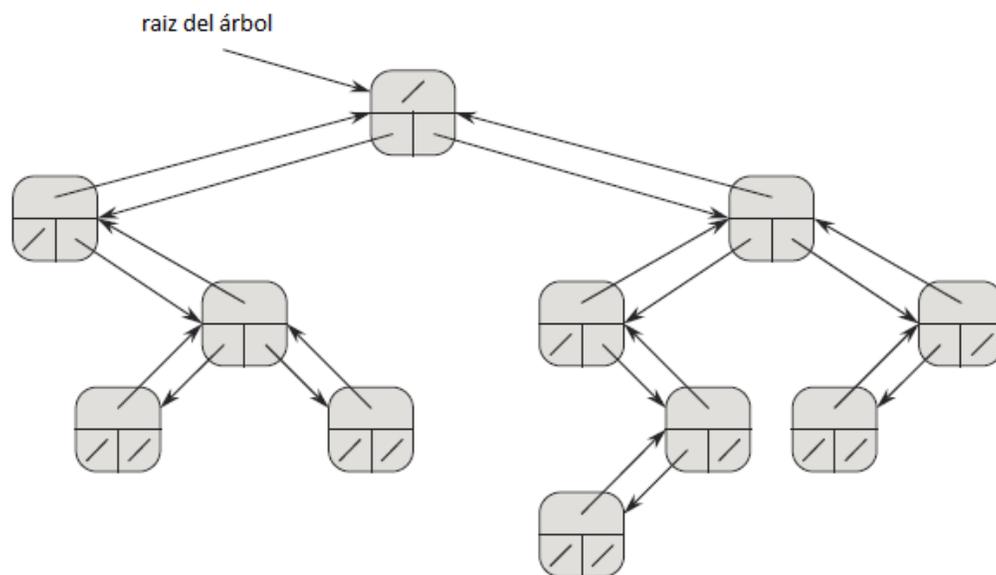


Figura 8.10[2]

Aplicaciones de los árboles binarios.

Una aplicación es la representación de otro tipo de árboles. Esto es importante porque resulta más complejo manipular nodos de grado variable (número variable de relaciones) que nodos de grado fijo. Entonces es posible establecer una relación de equivalencia entre cualquier árbol no binario y un árbol binario, es decir obtener un árbol binario equivalente.

Otra aplicación de los árboles binarios es hallar soluciones a problemas cuyas estructuras son binarias, por ejemplo, las expresiones aritméticas y lógicas.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	101/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Árboles binarios de búsqueda.

Son árboles binarios donde los nodos se identifican por una llave k y no existen dos elementos con la misma, además sus nodos están ordenados de manera conveniente para la búsqueda de alguna llave como se indica:

- Todos los elementos almacenados en el subárbol izquierdo de un nodo con valor o llave k , tienen valores menores a k .
- Todos los elementos almacenados en el subárbol derecho de un nodo con valor o llave k , tiene valores mayores o iguales a k .

Tienen la característica de que sus subárboles izquierdo y derecho son también árboles binarios de búsqueda.

Un algoritmo en pseudocódigo propuesto en [2] para realizar la búsqueda en este tipo de árboles es:

Busqueda (nodoX, llave)

Inicio

Si (nodoX==Ninguno) o (valor==nodoX.llave)
Retornar nodoX

Fin Si

Si llave < nodox.llave
Retornar Busqueda(nodoX.hijoIzq, llave)

En otro caso
Retornar Busqueda(nodoX.hijoDer, llave)

Fin Si

Fin

El procedimiento recibe información del nodo y la llave para buscar de forma recursiva en el subárbol izquierdo o derecho según el valor de la llave.

En este tipo de árboles la obtención de las llaves se puede realizar en un orden dependiendo del recorrido que se realice (*preorden*, *in-orden*, *post orden*) y lo anterior se consigue con un procedimiento recursivo. Por ejemplo, para el recorrido en *inorden* se tiene:

In_Orden(Nodo)

inicio

Si es Nodo valido

In_Orden(Nodo.hijoIzquierdo)

Imprimir Nodo.llave

In_Orden(Nodo.derecho)

Fin Si

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	102/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

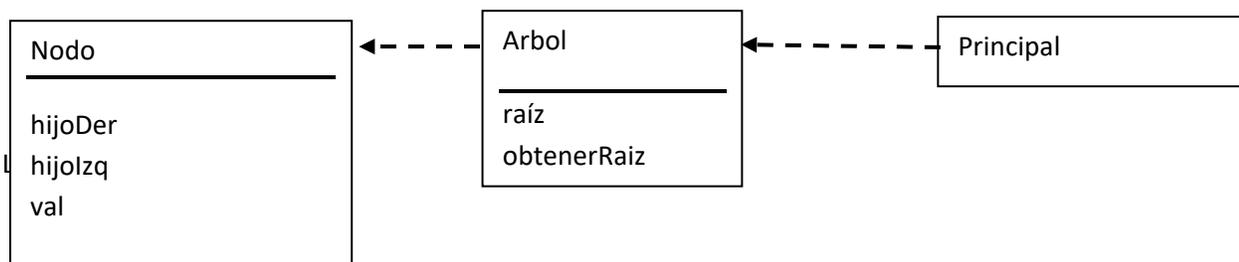
Se pueden entonces tener diferentes operaciones a realizar en esta estructura de datos, como crear nodo, insertar nodo, borrar, recorrido, buscar etc.

Desarrollo

Actividad 1:

Realizar un programa que forme un árbol binario de búsqueda, para lo cual se seguirá el diseño e implementación sugerido abajo.

Primero se formará el árbol binario de acuerdo al siguiente diagrama de clases y después se agregarán algunos métodos propios de un árbol binario de búsqueda.



Las implementaciones de la clase *Nodo* y *Arbol* en Python son las siguientes:

```

class Nodo:
    def __init__(self, valor):
        self.hijoIzq = None
        self.hijoDer = None
        self.val = valor
  
```

```

class Arbol:
    def __init__(self):
        self.raiz = None

    def obtenerRaiz(self):
        return self.raiz
  
```

Después, para insertar nodos se utilizará el criterio de árboles binarios de búsqueda, donde para mantener un orden entre los elementos, cualquier elemento menor está en la rama izquierda, mientras que cualquier

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	103/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

elemento mayor está en la rama derecha. Entonces se colocan los métodos *agregar()* y *agregarNodo()* a la clase *Arbol* para ir formando el árbol binario de acuerdo al criterio mencionado.

```
def agregar(self, val):
    #Si árbol vacío, agregar nodo raíz
    if(self.raiz == None):
        self.raiz = Nodo(val)
    else:
        #Si el árbol tiene raíz
        self.agregarNodo(val, self.raiz)

def agregarNodo(self, val, nodo):
    #Si el valor a introducir es menor al valor que se encuentra
    #en el nodo actual se revisa el hijo izquierdo
    if(val < nodo.val):
        #Si hay hijo izquierdo
        if(nodo.hijoIzq != None):
            self.agregarNodo(val, nodo.hijoIzq)
        else:
            #Si no hay hijo izquierdo se crea un nodo con el valor
            nodo.hijoIzq = Nodo(val)
    #Si el valor a agregar es mayor al valor que tiene el nodo actual
    #Se revisa hijo derecho
    else:
        if(nodo.hijoDer != None):
            self.agregarNodo(val, nodo.hijoDer)
        else:
            nodo.hijoDer = Nodo(val)
```

Ahora para poder visualizar al árbol se adicionan los métodos *preorden()* e *imprimePreorden()* que permiten recorrerlo en preorden.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	104/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```

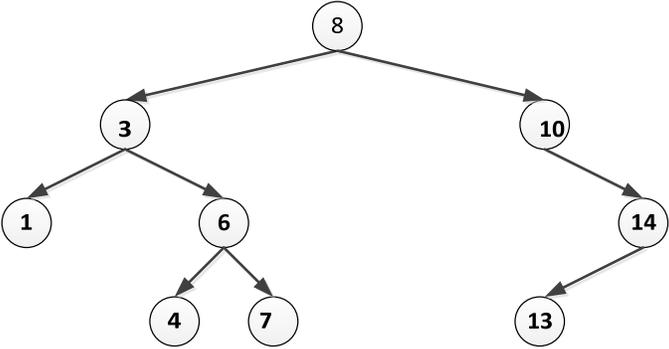
def preorden(self,nodo):
    if(nodo != None):
        print (str(nodo.val))
        if nodo.hijoIzq !=None:
            self.preorden(nodo.hijoIzq)

        if nodo.hijoDer != None :
            self.preorden(nodo.hijoDer)

def ImprimePreorden(self):
    if(self.raiz != None):
        self.preorden(self.raiz)

```

Finalmente con las clases *Nodo* y *Arbol* completas, se tiene que realizar una clase controladora que forme el siguiente árbol y se liste en preorden.



Actividad 2

Modificar el programa de la actividad 1 para que se realice la búsqueda de un valor dado en el árbol binario.

Para ello se puede implementar el algoritmo en pseudocódigo propuesto en [2] y que se menciona en esta guía en el apartado de árboles binarios

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	105/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Actividad 3

Ejercicios sugeridos por el profesor

Referencias

[1]Zenón José

Fundamentos de Estructura de Datos

Thomson

[2]CORMEN, Thomas, LEISERSON, Charles,et al.

Introduction to Algorithms

3rd edition

MA, USA

The MIT Press, 2009

[3]Ziviani, Nivio

Diseño de algoritmos con implementaciones en Pascal y C

Ediciones paraninfo /Thomson Learning

2007

[5] Alfred V. Aho, Jeffrey D. Ullman y John E. Hopcroft

Estructura de datos y algoritmos

S.A. ALHAMBRA MEXICANA

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	106/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 9

Árboles parte 2

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	107/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 9

Estructura de datos y Algoritmos II

Árboles. Parte 2.

Objetivo: El estudiante conocerá e identificará las características de los árboles-B.

Actividades

Implementar algunas operaciones realizadas sobre un árbol-B en algún lenguaje de programación.

Antecedentes

- Análisis previo del concepto de árbol-B y su representación visto en clase teórica.
- Manejo de listas, diccionarios, estructuras de control, funciones y clases en Python 3.
- Conocimientos básicos de la programación orientada a objetos.

Introducción

En los sistemas computacionales se cuenta con dos sistemas de almacenamiento en dos niveles, el almacenamiento en memoria principal y el almacenamiento secundario (como en discos magnéticos u otros periféricos).

En los almacenamientos en memoria secundaria el costo es más significativo y viene dado por el acceso a este tipo de memoria, muy superior al tiempo necesario para acceder a cualquier posición de memoria principal. Entonces al tratarse de un dispositivo periférico y, por tanto, de acceso lento, resulta primordial minimizar en lo posible el número de accesos.

Para realizar un acceso a un disco (para lectura / escritura) se requiere de todo un proceso de movimientos mecánicos que toman un determinado tiempo, por lo que para amortizar el tiempo gastado en esperar por todos esos movimientos la información es dividida en páginas del mismo tamaño en bits que se encuentran en las llamadas pistas del disco, así cada lectura y/o escritura al disco es de una o más páginas.

Los **árboles-B** (en inglés B-Tree), se utilizan cuando la cantidad de datos que se está trabajando es demasiado grande que no cabe toda en memoria principal. Así que, son árboles de búsqueda balanceados diseñados para trabajar sobre discos magnéticos u otros accesos o dispositivos de almacenamiento secundario.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	108/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En los algoritmos que utilizan árboles-B se copian las páginas necesarias desde el disco a memoria principal y una vez modificadas las escriben de regreso al disco.

El tiempo de ejecución del algoritmo que utilice un árbol-B depende principalmente del número de lecturas y escrituras al disco, por lo que se requiere que estas operaciones lean y escriban lo más que se pueda de información y para ello cada nodo de un árbol-B es tan grande como el tamaño de una página completa lo que pone un límite al número de hijos de cada nodo.

Para un gran árbol-B almacenado en disco, se tienen factores de ramificación de entre 50 y 2000, dependiendo del número de llaves o valores que tenga cada nodo (tamaño de la página). Un factor grande de ramificación reduce tanto el peso del árbol como el número de accesos al disco requerido para encontrar una llave o valor.

En la figura 9.1 se muestra un árbol-B con un factor de ramificación de 1001 y peso 2 que puede almacenar más de un billón de llaves. Dentro de cada nodo x se encuentran el atributo $x.n$ que indica el número de llaves que son 1000 por nodo.

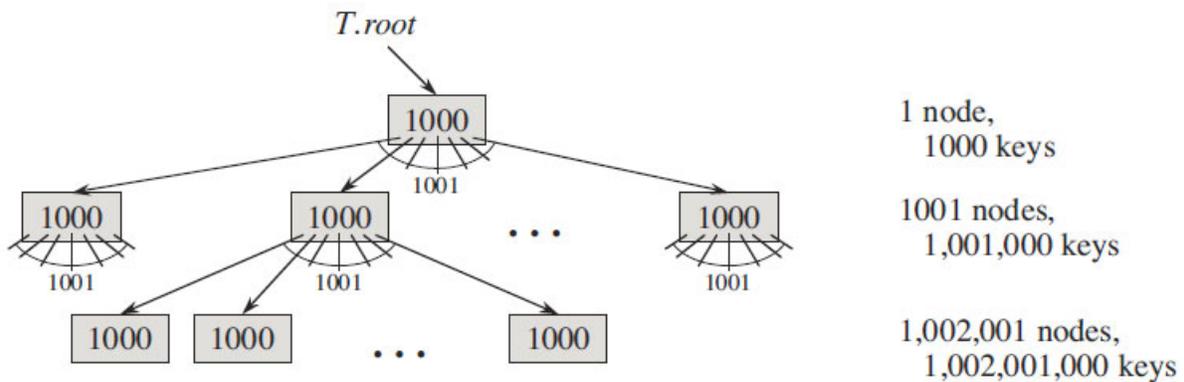


Figura 9.1. [1]

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	109/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Árboles B

Un Árbol-B es un árbol multirrama (con raíz T.Raiz) y tiene las siguientes propiedades:

1) Cada nodo x tiene la forma

$$((x.h_1, x.llave_1), (x.h_2, x.llave_2), (x.h_3, x.llave_3), \dots, (x.h_n, x.llave_n))$$

lo que indica que como mucho tiene n hijos $(x.h_1, x.h_2, \dots, x.h_n)$ y cuenta con las siguientes propiedades.

- a. Tiene $x.n$ llaves [o pares (llave , valor)] almacenadas en el nodo x .
- b. Las $x.n$ llaves, $x.llave_1, x.llave_2, x.llave_3, \dots, x.llave_n$ están ordenadas y almacenadas en orden creciente tal que $x.llave_1 \leq x.llave_2 \leq x.llave_3 \leq \dots \leq x.llave_n$.
- c. Puede ser hoja o nodo interno, se utiliza el atributo $x.hoja$ que contiene un valor verdadero si es hoja y falso si es nodo interno.

2) Cada nodo interno contiene $x.n + 1$ referencias o apuntadores a sus hijos $x.h_1, x.h_2, x.h_3, \dots, x.h_{n+1}$. Los nodos hojas no tienen hijos.

3) Las llaves $x.llave_i$ separan los rangos de las llaves almacenadas en cada sub-árbol. Si k_i es una llave almacenada en el sub-árbol $x.h_i$ entonces:

$$k_i \leq x.llave_1 \leq x.llave_2 \leq x.llave_3 \leq \dots \leq x.llave_n \leq x.llave_{n+1}$$

4) Todas las hojas tienen la misma profundidad y el árbol tiene profundidad h .

5) Existe una cota superior e inferior sobre el número de llaves que puede contener un nodo. Esta cota puede ser expresada en términos de un entero $t \geq 2$ llamado el grado mínimo del árbol-B:

- a. Todo nodo que no sea la raíz debe tener al menos $t - 1$ llaves. Todo nodo interno que no sea la raíz debe tener al menos t hijos. Si el árbol es vacío, la raíz debe tener al menos una llave.
- b. Todo nodo puede contener a lo más $2t - 1$ llaves. Por lo tanto, un nodo interno puede tener a lo más $2t$ hijos. Se dice que el nodo está lleno si este contiene exactamente $2t - 1$ llaves. El árbol-B más simple ocurre para cuando $t = 2$. Todo nodo interno entonces tiene ya sea 2, 3, o 4 hijos, también llamado árbol 2-3-4.

En la figura 9.2 se muestra el ejemplo de un árbol de altura 3 donde su grado mínimo es $t = 2$ y cada nodo puede tener a lo más $2t - 1$ llaves, en este caso 3 y al menos $t - 1$, que es 1.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	110/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

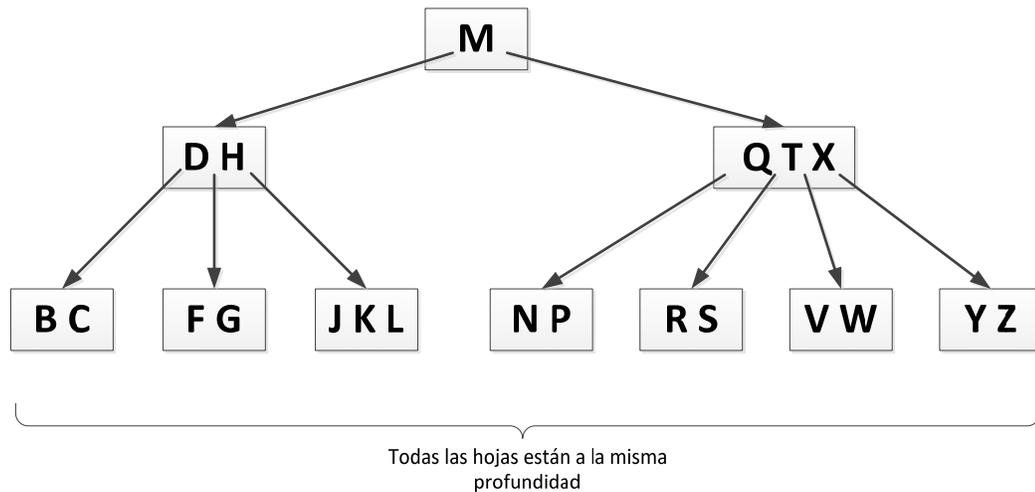


Figura 9.2 [1]

Operaciones Básicas

Algunas de las operaciones básicas realizadas en árboles B son búsqueda, inserción y eliminación de una llave. En esta práctica se trabajará con la búsqueda y la inserción.

Creación del árbol

Para la creación del árbol, lo primero es crear un nodo vacío y después ir insertando llaves e ir creando otros nodos si es necesarios. Para ello en este documento se utilizan los procedimientos propuestos en [1] B-TREE-CREATE() para crear un nodo raíz vacío y B-TREE-INSERT() para agregar nuevas llaves. En ambas funciones se utiliza un procedimiento auxiliar ALLOCATE-NODE(), el cual asigna una página del disco a un nuevo nodo. Se asume que cada nodo creado no requiere una lectura a disco desde que todavía no se utiliza la información almacenada en el disco para ese nodo.

El pseudocódigo del procedimiento B-TREE-CREATE () es [1]:

B-TREE-CREATE (T)

Inicio

