

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

El algoritmo queda [1]:

OrdenacionHeapSort(A)

Inicio

construirHeapMaxIni(A)

 Para i=longitudDeA hasta 2 hacer

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

TamañoHeapA= **TamañoHeapA**-1;

MaxHeapify (A,1,TamañoHeap)

Fin

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[posMax])

MaxHeapify(A,posMax)

Fin Si

Fin

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].

construirHeapMaxIni(A)

Inicio

TamañoHeapA=longitudDeA

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

MaxHeapify(A,i)

Fin Para

Fin

```
#Autor | Elba Karen Sáenz García
def intercambia( A, x, y ):
    tmp = A[x]
    A[x] = A[y]
    A[y] = tmp

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

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

```
#HeapSort
#Autor Elba Karen Sáenz García

import math

def hIzq(i):
    return 2*i

def hDer(i):
    return 2*i+1

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 and A[L]>A[i] ):
        posMax=L
    else:
        posMax=i
```

```

    if ( L <= tamanoHeap and A[L]>A[i] ):
        posMax=L
    else:
        posMax=i

    if (R <= tamanoHeap and A[R]>A[posMax]):
        posMax=R
    if (posMax != i):
        intercambia(A,i,posMax)
        MaxHeapify(A,posMax,tamanoHeap)

def construirHeapMaxIni(A,tamanoHeap):
    for i in range(math.ceil(tamanoHeap/2) - 1, 0, -1):
        MaxHeapify(A,i,tamanoHeap)

def OrdenacioHeapSort(A,tamanoHeap):
    construirHeapMaxIni(A,tamanoHeap)
    for i in range (len(A[1:]), 1, -1):
        intercambia(A,1,i)
        tamanoHeap=tamanoHeap-1
        MaxHeapify(A,1,tamanoHeap)

```

Couting

```
#Counting Sort  
#Autor Elba Karen Sáenz García  
def CreaLista(k):  
    L=[]  
    for i in range(k+1):  
        L.append(0)  
    return L  
  
def CountingSort(A,k):  
    C=CreaLista(k)  
    B=CreaLista(len(A)-1)  
    for j in range(1,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)-1,0,-1):  
        B[C[A[j]]]=A[j]  
        C[A[j]]=C[A[j]]-1  
    return B
```

```

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(1,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)-1,0,-1):
        B[ C[A[j][1]] ][1]=A[j][1]
        B[ C[A[j][1]] ][0]=