```

x=ALLOCATE-NODE()
x.hoja=Verdadero
x.n=0
escribirDisco(x)
T.raiz=x
  
```

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	111/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Insertar una llave

La inserción es un poco más complicada que en un árbol binario. En un árbol-B no se puede solo crear una nueva hoja e insertarla porque daría lugar a ya no tener un árbol-B. Lo que se hace es insertar una nueva llave en un nodo hoja existente. Como no se puede insertar una llave en un nodo hoja que está lleno, se introduce una operación que divide el nodo lleno (que tiene $2t - 1$ llaves) en dos, en torno a la llave del medio ($y.llave_i$). Los dos nodos van a tener $t - 1$ llaves cada uno, y la llave del medio ($y.llave_i$) se moverá a su nodo padre para identificar el punto de división entre los dos nuevos árboles (Figura 9.3). Pero si el padre está lleno, también se debe dividir antes de que se inserte la nueva llave, por lo que el proceso se repetiría con los nodos más arriba hasta llegar al nodo raíz, en ese caso se genera un nuevo nodo raíz que contendrá solo el elemento desplazado hacia arriba de la antigua raíz.

De esta forma, un árbol-B crece por la raíz, de manera que, siempre que la altura se incrementa en 1, la nueva raíz sólo tiene un elemento.

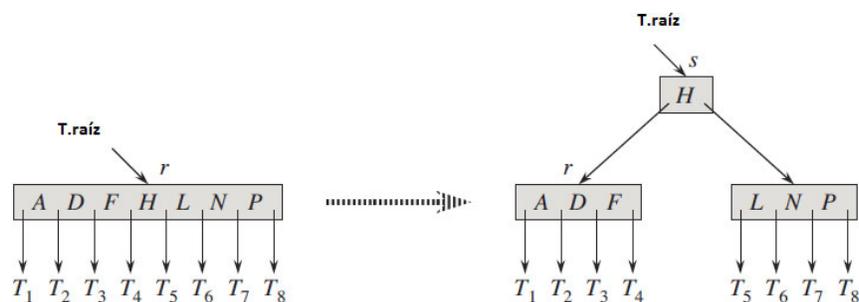


Figura 9.3 [1]

Para insertar en una sola pasada del nodo raíz a la hoja, primero se recorre el árbol buscando la posición donde se colocará la llave y en el camino se revisan qué nodos están llenos para dividirlos antes de realizar la inserción.

Cuando la llave encuentra su lugar en un nodo que no está lleno el proceso de inserción queda limitado a dicho nodo, pero si no lo está, el proceso puede implicar la creación de un nuevo nodo.

Ejemplo:

En la figura 9.4 se presenta un árbol-B donde su grado mínimo es $t = 3$, tal que cada nodo puede mantener 5 llaves.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	112/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

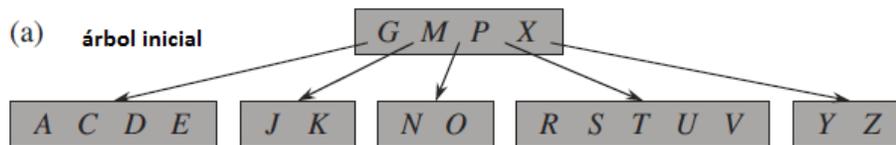


Figura 9.4. [1]

En las siguientes imágenes con color gris claro se muestran los nodos que se modifican conforme se van insertando las llaves que se mencionan.

Para insertar la *B*, se localiza primero donde se colocará la llave (en el nodo *ACDE*), y como tanto el nodo donde se insertará como su raíz no está lleno, solo se coloca en la posición adecuada. Figura 9.5.

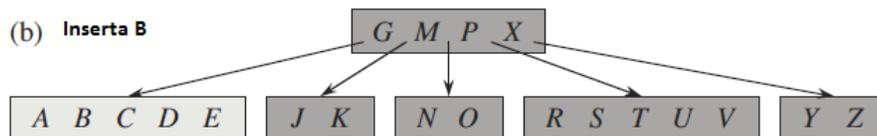


Figura 9.5. [1]

Ahora para insertar *Q* el nodo *RSTUV* se divide en 2 nodos que contienen *RS* y *UV*. La llave *T* se mueve a la raíz y *Q* se inserta al inicio de la mitad de la izquierda (nodo *RS*). Figura 9.6

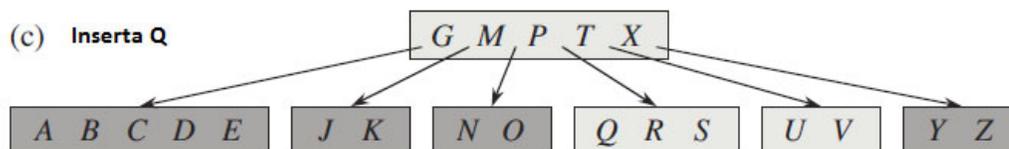


Figura 9.6. [1]

A partir del árbol anterior, para insertar la llave *L*, como la raíz está llena se divide en dos, además de formarse un nuevo nodo que es la nueva raíz. Después *L* se inserta en el nodo hoja que contiene *JK*. Figura 9.7.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	113/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

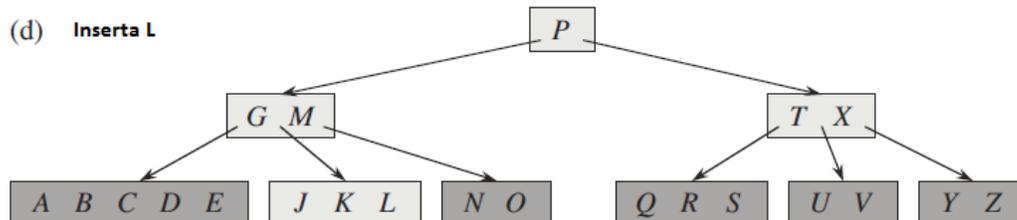


Figura 9.7 [1]

Finalmente, para insertar F , el nodo $ABCDE$ se divide en dos y el valor de en medio, en este caso C se coloca en el nodo padre, que no está lleno. Figura 9.8.

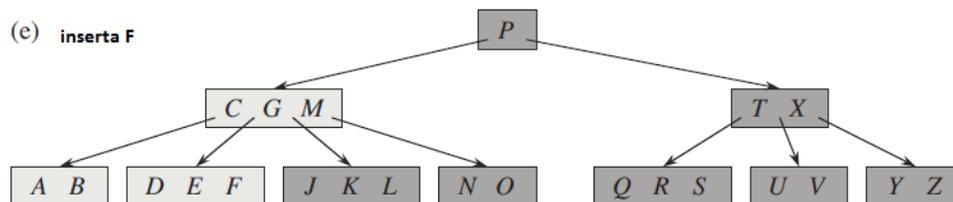


Figura 9.8 [1]

Un pseudocódigo de la inserción es el siguiente:

B-TREE-INSERT(T, k)

Inicio

```

r = T.raiz
Si r.n == 2t - 1
  s = ALLOCATE-NODO()
  T.raiz = s
  s.hoja = FALSO
  s.n = 0
  s.h1 = r
  B-TREE-SPLIT-CHILD(s, 1)
  B-TREE-INSERT-NONFULL(s, k)
En otro caso
  B-TREE-INSERT-NONFULL(r, k)

```

Fin Si

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	114/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Este procedimiento tiene dos casos, primero cuando el nodo raíz está lleno ($r.n == 2t - 1$), éste se divide (con B-TREE-SPLIT-CHILD()) y se forma un nuevo nodo s que será la nueva raíz, esto hace que se incremente la altura del árbol. El segundo caso se da cuando la raíz no está llena y se puede insertar la llave con TREE-INSERT-NONFULL().

El procedimiento B-TREE-INSERT-NONFULL() es recursivo, de forma tal que asegura revisar los niveles de abajo del árbol para insertar la llave en el nodo y posición adecuada y también garantiza que el nodo que se analiza, si está lleno, sea dividido con el llamado a la función B-TREE-SPLIT-CHILD() cuando sea necesario. A continuación, se muestran los pseudocódigos de estas funciones [1].

B-TREE-INSERT-NONFULL(x,k)

Inicio

$i = x.n$

Si $x.hoja == verdadero$

Mientras $i \geq 1$ y $k < x.llave_i$

$x.llave_{i+1} = x.llave_i$

$i = i - 1$

Fin Mientras

$x.llave_{i+1} = k$

$x.n = x.n + 1$

EscribirADisco(x)

En otro caso

Mientras $i \geq 1$ y $k < x.llave_i$

$i = i - 1$

Fin Mientras

$i = i + 1$

Leer Del Disco($x.h_i$)

Si $x.h_i.x == 2t - 1$

B-TREE-SPLIT-CHILD($x.i$)

Si $k > x.llave_i$

$i = i + 1$

Fin Si

Fin Si

B-TREE-INSERT-NONFULL($x.h_i, k$)

Fin Si

Fin

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	115/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

B-TREE-SPLIT-CHILD(x, i)

Inicio

$z = \text{ALLOCATE-NODE}()$

$y = x.h_i$

$z.hoja = y.hoja$

$z.n = t - 1$

Para $j = 1$ hasta $j = t - 1$

$z.llave_j = y.llave_{j+t}$

Fin Para

Si $y.hoja == \text{FALSO}$

Para $j = 1$ hasta t

$z.h_j = y.h_{j+t}$

Fin para

Fin Si

$y.n = t - 1$

Para $j = x.n + 1$ decrementando hasta $j = i + 1$

$x.h_{j+1} = x.h_j$

Fin Para

$x.h_{i+1} = z$

Para $j = x.n$ decrementando hasta $j = i$

$x.llave_{j+1} = x.llave_j$

Fin Para

$x.llave_i = y.llave_t$

$x.n = x.n + 1$

EscribirADisco(y)

EscribirADisco(z)

EscribirADisco(x)

Fin

Búsqueda

La búsqueda en un árbol-B se parece mucho a la búsqueda en un árbol binario, excepto que en lugar de hacerlo binario o de dos caminos, se hace una decisión de múltiples caminos de acuerdo al número de hijos que tiene el nodo.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	116/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Un algoritmo general para buscar la llave k puede ser:

- 1-Seleccionar nodo actual igual a la raíz del árbol.
- 2-Comprobar si la llave se encuentra en el nodo actual:
 - Si la llave está, termina algoritmo
 - Si la llave no está:
 - Si nodo actual es hoja, termina algoritmo
 - Si nodo actual no es hoja y n es el número de llaves en el nodo,
 - Nodo actual == hijo más a la izquierda, si $k < k_1$
 - Nodo actual == hijo más a la derecha a la derecha, si $k > k_n$
 - Nodo actual == i -ésimo hijo, si $k_i < k < k_{i+1}$
 - Volver al paso 2.

La figura 9.9 se puede observar un árbol-B cuyas llaves son las letras del alfabeto, cada nodo x contiene $x.n$ llaves y tiene $x.n + 1$ hijos, y además todas las hojas tienen la misma profundidad. También se ilustra en gris claro el progreso de la búsqueda de la llave R en ese árbol.

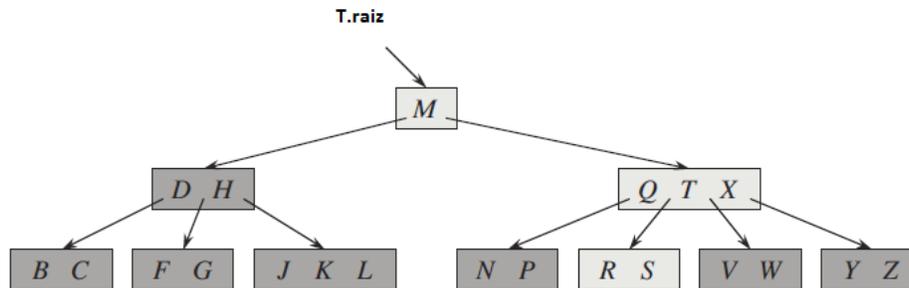


Figura 9.9[1]

Una manera de plantear el algoritmo es de forma recursiva, a continuación se presenta un procedimiento para la búsqueda en árboles-B (propuesto en [1]) donde se toma como entrada un apuntador al nodo raíz x de cada sub árbol y una llave k que se buscará dentro de cada subárbol. La llamada inicial al procedimiento se realiza con el nodo raíz y la llave a buscar k , esto es, $BTreeSearch(T.raiz, K)$. Si la llave k está dentro del árbol-B la función retorna el par ordenado (y, i) consistente de un nodo y y un índice i tal que $y.llave_i = k$. De otra forma el procedimiento regresa nulo.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	117/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

B-TREE-SEARCH(x, k)

Inicio

$i=1$

Mientras $i \leq x.n$ y $k > x.llave_i$

$i=i+1$

Fin Mientras

Si $i \leq x.n$ y $k > x.llave_i$

retorna (x, i)

Si no

Si $x.hoja == verdadero$

Retorna Ninguno

Si no

Leer-Disco($x.h_i$)

Retorna B-TREE-SEARCH($x.h_i, k$)

Fin Si

Fin Si

Fin

En este procedimiento se utiliza inicialmente una búsqueda lineal para encontrar el índice i tal que $k \leq x.llave_i$ o de lo contrario se coloca i a $x.n + 1$. Si se ha descubierto la llave k , se retorna el nodo x junto con su ubicación (x, i) . De otra forma se determina que la búsqueda no fue exitosa porque el nodo x es una hoja, o se llama recursivamente a la función B-TREE-SEARCH() indicando el subárbol de x adecuado, para seguir buscando. Los parámetros de la nueva llamada a B-TREE-SEARCH($x.h_i, k$) se obtiene después de efectuar una lectura a disco para obtener la información de ese nodo hijo.

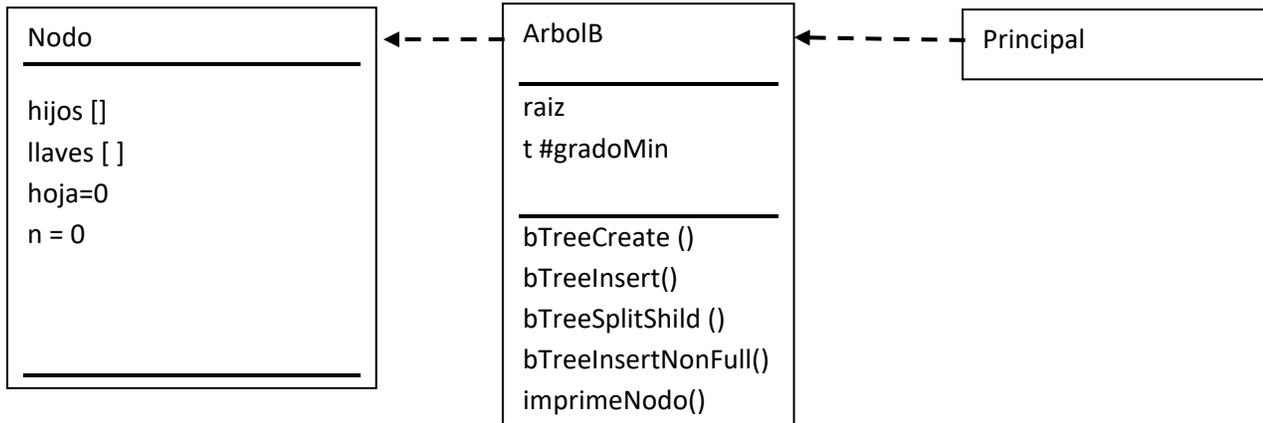
Desarrollo

Actividad 1

Se realizará de forma guiada un programa en Python que permita crear un árbol B e ir insertando datos. Para ello se utilizarán los pseudocódigos de los procedimientos explicados en este documento.

Primero se plantea el siguiente diagrama de clases:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	118/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			



Para la realización del programa se asumirá que los datos de los nodos ya no están en disco. Es decir, todo se trabajará en memoria principal.

La clase Nodo queda:

```

#Autor Elba Karen Sáenz García
class Nodo:
    def __init__(self,t):
        self.hijos = list()
        self.llaves = list()
        self.hoja=1
        self.n=0
        for k in range(2*t):
            self.llaves.append([None])
        for k in range(2*t+1):
            self.hijos.append([None])
  
```

El inicio de la clase ArbolB es:

```

#Auto Elba Karen Sáenz García
class ArbolB:
    def __init__(self,gradoMinimo):
        self.t= gradoMinimo
        self.raiz = None
  
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	119/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Ahora se muestran los métodos que hay que ir agregando a la clase ArbolB de acuerdo al diagrama de clases planteado y los pseudocódigos vistos en este documento.

El método bTreeCreate() crea un nuevo nodo :

```
def bTreeCreate(self):
    if(self.raiz == None):
        self.raiz = Nodo(self.t)
    return self.raiz
```

El método bTreeSplitShild():

```
def bTreeSplitShild(self,x,i):
    z=Nodo(self.t)
    y = x.hijos[i]
    z.hoja=y.hoja
    z.n=self.t-1

    for j in range(1,self.t):
        z.llaves[j]=y.llaves[j+self.t]
        y.llaves[j+self.t]=None

    if y.hoja==0:
        for j in range(1,self.t+1):
            z.hijos[j]=y.hijos[j+self.t]
            y.hijos[j+self.t]=None
    y.n=self.t-1
    for j in range(x.n+1,i,-1):
        x.hijos[j+1]=x.hijos[j]

    x.hijos[i+1]=z

    for j in range (x.n,i-1,-1):
        x.llaves[j+1]=x.llaves[j]
    x.llaves[i]=y.llaves[self.t]
    y.llaves[self.t]=None
    x.n=x.n+1
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	120/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

El método bTreeInsertNonFull():

```
def bTreeInsertNonFull(self, x, k):
    i=x.n
    if x.hoja == 1:
        while ( i >= 1) and (k < x.llaves[i]):
            x.llaves[i+1]= x.llaves[i]
            i=i-1
        x.llaves[i+1]=k
        x.n=x.n+1
        #escribir a disco
    else:
        #No es hoja
        while (i >= 1) and (k < x.llaves[i]):
            i=i-1
        i=i+1
        #leer disco
        if x.hijos[i].n == 2*self.t-1:
            self.bTreeSplitShild(x,i)
            if k > x.llaves[i]:
                i=i+1
        self.bTreeInsertNonFull(x.hijos[i],k)
```

El método bTreeInsert():

```
def bTreeInsert(self,nodo, k):
    r=self.raiz
    #nodo lleno
    if r.n == 2*self.t-1:
        s=Nodo(self.t)
        self.raiz=s
        s.hoja=0
        s.n=0
        s.hijos[1]=r
        self.bTreeSplitShild(s,1)
        self.bTreeInsertNonFull(s,k)
    else:
        self.bTreeInsertNonFull(r,k)
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	121/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

El método `imprimeNodo()`:

```
def imprimeNodo(self,nodo):
    for i in range(1,2+self.t,1):
        if (nodo.llaves[i] != None):
            print( nodo.llaves[i])
```

Una vez terminadas las dos clases se requiere realizar una controladora que permita ir formando un árbol B, creándolo e insertando la siguiente secuencia.

B,T,H,M,O,C,Z,G,L,E,N,P,R,D,J,Q,F,W,X.

Además, se irá mostrando la formación del árbol.

En la controladora, primero se colocará la inserción de B, T y H como se muestra en el siguiente código:

```
BT=ArbolB(2)

actual=BT.bTreeCreate()

print ("Se insertara B")
#print (BT.raiz.llaves)
BT.bTreeInsert(actual,ord("B"))

print ("Se insertara T")
BT.bTreeInsert(actual,ord("T"))

print ("Se insertara H")
BT.bTreeInsert(actual,ord("H"))

print ("Imprime raíz")
BT.imprimeNodo(BT.raiz)
```

. El proceso que se realiza se observa en la figura 9.10.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	122/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

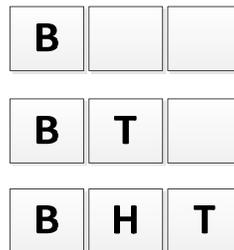


Figura 9.10

Después se insertará M y como ya no hay lugar en el nodo, se realiza la división de mismo y la creación de uno nuevo, como se muestra en el siguiente código y en la figura 9.11:

```

print ("Se insertara M")
BT.bTreeInsert(actual, ord("M"))

print (BT.raiz.llaves)
print (BT.raiz.hijos[1].llaves)
print (BT.raiz.hijos[2].llaves)

```

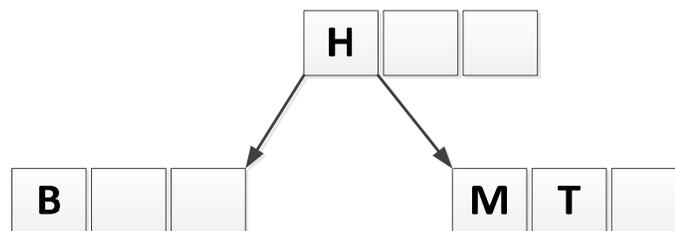


Figura 9.11

Ahora se insertarán las llaves O y C, además de visualizar la raíz y sus hijos. Como hay espacio en los nodos solo se colocan en el lugar correspondientes. El árbol resultante se muestra en la figura 9.12.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	123/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

print ("Se insertara O")
BT.bTreeInsert(actual,ord("O"))
print ("Se insertara C")

BT.bTreeInsert(actual,ord("C"))
print (BT.raiz.llaves)
print (BT.raiz.hijos[1].llaves)
print (BT.raiz.hijos[2].llaves)

```

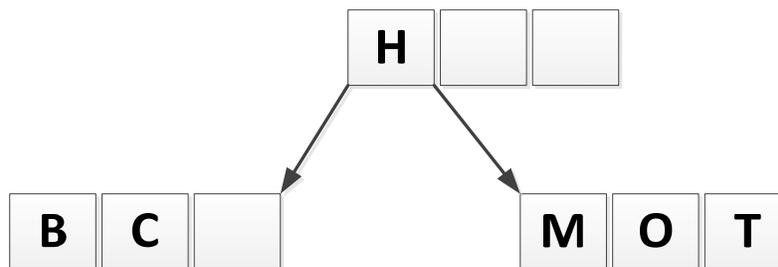


Figura 9.12

Cuando se inserta Z, no hay lugar en el nodo correspondiente, así que se realiza una división y creación de nodo.
Figura 9.13.

```

print ("Se insertara Z")
BT.bTreeInsert(actual,ord("Z"))

print (BT.raiz.llaves)
print (BT.raiz.hijos[1].llaves)
print (BT.raiz.hijos[2].llaves)
print (BT.raiz.hijos[3].llaves)

```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	124/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

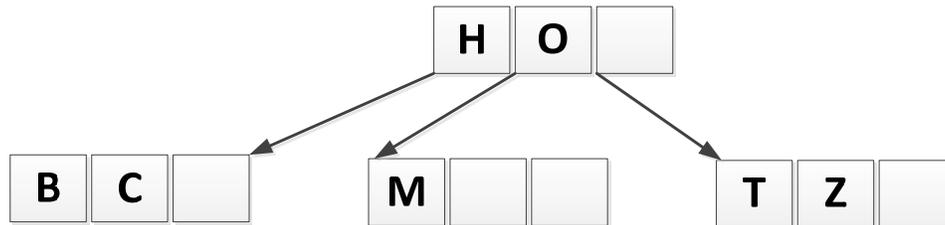


Figura 9.13

Actividad 2

Terminar de insertar la secuencia e ir mostrando en la salida estándar y dibujar como va quedando el árbol.

Actividad 3

Ejercicios sugeridos por el profesor

Referencias

[1]CORMEN, Thomas, LEISERSON, Charles, et al.

Introduction to Algorithms

3rd edition

MA, USA

The MIT Press, 2009

[2]Ziviani, Nivio

Diseño de algoritmos con implementaciones en Pascal y C

Ediciones paraninfo /Thomson Learning

2007

[3] <https://www.youtube.com/watch?v=HHoWd93mKjA>

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	125/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 10

Archivos

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	126/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 10

Estructura de datos y Algoritmos II

Archivos

Objetivo: El estudiante conocerá e identificará aspectos sobre los archivos, como las operaciones, el tipo de acceso y organización lógica.

Actividades

Trabajar con operaciones, tipo de acceso y organización lógica de TDA archivo en algún lenguaje de programación.

Antecedentes

- Análisis previo del concepto de archivo, operaciones y organización visto en clase teórica.
- Manejo de listas, estructuras de control, funciones en Python 3.
- Conocimientos básicos de la programación orientada a objetos.

Introducción

El sistema operativo hace una abstracción de las propiedades físicas de los dispositivos de almacenamiento secundario para definir una unidad lógica de almacenamiento, el archivo.

Un archivo es una colección de información relacionada con un nombre que se guarda en memoria secundaria y comúnmente representan programas y datos. En general es una secuencia de bits, bytes, líneas o registros cuyo significado es definido por el creador y usuario del archivo. Se puede visualizar como espacios de direcciones lógicas contiguas.

Atributos de un archivo

Además del nombre un archivo tiene otros atributos que varían de un sistema operativo a otro, en general son:

- Nombre: Nombre simbólico visible al usuario.
- Tipo: Esta información es necesaria cuando se pueden trabajar diferentes tipos.
- Ubicación.: Es un apuntador a un dispositivo y a la ubicación del archivo en el dispositivo

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	127/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

- Tamaño. Se tiene el tamaño actual (en bytes, palabras o bloques) y posiblemente el tamaño máximo permitido.
- Protección. Información del control de acceso, que determina quién puede leer, escribir, ejecutar, etc., el archivo
- Hora, fecha e identificación del usuario. Estos datos pueden ser útiles para la protección de seguridad y control de uso.

Operaciones sobre archivos

Un archivo es un tipo de dato abstracto (TDA) que se manipula por medio de una interfaz ya sea por el SO o por algún programa. Al ser un TDA se tienen operaciones que se realizan sobre él. Las principales son.

- Creación del archivo: Para crear un archivo primero, es necesario encontrar espacio para el mismo
- Escribir en un archivo: Para escribir en un archivo, se debe realizar una llamada al sistema especificando el nombre y la información que se escribirá. Con el nombre se localiza la ubicación así, el sistema mantendrá un apuntador de escritura a la ubicación en el archivo, donde va a tener lugar a la siguiente escritura y este debe actualizarse siempre que ocurra una escritura.
- Lectura del archivo: Para leer un archivo, se realiza una llamada al sistema especificando el nombre y dónde debe colocarse (dentro de la memoria) el siguiente bloque del archivo. Se explora el directorio para hallar la entrada asociada y el sistema necesita mantener un apuntador de lectura que haga referencia a la ubicación dentro del archivo en la que tendrá lugar la siguiente lectura. Una vez que la lectura se completó, se actualiza el apuntador de lectura
- Borrar un archivo: Para borrar un archivo, se explora el directorio en busca del archivo indicado. Una vez encontrada la entrada de directorio asociada, se libera todo el espacio del archivo y se borra la entrada del directorio.

Dado que en las operaciones descritas se realiza primero una búsqueda en el directorio para encontrar la entrada asociada con el archivo, muchos sistemas requieren que se realice una función de apertura (`open()`) antes de utilizar por primera vez el archivo, así cuando se solicita una operación sobre un archivo, se utiliza un índice (descriptor de archivo) en la llamada **tabla de archivos abiertos** que contiene información de todos los archivos abiertos, de manera que no se requiere de una búsqueda. Cuando el archivo deja de utilizarse será cerrado por un proceso y el S. O. eliminará la entrada correspondiente en la tabla de archivos abiertos.

Es importante mencionar que en la función de apertura de un archivo se toma el nombre del mismo y explora el directorio, copiando la entrada del directorio correspondiente en la tabla de archivos abiertos [1]. Esta función además de recibir el nombre del archivo también acepta información acerca del modo de acceso: creación, solo lectura, escritura, lectura-escritura, agregar etc.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	128/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Tipos de Archivos

Existen archivos de diferentes tipos: cada uno podría ser un documento de texto, un binario ejecutable, un archivo de audio o video, etc. Una de las estrategias principales para que el sistema operativo reconozca el tipo de un archivo es la extensión. Cada nombre de archivo se divide en dos porciones, empleando como elemento separador al punto: el nombre del archivo y su extensión. hola.txt, programa.exe.

Estructura interna de los archivos:

Los sistemas de disco normalmente tienen un tamaño de bloque bien definido y determinado por el tamaño de un sector. Todas las operaciones de entrada y salida en disco se realizan en unidades de un bloque (registro físico) y todos los bloques tienen un mismo tamaño. Es poco probable que el tamaño del registro físico corresponda exactamente a la longitud del registro lógico deseado el cual puede variar de longitud. Para resolver este problema se utiliza el llamado empaquetamiento de registros lógicos en bloques físicos [1].

Por ejemplo, el S.O. UNIX define todos los archivos como simples flujos de bytes. Cada byte es direccionable de manera individual, a partir de un desplazamiento con respecto al principio (o al final) del archivo. En este caso el tamaño de cada registro lógico es un byte. El sistema empaqueta y desempaqueta automáticamente los bytes en los bloques físicos del disco (por ejemplo 512 bytes por bloque) según sea necesario [1].

El tamaño del registro lógico, el tamaño del bloque físico y la técnica de empaquetamiento determinan cuántos registros lógicos se almacenan en cada bloque físico.

Métodos de Acceso

Los archivos almacenan información y cuando se requiere utilizarla es necesario tener acceso a ella y leerla en la memoria de la computadora.

El método más simple de acceso a la información es el **secuencial** donde la información se procesa en orden, un registro después de otro. Una operación de lectura lee la siguiente porción del archivo e incrementa automáticamente un apuntador de archivo, que controla la ubicación de E/S. De forma similar, la operación de escritura añade información al final del archivo y hace que el apuntador avance hasta el final de los datos recién escritos (el nuevo final del archivo). Los archivos pueden reinicializarse para situar el apuntador al principio de los mismos, y en algunos sistemas, puede que el programa sea capaz de saltar hacia adelante o hacia atrás n registros, para algún cierto valor n .

Otro método es el acceso **directo, relativo u aleatorio** surge con la llegada de los dispositivos de acceso directo como los discos magnéticos; en este método de acceso el archivo se considera como un conjunto de registros, cada uno de los cuales puede ser un byte. Se puede acceder al mismo desordenadamente moviendo el

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	129/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

apuntador de acceso al archivo a uno u otro registro. Esta forma de acceso se basa en un modelo de archivo almacenado en disco, ya que se asume que el dispositivo se puede mover aleatoriamente entre los distintos bloques que componen el archivo.

Organización.

Hay diferentes maneras en las que puede ser organizada la información de los archivos, así como diferentes formas para tener acceso a dicha información.

Directorios

Algunos sistemas almacenan millones de archivos en terabytes de disco. Para gestionar todos esos datos, se necesitan que se organicen y esa organización implica el uso de directorios.

La forma más simple de un sistema de directorios es tener un directorio que contenga todos los archivos. [2].

En la figura 10.1 se muestra un ejemplo de un sistema con un directorio que contiene cuatro archivos. A menudo se utiliza en dispositivos incrustados simples como teléfonos, cámaras digitales y algunos reproductores de música portátiles.

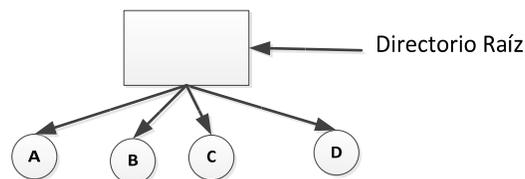


Figura 10.1

Tener un solo nivel es adecuado para aplicaciones dedicadas simples, pero para los usuarios modernos con miles de archivos, sería imposible encontrar algo si todos los archivos estuvieran en un solo directorio.

Lo que se necesita es una jerarquía. Cada directorio puede contener un número arbitrario de archivos, y también puede contener otros directorios (subdirectorios). Los otros directorios pueden contener todavía más archivos y directorios, y así sucesivamente, construyéndose una estructura en árbol en la que un directorio raíz puede contener cualquier número de niveles de otros directorios y archivos. Un ejemplo de este esquema se muestra en la figura 10.2. donde cada uno de los directorios A, B y C contenidos en el directorio raíz pertenecen a un usuario distinto, dos de los cuales han creado subdirectorios para proyectos en los que están trabajando [2].

El uso de directorios hace más fácil organizar los archivos de una manera lógica.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	130/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

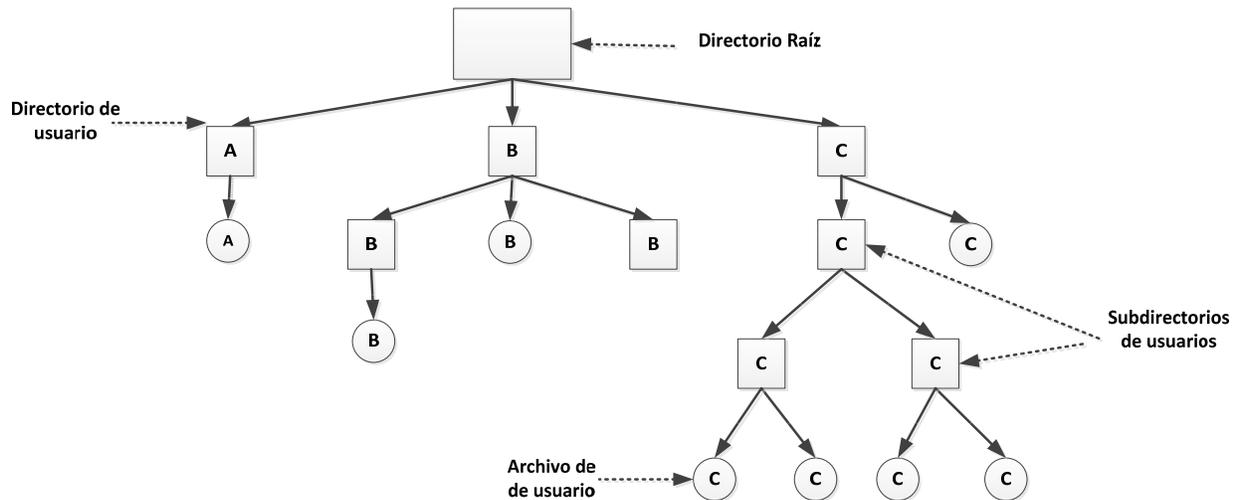


Figura 10.2

Cuando el sistema de archivos está organizado como un árbol de directorios, se necesita cierta forma de especificar los nombres de los archivos. Por lo general se utilizan dos métodos distintos. En el primer método, cada archivo recibe un **nombre de ruta absoluto** que consiste en la ruta desde el directorio raíz al archivo. Por ejemplo, en las siguientes rutas el directorio raíz contiene un subdirectorio llamado *usr* que a su vez contiene un subdirectorio *ast*, el cual contiene el archivo *mailbox* [2].

Windows: `\usr\ast\mailbox`

UNIX: `/usr/ast/mailbox`

MULTICS: `>usr>ast>mailbox`

El otro tipo de nombre es el **nombre de ruta relativa**. Éste se utiliza en conjunto con el concepto del directorio de trabajo (también llamado directorio actual). Un usuario puede designar un directorio como el directorio de trabajo actual, en cuyo caso todos los nombres de las rutas que no empiecen en el directorio raíz se toman en **forma relativa** al directorio de trabajo. Por ejemplo, si el directorio de trabajo actual es `/usr/ast`, entonces el archivo cuya ruta absoluta sea `/usr/ast/mailbox` se puede referenciar simplemente como `mailbox`[2].

Organización física.

Los datos son arreglados por su adyacencia física, es decir, de acuerdo con el dispositivo de almacenamiento secundario. Los registros son de tamaño fijo o de tamaño variable y pueden organizarse de varias formas para constituir archivos físicos.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	131/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

La organización física de un archivo en el almacenamiento secundario depende de la estrategia de agrupación y de la estrategia de asignación de archivos.

En esta guía no se abordará a profundidad la organización física, se deja a revisión en clase teórica.

Desarrollo

Para poder realizar actividades en esta parte práctica solo se tratarán algunos aspectos sobre los archivos, como las operaciones, el tipo de acceso y organización lógica.

Actividad 1: Leer e implementar los ejemplos que se van describiendo para después poder realizar la actividad 2 propuesta por el profesor.

Antes de poder realizar cualquier operación de lectura/escritura en Python hay que realizar la apertura del archivo, esto se realiza con la función **open()** indicando su ubicación y nombre seguido, opcionalmente, por el modo o tipo de operación a realizar y la codificación que tendrá el archivo. Si no se indica el tipo de operación el archivo se abrirá en modo de lectura y si se omite la codificación se utilizará la codificación actual del sistema. Si no existe la ruta del archivo o se intenta abrir para lectura un archivo inexistente se producirá una excepción del tipo *IOError*.

Uno ejemplos del uso de la función son.

```
archivo=open('C:/Users/pc/Documents/prueba.txt', mode='r', encoding='utf-8')
```

```
archivo=open('C:/Users/pc/Documents/prueba.txt', 'r')
```

Para revisar el formato de codificación del sistema se puede utilizar las siguientes líneas:

```
import locale
print(locale.getpreferredencoding())
```

Una vez que el archivo ya no se utilizará debe cerrarse, eso se consigue con la siguiente línea:

```
archivo.close
```

Entonces una estructura general que considere la apertura, el fallo de apertura y el cierre es la siguiente:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	132/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

try:
    a=None
    a=open("C:/Users/alumno/Desktop/Practica 10/prueba.txt",'r')
except:
    print('Error al abrir')
finally:
    if a is not None:
        print('Abrio correctamente')
        a.close()

```

En este segmento de código se utiliza una estructura *try* para la captura de las excepciones.

Implícitamente en Python si el archivo no existe, para crearlo hay que indicar en el modo o tipo de operación, que lo haga. En la siguiente tabla se muestran los diferentes modos de apertura que se tienen en Python.

r	Modo de apertura	Ubicación del puntero
r	Solo lectura	Al inicio del archivo
rb	Solo lectura en modo binario	Al inicio del archivo
r+	Lectura y escritura	Al inicio del archivo
rb+	Lectura y escritura en modo binario	Al inicio del archivo
w	Solo escritura. Sobrescribe el archivo si existe. Crea el archivo si no existe	Al inicio del archivo
wb	Solo escritura en modo binario. Sobrescribe el archivo si existe. Crea el archivo si no existe	Al inicio del archivo
w+	Escritura y lectura. Sobrescribe el archivo si existe. Crea el archivo si no existe	Al inicio del archivo
wb+	Escritura y lectura en modo binario. Sobrescribe el archivo si existe. Crea el archivo si no existe	Al inicio del archivo
a	Añadido (agregar contenido). Crea el archivo si éste no existe	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo
ab	Añadido en modo binario (agregar contenido). Crea el archivo si éste no existe	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo
a+	Añadido (agregar contenido) y lectura. Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo
ab+	Añadido (agregar contenido) y lectura en modo binario. Crea el archivo si éste no existe	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo

Ahora para realizar la operación de lectura a un archivo en Python se tienen diferentes funciones, a continuación, se menciona y se muestra un ejemplo de uso de cada una.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	133/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Con el método **read()** es posible leer un número de bytes determinados. Si no se indica número se leerá todo lo que reste o si se alcanzó el final de archivo devolverá una cadena vacía.

Ejemplo:

```
archivo=open('C:/Users/pc/Documents/prueba.txt', 'r')
c1=a.read(3)
c2=a.read()
print(c1)
#print(c2)
a.close()
```

El método **readline()** lee de un archivo una línea completa. Ejemplo:

```
a3=open('C:/Users/pc/Documents/prueba.txt', 'r')
while True:
    linea=a3.readline()
    if not linea:
        break
    print(linea)
a3.close()
```

El método **readlines()** lee todas las líneas de un archivo como una lista. Si se indica el parámetro de tamaño leerá esa cantidad de bytes del archivo y lo necesario hasta completar la última línea. Ejemplo:

```
archivo=open('C:/Users/pc/Documents/prueba.txt', 'r')
lista = archivo.readlines() #lee todas las lieneas a una lista
numlin = 0
for linea in lista:
    numlin += 1
    print(numlin, linea)
archivo.close()
```

La sentencia **with-as** permite usar los archivos de forma óptima cerrándolos y liberando la memoria al concluir el proceso de lectura.

```
with open('C:/Users/pc/Documents/prueba.txt', 'r') as archivo:
    for linea in archivo: # recorre línea a línea el archivo
        print(linea)
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	134/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Para escribir en un archivo se utilizan los métodos `write()` y `writelines()`. El método `write()` escribe una cadena y el método `writelines()` escribe una lista a un archivo. Si en el momento de escribir el archivo no existe se creará uno nuevo. Ejemplo:

```
cadena1 = 'Datos' # declara cadena1
cadena2 = 'Secretos' # declara cadena2
archivo2 = open('C:/Users/pc/Documents/datos2.txt', 'w') # abre archivo para escribir
print(cadena1 + '\n')
archivo2.write(cadena1 + '\n') # escribe cadena1 añadiendo salto de línea
archivo2.write("hola") # escribe la cadena hola
archivo2.write(cadena2) # escribe cadena2 en archivo
archivo2.close # cierra archivo

lista = ['lunes', 'martes', 'miercoles', 'jueves', 'viernes'] # declara lists
archivo2 = open('datos2.txt', 'w') # abre archivo en modo escritura
archivo2.writelines(lista) # escribe toda la lista en el archivo
archivo2.close # cierra archivo
```

Para mover el apuntador de archivo se usa el método `seek()` que desplaza el puntero a una posición del archivo. También es posible conocer información sobre su posición para lo que se usa el método `tell()`, que devuelve la posición del puntero en un momento dado (en bytes). Ejemplo:

```
archivo=open('C:/Users/pc/Documents/prueba.txt', 'r')
archivo.seek(5) # mueve puntero al quinto byte
cadena1 = archivo.read(5) # lee los siguientes 5 bytes
print(cadena1) # muestra cadena
print(archivo.tell()) # muestra posición del puntero
archivo.close # cierra archivo
```

Para leer y escribir cualquier tipo de objeto Python se importa el módulo `pickle` y se usan sus métodos `dump()` y `load()` para leer y escribir los datos. Ejemplo:

```
import pickle # importa módulo pickle
lista = ['algoritmos 1', 'algoritmos 2', 'estructuras'] # declara lista
archivo = open('materias.dat', 'wb') # abre archivo binario para escribir
pickle.dump(lista, archivo) # escribe lista en archivo
archivo.close # cierra archivo
del lista # borra de memoria la lista
archivo = open('materias.dat', 'rb') # abre archivo binario para leer
lista = pickle.load(archivo) # carga lista desde archivo
print(lista) # muestra lista
archivo.close # cierra archivo
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	135/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

También es posible crear directorios y recorrerlos desde Python. Ejemplo:

```
#crear un directorio
import os
def crear_directorio(ruta):
    try:
        os.makedirs(ruta)
    except OSError:
        pass
    os.chdir(ruta)
crear_directorio('C:/Users/pc/Documents/nuevo')
```

```
#recorrer un directorio
import os
rootDir = 'C:/Users/pc/Documents'
for dirName, subdirList, fileList in os.walk(rootDir):
    print('Directorio encontrado: %s' % dirName)
    for fname in fileList:
        print('\t%s' % fname)
```

Finalmente, algo importante es que se pueden conocer algunos de los atributos de un archivo. Ejemplo:

```
archivo=open('C:/Users/pc/Documents/prueba.txt', 'r')
contenido = archivo.read()
nombre = archivo.name
modo = archivo.mode
encoding = archivo.encoding
archivo.close()

if archivo.closed:
    print ("El archivo se ha cerrado correctamente")
else:
    print ("El archivo permanece abierto")
print (contenido)
print (nombre)
print (modo)
print (encoding)
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	136/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Actividad 2

Ejercicios sugeridos por el profesor

Referencias

[1] Abraham Silberschatz, Peter Baer Galvin & Greg Gagne

Fundamentos de Sistemas Operativos

Séptima Edición

MacGrall Hill

[2] Andrew S. Tanenbaum

Sistemas Operativos Modernos

Tercera Edición

[4] <https://www.python.org/>

[3] <http://python-para-impacientes.blogspot.mx/2014/02/operaciones-con-archivos.html>

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	137/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 11

Introducción a OpenMP

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	138/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 11

Estructura de datos y Algoritmos II

Introducción a OpenMP

Objetivo: El estudiante conocerá y aprenderá a utilizar extensiones de un lenguaje o algún lenguaje de programación paralela que le permita llevar a la programación algunos algoritmos paralelos.

Actividades

- Revisar el ambiente necesario para trabajar con OpenMP y el lenguaje C.
- Realizar ejemplos del funcionamiento de las primeras directivas de OpenMP en el lenguaje C
- Realizar un primer programa que se resuelva un problema de forma paralela utilizando las primeras directivas de *OpenMP*.

Antecedentes

- Análisis previo del concepto de proceso e hilo visto en clase teórica.
- Conocimientos sólidos de programación en Lenguaje C.

Introducción

Conceptos básicos en el procesamiento paralelo.

La entidad software principal en un sistema de cómputo es **el proceso**, y su contraparte en hardware es **el procesador** o bien la unidad de procesamiento. En la actualidad tanto los sistemas de cómputo como los dispositivos de comunicación cuentan con varias unidades de procesamiento, lo que permite que los procesos se distribuyan en éstas para agilizar la funcionalidad de dichos sistemas/dispositivos. Esta distribución la puede hacer el sistema operativo o bien los desarrolladores de sistemas de software. Por lo tanto, es necesario entender lo que es un proceso, sus características y sus variantes.

El proceso.

Un proceso tiene varias definiciones, por mencionar algunas: a) programa en ejecución, b) procedimiento al que se le asigna el procesador y c) centro de control de un procedimiento. Por lo que no solamente un proceso se limita a ser un programa en ejecución. Por ejemplo, al ejecutar un programa éste puede generar varios procesos

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	139/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

que no necesariamente están incluidos en un programa o bien partes del código de un mismo programa son ejecutados por diversos procesos.

Los procesos se conforman básicamente de las siguientes partes.

- El código del programa o segmento de código (sección de texto)
- La actividad actual representada por el contador de programa (PC) y por los contenidos de los registros del procesador.
- Una pila o *stack* que contiene datos temporales, como los parámetros de las funciones, las direcciones de retorno y variables locales. También el puntero de pila o stack pointer.
- Una sección de datos que contiene variables globales.
- Un *heap* o cúmulo de memoria, que es asignada dinámicamente al proceso en tiempo de ejecución.

Como se muestra en la figura 11.1

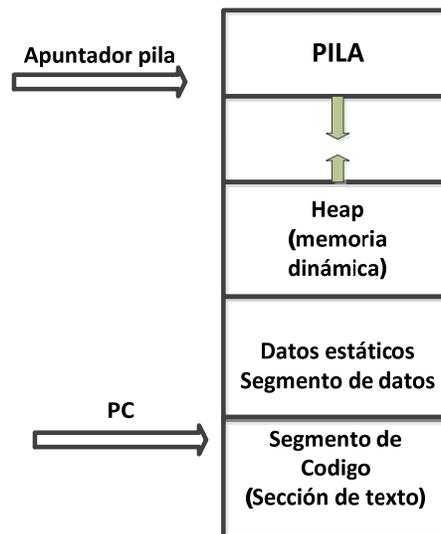


Figura 11.1

Respecto a los estados de un proceso, se puede encontrar en cualquiera de los siguientes: Listo, en ejecución y bloqueado. Los procesos en el estado listo son los que pueden pasar a estado de ejecución si el planificador los selecciona. Los procesos en el estado ejecución son los que se están ejecutando en el procesador en ese momento dado. Los procesos que se encuentran en estado bloqueado están esperando la respuesta de algún otro proceso para poder continuar con su ejecución (por ejemplo, realizar una operación de E/S).

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	140/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Hilos

Un proceso puede ser un programa que tiene solo un hilo de ejecución. Por ejemplo, cuando un proceso está ejecutando un procesador de texto, se ejecuta solo un hilo de instrucciones. Este único hilo de control permite al proceso realizar una tarea a la vez, por ejemplo, el usuario no puede escribir simultáneamente caracteres y pasar al corrector ortográfico dentro del mismo proceso, para hacerlo se necesita trabajar con varios hilos en ejecución, para llevar a cabo más de una tarea de forma simultánea [1]. Entonces un proceso tradicional (o proceso pesado) tiene un solo hilo de control que realiza una sola tarea y un proceso con varios hilos de control, puede realizar más de una tarea a la vez. Así también se pueden tener varios procesos y cada uno con varios hilos. Figura 11.2.

Un proceso es un hilo principal que puede crear hilos y cada hilo puede ejecutar concurrentemente varias secuencias de instrucciones asociadas a funciones dentro del mismo proceso principal. Si el hilo principal termina entonces los hilos creados por este salen también de ejecución.

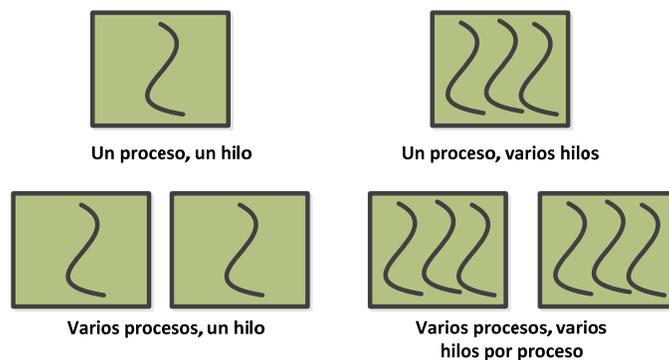


Figura 11.2

Todos los hilos dentro de un proceso comparten la misma imagen de memoria, como el segmento de código, el segmento de datos y recursos del sistema operativo. Pero no comparte el contador de programa (por lo que cada hilo podrá ejecutar una sección distinta de código), la pila en la que se crean las variables locales de las funciones llamadas por el hilo, así como su estado. Figura 11.3.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	141/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

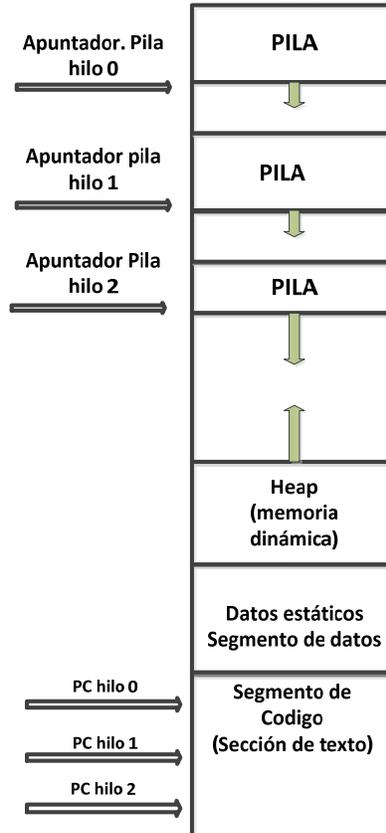


Figura 11.3

Concurrencia vs Paralelismo

Concurrencia: Es la existencia de varias actividades realizándose simultáneamente y requieren de una sincronización para trabajar en conjunto. Se dice que dos procesos o hilos son concurrentes cuando están en progreso simultáneamente, pero no al mismo tiempo.

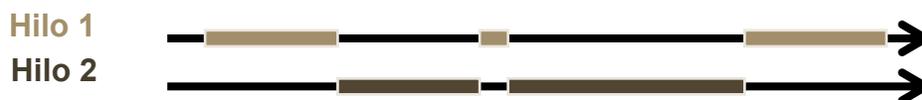


Figura 11.4

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	142/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Paralelismo: Actividades que se pueden realizar al mismo tiempo. Dos procesos o hilos son paralelos cuando se están ejecutando al mismo tiempo, para lo cual se requiere que existan dos unidades de procesamiento.

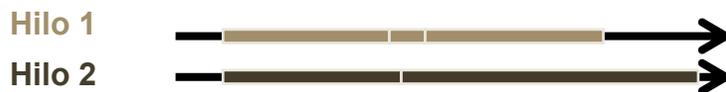


Figura 11.5

Programación secuencial, concurrente y paralela

Cuando se realiza un programa secuencial, las acciones o instrucciones se realizan una tras otra, en cambio en un programa concurrente se tendrán actividades que se pueden realizar de forma simultánea y se ejecutarán en un solo elemento de procesamiento.

En un programa paralelo se tienen acciones que se pueden realizar también de forma simultánea, pero se pueden ejecutar en forma independiente por diferentes unidades de procesamiento.

Por lo anterior, para tener un programa paralelo se requiere de una computadora paralela, es decir una computadora que tenga dos o más unidades de procesamiento.

Memoria Compartida en una computadora

En el hardware de una computadora, la memoria compartida se refiere a un bloque de memoria de acceso aleatorio a la que se puede acceder por varias unidades de procesamiento diferentes. Tal es el caso de las computadoras *Multicore* donde se tienen varios núcleos que comparten la misma memoria principal. Figura 11.6.

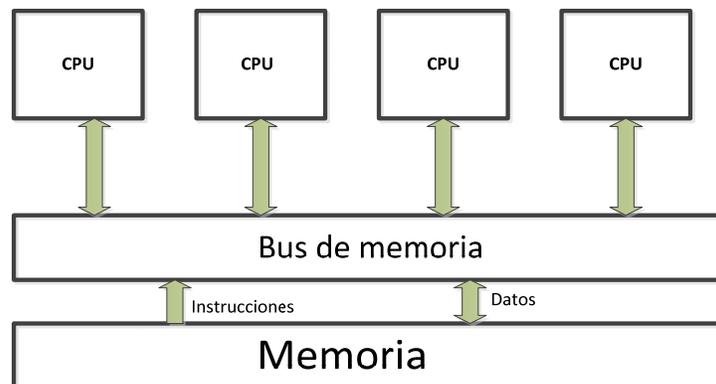


Figura 11.6

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	143/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En software, la memoria compartida es un método de comunicación entre procesos/hilos, es decir, una manera de intercambiar datos entre programas o instrucciones que se ejecutan al mismo tiempo. Un proceso/hilo creará un espacio en la memoria RAM a la que otros procesos pueden tener acceso.

OpenMP (Open Multi-Processing)

Es un interfaz de programación de aplicaciones (API) multiproceso portable, para computadoras paralelas que tiene una arquitectura de memoria compartida.

OpenMp está formado

- Un conjunto de directivas del compilador, las cuales son ordenes abreviadas que instruyen al compilador para insertar ordenes en el código fuente y realizar una acción en particular.
- una biblioteca de funciones y
- variables de entorno.

que se utilizan para paralelizar programas escritos es lenguaje C, C++ y Fortran.

Entonces para utilizar OpenMP basta con contar con un compilador que incluya estas extensiones al lenguaje. En [2] se pueden encontrar la lista de compiladores que implementa este API.

Arquitectura de OpenMP

Para paralelizar un programa se tiene que hacer de forma explícita, es decir, el programador debe analizar e identificar qué partes del problema o programa se pueden realizar de forma concurrente y por tanto se pueda utilizar un conjunto de hilos que ayuden a resolver el problema.

OpenMp trabaja con la llamada arquitectura fork-join, donde a partir del proceso o hilo principal se genera un número de hilos que se utilizarán para la solución en paralelo llamada región paralela y después se unirán para volver a tener solo el hilo o proceso principal. El programador especifica en qué partes del programa se requiere ese número de hilos. De aquí que se diga que OpenMP combina código serial y paralelo. Figura 11.7

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	144/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

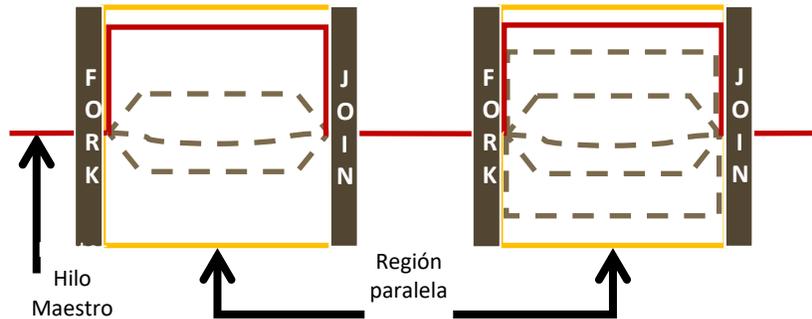


Figura 11.7

Directivas o pragmas

Agregar una directiva o *pragma* en el código es colocar una línea como la que sigue

`#pragma omp nombreDelConstructor <clausula o clausulas>`

Donde como se puede observar se tiene los llamados constructores y las cláusulas. Los constructores es el nombre de la directiva que se agrega y las cláusulas son atributos dados a algunos constructores para un fin específico; una cláusula nunca se coloca si no hay antes un constructor.

En la parte del desarrollo de esta guía se irán explicando constructores, cláusulas y funciones de la biblioteca omp.h. mediante actividades que facilitan la comprensión

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	145/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Desarrollo

Para aprender más sobre openMP en lo siguiente se desarrollan actividades que facilitaran la comprensión.

Para el desarrollo de la parte práctica, en esta guía se utilizará el compilador gcc de Linux. Por lo que para realizar las siguientes actividades es necesario estar en un ambiente Linux, contar con un editor (vi, emacs, gedit, nano, etc..) y tener el compilador gcc (versión mayor o igual a la 4.2).

El primer constructor por revisar es el más importante, *parallel*, el cual permite crear regiones paralelas, es decir generar un número de hilos que ejecutarán ciertas instrucciones y cuando terminen su actividad se tendrá solo al maestro. Figura 11.8.

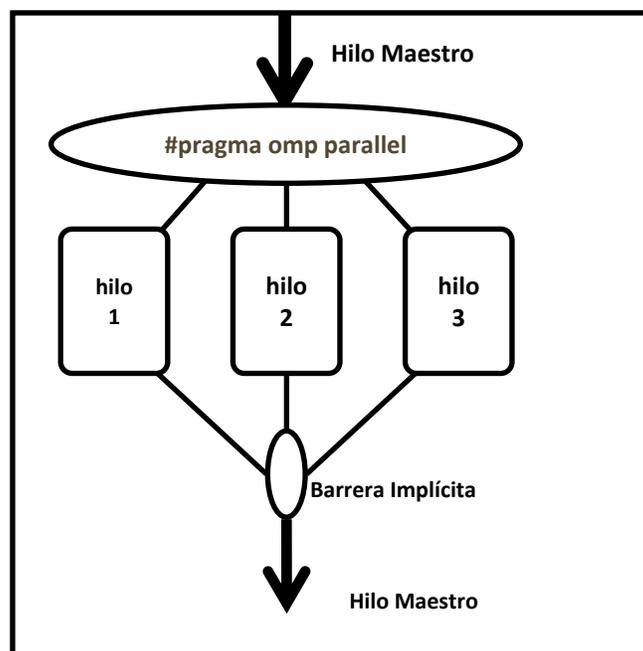


Figura 11.8

La sintaxis es como sigue:

```
#pragma omp parallel
{
//Bloque de código
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	146/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Nota: La llave de inicio de la región debe ir en el renglón siguiente de la directiva, si no se coloca así ocurrirá error.

Actividad 1

En un editor de texto teclear el siguiente código y guardarlo con extensión “.c” , por ejemplo *hola.c* .

```
#include <stdio.h>

int main() {
    int i;
    printf("Hola Mundo\n");
    for(i=0;i<10;i++)
        printf("Iteración:%d\n",i);
    printf("Adiós \n");
    return 0;
}
```

Después desde la consola o línea de comandos de Linux, situarse en el directorio donde se encuentra el archivo fuente y compilarlo de la siguiente manera.

```
gcc hola.c -o hola
```

Para ejecutarlo, igual en la línea de comandos escribir:

```
./hola
```

Una vez que ya se tiene el código en su versión serial, se formará una región paralela. Entonces, agregar al código anterior el constructor **parallel** desde la declaración de la variable y hasta antes de la última impresión a pantalla, como se muestra en el siguiente código.

```
#include <stdio.h>

int main() {
    #pragma omp parallel
    {
        int i;
        printf("Hola Mundo\n");
        for(i=0; i<10;i++)
            printf("Iteración:%d\n",i);
    }
    printf("Adiós \n");
    return 0;
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	147/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guardar el archivo, compilarlo y ejecutarlo.

Para compilarlo hay que indicar que se agregaran las directivas de *openMP*, lo que se realiza agregando la bandera en la compilación *-fopenmp*. Ejemplo.

gcc -fopenmp hola.c -o hola

Para ejecutarlo, se realiza de la misma manera que el secuencial.

¿Qué diferencia hay en la salida del programa con respecto a la secuencial?

¿Por qué se obtiene esa salida?

Actividad 2

En cada región paralela hay un número de hilos generados por defecto y ese número es igual al de unidades de procesamiento que se tengan en la computadora paralela que se esté utilizando, en este caso el número de núcleos que tenga el procesador.

En el mismo código, cambiar el número de hilos que habrá en la región paralela a un número diferente *n* (entero), probar cada una de las formas indicadas a continuación. Primero modificar, después compilar y ejecutar nuevamente el programa en cada cambio.

1-Modificar la variable de ambiente *OMP_NUM_THREADS* desde la consola, de la siguiente forma:

```
export OMP_NUM_THREADS=4
```

2- Cambiar el número de hilos a *n* (un entero llamando a la función *omp_set_num_threads(n)* que se encuentra en la biblioteca *omp.h* (hay que incluirla).

3-Agregar la cláusula *num_threads(n)* seguida después del constructor *parallel*, esto es:

```
#pragma omp parallel num_threads (n)
```

¿Qué sucedió en la ejecución con respecto al de la actividad 1?

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	148/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Actividad 3

En la programación paralela en computadoras que tienen memoria compartida puede presentarse la llamada condición de carrera (*race condition*) que ocurre cuando varios hilos tienen acceso a recursos compartidos **sin control**. El caso más común se da cuando en un programa varios hilos tienen acceso concurrente a una misma dirección de memoria (variable) y todos o algunos en algún momento intentan escribir en la misma localidad al mismo tiempo. Esto es un conflicto que genera salidas incorrectas o impredecibles del programa.

En *OpenMP* al trabajar con hilos se sabe que hay partes de la memoria que comparten entre ellos y otras no. Por lo que habrá variables que serán compartidas entre los hilos, (a las cuales todos los hilos tienen acceso y las pueden modificar) y habrá otras que serán propias o privadas de cada uno.

Dentro del código se dirá que cualquier variable que esté declarada fuera de la región paralela será compartida y cualquier variable declarada dentro de la región paralela será privada.

Del código que se está trabajando, sacar de la región paralela la declaración de la variable entera *i*, compilar y ejecutar el programa varias veces.

```
#include <stdio.h>

int main() {
    int i;
    #pragma omp parallel
    {
        printf("Hola Mundo\n");
        for(i=0;i<10;i++)
            printf("Iteración:%d\n",i);
    }
    printf("Adiós \n");
    return 0;
}
```

¿Qué sucedió? Y ¿Por qué? _____

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	149/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Actividad 4

Existen dos cláusulas que pueden forzar a que una variable privada sea compartida y una compartida sea privada y son las siguientes:

shared(): Las variables colocadas separadas por coma dentro del paréntesis serán compartidas entre todos los hilos de la región paralela. Sólo existe una copia, y todos los hilos acceden y modifican dicha copia.

private(): Las variables colocadas separadas por coma dentro del paréntesis serán privadas. Se crean p copias, una por hilo, las cuales no se inicializan y no tienen un valor definido al final de la región paralela ya que se destruyen al finalizar la ejecución de los hilos.

Al código resultante de la actividad 3, agregar la cláusula *private()* después del constructor *parallel* y colocar la variable i :

```
#pragma omp parallel private(i)
{
}
```

¿Que sucedió? _____

Actividad 5

Dentro de una región paralela hay un número de hilos generados y cada uno tiene asignado un identificador. Estos datos se pueden conocer durante la ejecución con la llamada a las funciones de la biblioteca *omp_get_num_threads()* y *omp_get_thread_num()* respectivamente.

Probar el siguiente ejemplo, y notar que para su buen funcionamiento se debe indicar que la variable *tid* sea privada dentro de la región paralela, ya que de no ser así todos los hilos escribirán en la dirección de memoria asignada a dicha variable sin un control (*race condition*), es decir “competirán” para ver quién llega antes y el resultado visualizado puede ser inconsistente e impredecible.

Ejemplo:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	150/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

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

int main(){
    int tid,nth;
    #pragma omp parallel private(tid)
    {

        tid = omp_get_thread_num();
        nth = omp_get_num_threads();
        printf("Hola Mundo desde el hilo %d de un total de %d\n",tid,nth);
    }
    printf("Adios");
    return 0;
}
```

Probar el ejemplo quitando del código la cláusula *private* para visualizar el comportamiento del programa.

¿Qué sucedió? _____

Actividad 6

Se requiere realizar la suma de dos arreglos unidimensionales de 10 elementos de forma paralela utilizando solo dos hilos. Para ello se utilizará un paralelismo de datos o descomposición de dominio, es decir, cada hilo trabajará con diferentes elementos de los arreglos a sumar *A* y *B*, pero ambos utilizarán el mismo algoritmo para realizar la suma. Figura.11.9.

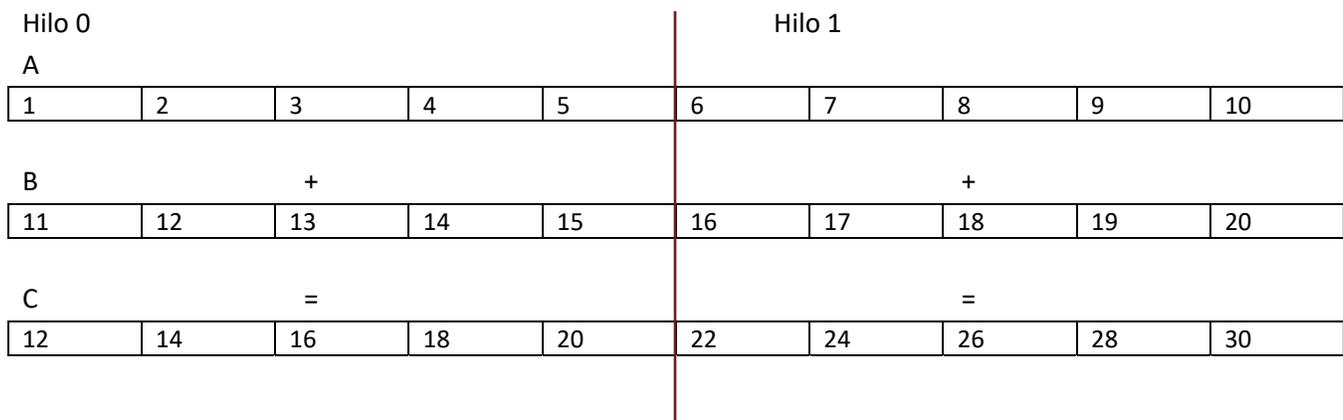


Figura 11.9

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	151/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Se parte de la versión serial, donde la suma se realiza de la siguiente manera (se asume que A y B ya tienen valores para ser sumados):

```
for(i=0; i<10; i++)
    C[i]=A[i]+B[i]
```

Actividad 6.1

Realizar programa en su versión serial.

```
#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#define n 10

void llenaArreglo(int *a);
void suma(int *a,int *b,int *c);

main(){

    int max,*a,*b,*c;

    a=(int *)malloc(sizeof(int)*n);
    b=(int *)malloc(sizeof(int)*n);
    c=(int *)malloc(sizeof(int)*n);

    llenaArreglo(a);
    llenaArreglo(b);

    suma(a,b,c);
}

void llenaArreglo(int *a){
    int i;
    for(i=0;i<n;i++)
    {
        a[i]=rand()%n;
        printf("%d\t", a[i]);
    }
    printf("\n");
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	152/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```
void suma(int *A, int *B, int *C){
    int i;
    for(i=0;i<n;i++){
        C[i]=A[i]+B[i];
        printf("%d\t", C[i]);
    }
}
```

Actividad 6.2

Para la versión paralela, el *hilo 0* sumará la primera mitad de *A* con la primera de *B* y el *hilo 1* sumará la segunda mitad de *A* con la segunda de *B*. Para conseguir esto cada hilo realizará las mismas instrucciones, pero utilizará índices diferentes para referirse a diferentes elementos de los arreglos, entonces cada uno iniciará y terminará el índice *i* en valores diferentes

```
for(i=inicio; i<fin; i++)
    C[i]=A[i]+B[i]
```

Inicio y fin se pueden calcular de la siguiente manera, siendo *tid* el identificador de cada hilo:

```
inicio = tid* 5
fin = (tid+1)*5-1
```

El 5 puede ser sustituido por (número de elementos del arreglo) / (número de hilos), pero el resultado debe ser entero.

Entonces la función donde se realiza la suma queda:

```
void suma(int *A, int *B, int *C){
    int i,tid,inicio,fin;
    omp_set_num_threads(2);
    #pragma omp parallel private(inicio,fin,tid,i)
    {
        tid = omp_get_thread_num();
        inicio = tid* 5;
        fin = (tid+1)*5-1;
        for(i=inicio;i<fin;i++)
        {
            C[i]=A[i]+B[i];
            printf("hilo %d calculo C[%d]= %d\n",tid,i, C[i]);
        }
    }
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	153/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Implementar el código completo.

Actividad 7

Otro constructor es el **for**, el cual divide las iteraciones de una estructura de repetición *for*. Para utilizarlo se debe estar dentro de una región paralela.

Su sintaxis es:

```
#pragma omp parallel
{
...
#pragma omp for
    for(i=0;i<12;i++) {
        Realizar Trabajo();
    }
...
}
```

La variable índice de control *i* se hará privada de forma automática. Esto para que cada hilo trabaje con su propia variable *i*.

Este constructor se utiliza comúnmente en el llamado paralelismo de datos o descomposición de dominio, lo que significa que, cuando en el análisis del algoritmo se detecta que varios hilos pueden trabajar con el mismo algoritmo o instrucciones que se repetirán, pero sobre diferentes datos y no hay dependencias con iteraciones anteriores.

Por lo anterior, **no siempre** es conveniente dividir las iteraciones de un ciclo *for*.

Modificar el código de la actividad 1, de manera que se dividan las iteraciones de la estructura de repetición *for*.

```
#include <stdio.h>
int main() {
    #pragma omp parallel
    {
        printf("Hola Mundo\n");
        #pragma omp for
        for(i=0;i<10;i++)
            printf("Iteración:%d\n",i);
    }
    printf("Adiós \n");
    return 0;
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	154/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Actividad 8

Realizar la suma de dos arreglos unidimensionales de n elementos de forma paralela, utilizando los hilos por defecto que se generen en la región paralela y el constructor *for*.

Como se explicó en la actividad 6 los hilos realizarán las mismas operaciones, pero sobre diferentes elementos del arreglo y eso se consigue cuando cada hilo inicia y termina sus iteraciones en valores diferentes, para referirse a diferentes elementos de los arreglos *A* y *B*. Esto lo hace el constructor *for*, ya que al dividir las iteraciones cada hilo trabaja con diferentes valores del índice de control.

Entonces la solución queda:

```
void suma(int *A, int *B, int *C){
    int i,tid;
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        #pragma omp for
        for(i=0;i<n;i++){
            C[i]=A[i]+B[i];
            printf("hilo %d calculo C[%d]= %d\n",tid,i, C[i]);
        }
    }
}
```

Actividad 9

Ejercicios sugeridos por el profesor

En las siguientes guías se revisarán otros constructores y cláusulas.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	155/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Referencias

[1] Abraham Silberschatz, Peter Baer Galvin & Greg Gagne

Fundamentos de Sistemas Operativos

Séptima Edición

MacGrall Hill

[2] <http://openmp.org/wp/>

[3] Introduction to parallel programming, Intel Software College, junio 2007

[4] B. Chapman ,Using OpenMP, The MIT Press, 2008

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	156/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 12

Algoritmos Paralelos parte 1.

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	157/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 12

Estructura de datos y Algoritmos II

Algoritmos paralelos.

Objetivo: El estudiante conocerá y aprenderá a utilizar algunos patrones paralelos para el acceso a datos y distribución de tareas que le servirán en el diseño e implementación de un algoritmo paralelo en computadoras con arquitectura de memoria compartida.

Actividades

- Realizar ejemplos del funcionamiento de directivas de OpenMP en el lenguaje C, tales como los constructores *critical*, *section*, *single*, *master*, *barrier* y las cláusulas *reduction* y *nowait* que permitirán implementar algún patrón paralelo.
- Realizar programas que resuelvan un problema de forma paralela utilizando distintas directivas de OpenMP.

Antecedentes

- Guía de estudio práctica 11.
- Conocimientos sólidos de programación en Lenguaje C.

Introducción.

En esta guía se irán revisando otros constructores y cláusulas de OpenMP así como diversos conceptos y ejemplos que ayudarán a la comprensión de las mismas.

Constructor *critical*

En una aplicación concurrente, una **región crítica** es una porción de código que contienen la actualización de una o más variables compartidas, por lo que puede ocurrir una condición de carrera.

La **exclusión mutua** consiste en que un solo proceso/hilo excluye temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema.

En OpenMP existe una directiva que permite que un segmento de código que contiene una secuencia de instrucciones no sea interrumpido por otros hilos (realiza una exclusión mutua). Es decir, que al segmento de

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	158/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

código delimitado por la directiva solo pueda entrar un hilo a la vez y así evitar una condición de carrera, el nombre de esta directiva es ***critical*** y la sintaxis de uso es:

```
#pragma omp critical
{
}
}
```

La siguiente función permite encontrar el máximo valor entero de entre los elementos de un arreglo unidimensional de n elementos enteros. Y se requiere realizar su versión paralela.

```
int buscaMaximo(int *a, int n){
    int max,i;
    max=a[0];
    for(i=1;i<n;i++) {
        if(a[i]>max)
            max=a[i];
    }
    return max;
}
```

Analizando, se puede realizar una descomposición de dominio o datos, en otras palabras, si se tienen varios hilos, cada uno puede buscar el máximo en un sub-arreglo asignado del arreglo original y utilizar el mismo algoritmo de búsqueda sobre cada sub-arreglo. Figura 12.1.

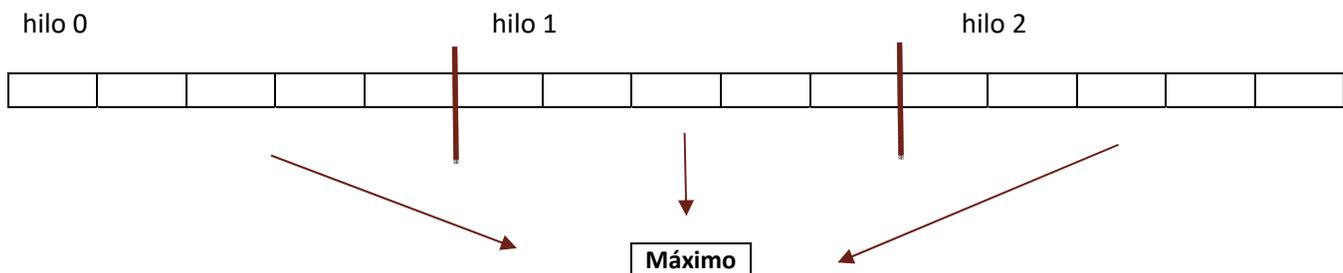


Figura 12.1

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	159/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Utilizando la función del ejemplo que realiza la búsqueda, para dividir el arreglo entre los hilos cada uno debe empezar y terminar su índice del arreglo en diferente valor. Para conseguir esto con OpenMP (como se vio en la práctica anterior) se puede utilizar el constructor **for** ya que este divide las iteraciones del ciclo entre los hilos. La función queda:

```
int buscaMaximo(int *a, int n){
    int max,i;
    max=a[0];
    #pragma omp parallel
    {
        #pragma omp for
        for(i=1;i<n;i++){
            if(a[i]>max)
                max=a[i];
        }
    }
    return max;
}
```

NOTA: Cuando lo que está dentro de la región paralela solo es una estructura for y esta es posible paralelizarla, se pueden anidar los constructores **parallel** y **for**, entonces el código de la función anterior se reescribe como:

```
int buscaMaximo(int *a, int n){
    int max,i;
    max=a[0];
    #pragma omp parallel for
    for(i=1;i<n;i++){
        if(a[i]>max)
            max=a[i];
    }
    return max;
}
```

Agregando el constructor **for** cada hilo trabaja con diferentes partes del arreglo, pero, cada uno revisa si **a[i]>max** y si por lo menos dos de ellos encuentran la proposición verdadera, actualizan la variable **max** donde se almacena el valor máximo encontrado, y entonces ocurrirá una condición de carrera. Una forma de arreglar este problema es que un hilo a la vez modifique la variable **max** y los demás esperen su turno, esto se consigue con el constructor **critical**.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	160/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```
int buscaMaximo(int *a, int n){

    int max,i;
    max=a[0];
    #pragma omp parallel for
    for(i=1;i<n;i++){
        if(a[i]>max){
            #pragma omp critical
            {
                max=a[i];
            }
        }
    }
}
```

Hasta aquí todavía existe un problema ya que, aunque cada hilo espera su turno en actualizar **max**, el valor de **max** que utiliza uno de los hilos en espera puede que sea menor al valor de $a[i]$ que analiza en ese momento debido a que fue actualizado por uno de los hilos que anteriormente entró a modificarlo. Entonces para que no exista ese problema, cuando cada hilo entra a actualizar a la variable **max** debe revisar nuevamente si $a[i]>max$.

Y finalmente el código queda:

```
int buscaMaximo(int *a, int n){
    int max,i;
    max=a[0];
    #pragma omp parallel for
    for(i=1;i<n;i++){
        if(a[i]>max) {
            #pragma omp critical
            {
                if(a[i]>max)
                    max=a[i];
            }
        }
    }
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	161/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Clausula *reduction*

Para explicar esta cláusula, se partirá del ejemplo de realizar el producto punto entre dos vectores de n elementos. El cual se realiza como se muestra en la figura 12.2.

$$A \cdot B = [A_1 \quad A_2 \quad \dots \quad A_n] \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = A_1 B_1 + A_2 B_2 + \dots + A_n B_n$$

Figura 12.2

Una función para la solución es la siguiente:

```
double prodpunto(double *a, double *b, int n){
    double res=0;
    int i;

    for(i=0;i<n;i++){
        res+=a[i]*b[i];
    }
    return res;
}
```

Si se quiere una solución concurrente/paralela utilizando OpenMP, es decir que n hilos cooperen en la solución, se analiza primero como dividir las tareas entre los n hilos. La solución es similar al ejemplo anterior, que cada hilo trabaje con diferentes elementos de los vectores A y B y cada uno obtenga resultados parciales. Por ejemplo, si los vectores son de 15 elementos y se tienen 3 hilos, el producto punto se puede dividir como:

$$A \cdot B = A_0 \cdot B_0 + A_1 \cdot B_1 + A_2 \cdot B_3$$

Y cada hilo hace los siguientes cálculos:

$$A_1 \cdot B_1 = a_0 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 + a_4 \cdot b_4$$

$$A_2 \cdot B_2 = a_5 \cdot b_5 + a_6 \cdot b_6 + a_7 \cdot b_7 + a_8 \cdot b_8 + a_9 \cdot b_9$$

$$A_3 \cdot B_3 = a_{10} \cdot b_{10} + a_{11} \cdot b_{11} + a_{12} \cdot b_{12} + a_{13} \cdot b_{13} + a_{14} \cdot b_{14}$$

Se observa que cada hilo realiza los mismos cálculos, pero sobre diferentes elementos del vector. Así que para asignar diferentes elementos del vector a cada hilo se puede utilizar el constructor **for**.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	162/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```
double prodpunto(double *a, double *b, int n){
    double res=0;
    int i;
    #pragma omp parallel for
    for(i=0;i<n;i++){
        res+=a[i]*b[i];
    }
    return res;
}
```

Pero todavía no se tiene la solución, porque, todos acumulan los resultados de sus sumas en una variable **res** que es compartida por lo que se sobre escriben los resultados parciales. Para una posible solución lo que se hará es utilizar un arreglo **resp[]** del tamaño del número de hilos que se quiere tener en la región paralela y en cada elemento guardar las soluciones parciales de cada hilo. Después un solo hilo suma esos resultados parciales y obtener la solución final. La solución queda:

```
double prodpunto(double *a ,double *b, int n){
    double res=0,resp[n_hilos];
    int i, tid, nth;

    #pragma omp parallel private (tid) nthreads(n_hilos)
    {
        tid = omp_get_thread_num();
        resp[tid]=0;

        #pragma omp for
        for(i=0;i<n;i++){
            resp[tid]+=a[i]*b[i];
        }

        if(tid==0){
            nth = omp_get_num_threads();
            for(i=0;i<nth;i++){
                res+= resp[i];
            }
        }
        return res;
    }
}
```

Ahora, en lugar de utilizar un arreglo para almacenar resultados parciales se puede utilizar la cláusula **reduction** . Lo que hace la cláusula **reduction** es tomar el valor de una variable aportada por cada hilo y aplicar la operación indicada sobre esos datos para obtener un resultado . Así en el ejemplo del producto punto cada hilo aportaría su cálculo parcial **res** y después se sumarían todos los **res** y el resultado quedaría sobre la misma variable.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	163/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Sintaxis

```
#pragma omp parallel reduction (operador:variable)
{
}

```

La solución del producto punto con la cláusula **reduction** queda:

```
double prodpunto(double *a,double *b, int n){
    double res=0;
    int i;
    #pragma omp parallel for reduction(+:res)
    for(i=0;i<n;i++){
        res+=a[i]*b[i];
    }
    return res;
}

```

Algunos de los operadores utilizados en la cláusula **reduction** son los siguientes:

Operador	Valor inicial
+	0
*	1
-	0
^	0
&	0
	0
&&	1
	0
min y max	

Es importante mencionar que a las actividades realizadas por la cláusula **reduction** se le conoce como operaciones de reducción y son muy utilizadas en la programación paralela.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	164/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Constructor *sections*

Este constructor permite usar paralelismo funcional (*descomposición funcional*) debido a que permite asignar secciones de código independiente a *hilos* diferentes para que trabajen de forma concurrente/paralela. Figura 12.3.

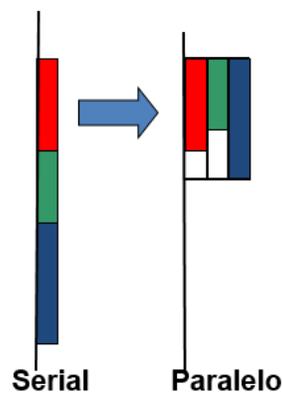


Figura 12.3

Cada sección paralela es ejecutada por un sólo *hilo*, y cada *hilo* ejecuta ninguna o alguna sección. Este constructor tiene una barrera implícita que sincroniza el final de las secciones.

Por ejemplo, para el segmento de código:

```
v = alfa ();
w = beta ();
x = gama (v, w);
y = delta ();
printf ("%6.2f\n", épsilon(x,y));
```

Se observa que se pueden ejecutar en paralelo `alfa()`, `beta()` y `delta()` y por tanto separarlas como sigue:

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	165/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```
#pragma omp parallel sections
{
    #pragma omp section
        v = alfa();
    #pragma omp section
        w = beta();
    #pragma omp section
        y = delta();
}
x = gama(v, w);
printf ("%6.2f\n", epsilon(x,y));
```

Otra posibilidad es que primero se ejecuten `alfa()` y `beta()` y una vez que terminen `gama()` y `delta()` :

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            v = alfa();
        #pragma omp section
            w = beta();
    }
    #pragma omp sections
    {
        #pragma omp section
            x = gama(v, w);
        #pragma omp section
            y = delta();
    }
}
printf ("%6.2f\n", epsilon(x,y));
```

Otros constructores y cláusulas

Constructor *barrier*

¿Cómo obtener la secuencia apropiada cuando hay dependencias presentes? Por ejemplo, cuando un hilo 0 produzca información en alguna variable y otro hilo 1 quiere imprimir esa variable, el hilo 1 debe esperar a que el hilo 0 termine. Para hacerlo se necesita alguna forma de sincronización por ejemplo las barreras.

En OpenMp se tiene el constructor *barrier*, que coloca una barrera explícita para que cada hilo espere hasta que todos lleguen a la barrera. Es una forma de sincronización.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	166/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Ejemplo:

```
#pragma omp parallel shared (A, B, C)
{
    realizaUnTrabajo(A,B);
    printf("Procesado A y B\n");

    #pragma omp barrier // esperan

    realizaUnTrabajo (B,C);
    printf("Procesando B y C\n");
}
```

Constructor *single*

Este constructor permite definir un bloque básico de código dentro de una región paralela, que debe ser ejecutado por un único hilo. Aquí todos los hilos esperan

Por ejemplo, una operación de entrada/salida solo debe realizarse por un solo hilo mientras todos los demás esperan. En el constructor no se especifica qué hilo ejecutará la tarea, puede ser cualquiera de los que están en la región paralela.

Ejemplo:

```
#pragma omp parallel
{
    todosRalizanUnasActividades();
    #pragma omp single
    {
        ActividadE/S();
    } // Hilos esperan
    RealizanMasActividades();
}
```

Constructor *master*

Este constructor es similar a *single* con la diferencia de que las actividades realizadas son hechas por el hilo maestro y no se tiene una barrera implícita, es decir, los hilos restantes no esperan a que el hilo maestro termine la actividad asignada.

Sintaxis

```
#pragma omp master
{
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	167/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Constructor *barrier*

Permite colocar una barrera de forma explícita. Se coloca cuando se requiere que todos los hilos esperen en un punto del programa.

Sintaxis

```
#pragma omp barrier
```

Ejemplo: En el siguiente código primero se genera una región paralela y dentro de esta con el constructor *for* cada hilo asigna valores a diferentes elementos del arreglo *a*, después con el constructor master se indica que solo el hilo maestro imprima el contenido de arreglo. Como el constructor master no tiene barrera implícita se usa el constructor *barrier* para lograr que todos los hilos esperen a que se imprima el arreglo *a* antes de modificarlo.

```
#include <stdio.h>
int main( )
{
    int a[5], i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

        #pragma omp master
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);

        #pragma omp barrier

        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

Cláusula *nowait*

Esta cláusula permite quitar las barreras implícitas que tienen los constructores, como lo son *for*, *single*, *sections*, etc.. Quitar las barreras que no son necesarias ayudan a mejorar el desempeño del programa.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	168/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Desarrollo

Después de haber leído y analizado el contenido de la guía, realizar con ayuda del profesor las siguientes actividades:

Actividad 1

Completar la versión serie y paralela del ejemplo explicado de búsqueda del valor mayor de los elementos de un arreglo unidimensional de enteros.

Actividad 2

Completar la versión serie y sus dos versiones paralelas del ejemplo explicado del producto punto de dos vectores de n elementos enteros.

Actividad 3

Una forma de obtener la aproximación del número irracional PI es utilizar la regla del trapecio para dar solución aproximada a la integral definida

$$\pi = \int_0^1 \frac{4}{(1-x^2)} dx.$$

A continuación, se proporciona el código en su versión serial y se requiere se obtenga su versión paralela.

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

long long num_steps = 100000000;
double step;
double empezar, terminar;

int main(int argc, char* argv[])
{
    double x, pi, sum=0.0;
    int i;
    step = 1.0/(double)num_steps;
    empezar=omp_get_wtime( );

    for (i=0; i<num_steps; i++)
    {
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	169/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

        x = (i + .5)*step;
        sum = sum + 4.0/(1.+ x*x);
    }

    pi = sum*step;
    terminar=omp_get_wtime();

    printf("El valor de PI es %15.12f\n",pi);
    printf("El tiempo de calculo del numero pi es: %lf segundos ",terminar-
empezar);
    return 0;
}

```

Actividad 4

Para el siguiente programa obtener dos versiones paralelas, una utilizando el constructor **section** y otra con el constructor **for**. ¿Cuál de las dos versiones tarda más tiempo?.

```

#include<stdio.h>
#include<omp.h>
#define N 100000

int main (int argc, char *argv[]) {

    double empezar,terminar;
    int i,j;
    float a[N], b[N], c[N],d[N], e[N],f[N];

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    empezar=omp_get_wtime( );

    for(i=0; i<N; i++)
        c[i]=a[i]+b[i];

    for(j=0; j<N; j++)
        d[j]=e[j]+f[j];

    terminar=omp_get_wtime();

    printf("TIEMPO=%lf\n",empezar-terminar);
}

```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	170/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Actividad 5

Probar el siguiente ejemplo visto en la guía y responder a las preguntas:

```
#include <stdio.h>
int main( )
{
    int a[5], i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

        #pragma omp master
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);

        #pragma omp barrier

        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

¿Qué sucede si se quita la barrera? _____

Si en lugar de utilizar el constructor **master** se utilizara **single**, ¿Qué otros cambios se tienen que hacer en el código? _____

Realizar los cambios.

Actividad 6

Ejercicios sugeridos por el profesor

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	171/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Referencias

[1] <http://openmp.org/wp/>

[2] Introduction to parallel programming, Intel Software College, junio 2007

[3] B. Chapman , Using OpenMP, The MIT Press, 2008

[4] R. Chandra et al, Parallel Programming in OpenMP ,Morgan Kaufmann, 2001.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	172/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Guía Práctica de Estudio 13

Algoritmos Paralelos parte 2.

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaña

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	173/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía Práctica 13

Estructura de datos y Algoritmos II

Algoritmos paralelos.

Objetivo: El estudiante utilizará algunos patrones paralelos para resolver algunos problemas de forma paralela y con ello adquiera experiencia en el desarrollo de programas multihilo en sistemas multiprocesador de memoria compartida.

Actividades

- Realizar programas que resuelvan algunos problemas y algoritmos de forma paralela utilizando distintas directivas de OpenMP.

Antecedentes

- Guía de estudio práctica 11 y 12.
- Conocimientos sólidos de programación en Lenguaje C.

Introducción.

En el proceso de análisis de problemas que se deseen paralelizar, surgen diversas opciones o formas de resolverlos. Ello dependerá, en gran medida, de las diferentes técnicas y herramientas que nos pueda proporcionar el software que empleemos en esta tarea. En esta guía revisaremos algunos problemas ejemplo que emplean estas técnicas con las directivas de OpenMP.

Ejemplos de problemas y algoritmos paralelos

Ejemplo 1-Multiplicación de matrices

Los algoritmos basados en arreglos unidimensionales y bidimensionales a menudo son paralelizables de forma simple debido a que es posible tener acceso simultáneamente a todas las partes de la estructura, no se tienen que seguir ligas como en las listas ligadas o árboles.

Un ejemplo común es el producto de dos matrices A y B de orden $n \times m$ y $m \times r$. Recordando la fórmula y la forma de obtener un elemento de C. Figura 13.1.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	174/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

$$C_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

para $0 \leq i, j < n$

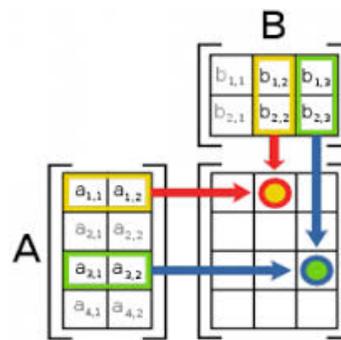


Figura 13.1

Para lograr dividir este problema se realiza un paralelismo de datos, donde cada hilo trabaje con datos distintos, pero con el mismo algoritmo.

Un planteamiento de solución paralela es considerar que, en un ambiente con varios hilos, cada uno puede trabajar con diferentes datos, de acuerdo con las siguientes opciones:

- Calcular un elemento de la matriz resultante, se requiere un renglón de A y una columna de B .
- Calcular todo un renglón de C , se requiere un renglón de A y toda la matriz B .
- Calcular i renglones de C , se requieren i renglones de A y toda B .

Dependiendo de la granularidad escogida se necesitan un número de hilos en la región paralela y como lo ideal es tener un número de hilos igual al número de unidades de procesamiento, que no puede ser muy grande se toma la tercera opción.

Partiendo del algoritmo secuencial, el segmento de código para el cálculo de C es:

```
for (i=0; i<NRA; i++)
{
    for(j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	175/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Donde NRA = número de renglones de A , NCB = número de columnas de B y NCA = número de columnas de A .

Como cada hilo tomará n diferentes renglones de la matriz A y calculará n diferentes renglones de la matriz C , se pueden dividir las iteraciones del primer ciclo *for* que hace referencia al renglón i de A utilizado en el cálculo actual de $C[i][j]$, con esto cada hilo usará distintos valores del índice i para la lectura de A y para el cálculo de C .

```
#pragma omp parallel for
for (i=0; i<NRA; i++)
{
    for(j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
}
```

Hasta aquí hace falta considerar si todas las variables involucradas deben ser compartidas o algunas deben ser privadas. A , B y C deben ser compartidas ya que en ningún momento los hilos tratan de escribir en un elemento compartido, pero j y k deben ser privadas porque cada hilo las modifica al realizar los ciclos internos.

El segmento paralelizado queda.

```
#pragma omp parallel for private (j,k)
for (i=0; i<NRA; i++)
{
    for(j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
}
```

Ejemplo 2- Cálculo del histograma de una imagen en tono de grises.

El histograma es el número de píxeles de cada nivel de gris encontrado en la imagen. En el programa la imagen es representada por una matriz de $n \times n$ cuyos elementos pueden ser mayores o iguales a 0 y menores a NG (tonos de gris) y se cuenta el número de veces que aparece un número (pixel) en la matriz y éste es almacenado en un arreglo unidimensional.

Por ejemplo, para la matriz (imagen) de 5×5 de la figura 13.2 que almacena valores de entre 0 y 5, en un arreglo unidimensional llamado histograma se almacena en el elemento con índice 0 el número de ceros encontrados, en el elemento con índice 1 el número de unos, en el elemento con índice 2 el número de dos etc..

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	176/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Imagen

1	3	5	0	1
0	3	4	2	3
1	0	5	3	3
2	0	4	0	1
2	0	5	3	3

histograma

0	1	2	3	4
6	4	3	7	2

Figura 13.2

El segmento de código correspondiente al cálculo del vector histograma es:

```
for(i=0; i<NG; i++)
    histo[i] = 0;

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        histograma[IMAGEN[i][j]]++;
```

Para paralelizarlo, se realiza una descomposición de datos, es decir, se dividirá la matriz de forma que cada hilo trabaje con un número de renglones diferentes y almacene la frecuencia de aparición de cada valor entre 0 y *NG* de la sub-matriz en un arreglo privado o propio de cada hilo llamado *histop*[]. Después se deben sumar todos los arreglos *histop*[] de cada hilo para obtener el resultado final en el arreglo *histo*[]. Figura 13.3.

1	3	5	0	1
0	3	4	2	3
1	0	5	3	3
2	0	4	0	1
2	0	5	3	3

Hilo 0
Hilo 1

histop de hilo 0

0	1	2	3	4
2	2	1	3	1

histop de hilo 1

0	1	2	3	4
4	2	2	4	1

histo[] = histop de hilo 0 + histop de hilo1

0	1	2	3	4
6	4	3	7	2

Figura 13.3

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	177/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

El segmento de código que realiza lo planteado es:

```

for(i=0; i<NG; i++)
    histo[i] = 0;

/*Calculo del histograma de IMAGEN*/
#pragma omp parallel private(histop) num_threads(2)
{
    for(i=0; i< NG; i++)
        histop[i]=0;

    #pragma omp for private(j)
        for (i=0; i<N; i++)
            for (j=0; j<N; j++)
                histop[IMA[i][j]] ++;

    #pragma omp critical
    {
        for( i=0 ; i < NG ; i++)
            histo[i]+=histop[i];
    }
}

```

Ejemplo 3. Cálculo de los números de la sucesión de Fibonacci

Para este ejemplo se requiere conocer de otros dos constructores que son útiles para paralelizar otro tipo de problemas como los que requieren manejo de recursividad o los que contienen ciclos con un número indeterminado de iteraciones. Estos son **task** y **taskwait**.

El constructor **task** sirve para generar tareas de forma explícita y esas tareas generadas pueden ser asignadas a otros hilos que se encuentran en la región paralela. Figura 13.4.

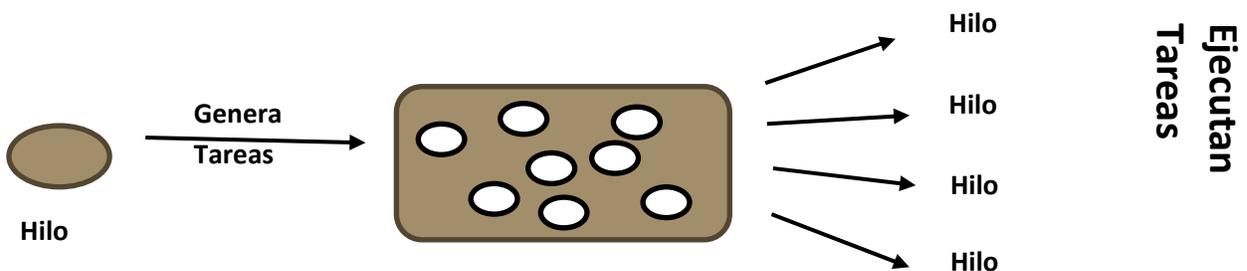


Figura 13.4

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	178/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Cuando un hilo encuentra el constructor **task**, se genera una nueva tarea, así que, si un hilo encuentra n veces al constructor **task**, genera n tareas.

El constructor **taskwait** permite esperar a que todas las tareas hijas generadas desde la tarea actual terminen su actividad.

Usar la directiva **task** en un programa recursivo para calcular el número $f(n)$ de la sucesión Fibonacci.

$$f(n) = f(n-1) + f(n-2) \text{ con } f(0) = 1 \text{ y } f(1) = 1$$

Una solución secuencial es:

```
#include <stdio.h>
long fibonacci(int n);

main (){
    int nthr=0;
    int n;
    long resul;
    printf("\n Número a calcular? ");
    scanf("%d", &n);
    resul = fibonacci(n);
    printf ("\nEl numero Fibonacci de %5d es %d", n, resul);
}

long fibonacci(int n) {

    long fn1, fn2, fn;
    int id;
    if ( n == 0 || n == 1 )
        return(n);

    if ( n < 30 ){
        fn1 = fibonacci(n-1);
        fn2 = fibonacci(n-2);
        fn = fn1 + fn2;
    }
    return(fn);
}
```

Ahora para la versión paralela se puede que un hilo empiece con la llamada a la función *fibonacci()* y después genere dos tareas una para el cálculo de f_1 y otra para f_2 que puedan ser realizadas por otros hilos de la región paralela y que estos a su vez al llamar a la función *fibonacci()* generaran otras nuevas tareas hasta que se llegue al caso base. Finalmente, el hilo que generó las primeras tareas realizará el cálculo final de f_n , solo que con el constructor **taskwait** esperará a que terminen todas las tareas generadas anteriormente.

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	179/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

El código en su versión paralela queda:

```
#include <stdio.h>
long fibonacci(int n);
main () {
    int nthr=0;
    int n;
    long resul;

    printf("\n Numero a calcular? ");
    scanf("%d", &n);

    #pragma omp parallel
    {
        #pragma omp single
        {
            resul = fibonacci(n);
        }
    }
    printf ("\nEl numero Fibonacci de %5d es %d", n, resul);
}

long fibonacci(int n){
    long fn1, fn2, fn;
    if ( n == 0 || n == 1 )
        return(n);
    if ( n < 30 ){
        #pragma omp task shared(fn1)
        {
            fn1 = fibonacci(n-1);
        }
        #pragma omp task shared(fn2)
        {
            fn2 = fibonacci(n-2);
        }
        #pragma omp taskwait
        {
            fn = fn1 + fn2;
        }
        return(fn);
    }
}
```

	Manual de prácticas del Laboratorio de Estructuras de datos y algoritmos II	Código:	MADO-20
		Versión:	01
		Página	180/180
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Desarrollo

Después de haber leído y analizado el contenido de la guía, realizar con ayuda del profesor las siguientes actividades:

Actividad 1

Completar la versión serie y paralela del ejemplo 1 explicado en la guía y medir el tiempo de ejecución de ambas versiones utilizando matrices de orden 500×500 .

Actividad 2

Completar la versión serie y paralela del ejemplo 2 explicado en la guía y medir el tiempo de ejecución de ambas versiones utilizando $N = 1000$ y $NG = 256$. Tomar tres lecturas y sacar el tiempo promedio para cada caso.

¿Cuánto tiempo tardaron ambas versiones?

¿Por qué en la versión paralela el cálculo de `histo[]` está delimitado con el constructor *critical*?

Actividad 3

Probar las versiones serie y paralela del ejemplo 3 para verificar que se obtienen los mismos resultados. Además, agregar a la versión paralela la impresión del identificador del hilo en la generación de cada tarea para visualizar los hilos que participan en los cálculos.

¿Qué pasa si $f1$ y $f2$ no se colocan como compartidas? Probar

¿Por qué sucede lo observado?

Actividad 4

Ejercicios sugeridos por el profesor

Nota: Existen otros constructores y cláusulas y se espera que con lo visto en la guía 11,12 y 13 el estudiante pueda comprenderlas con mayor facilidad.

Referencias

[1] <http://openmp.org/wp/>

[2] B. Chapman ,Using OpenMP, The MIT Press, 2002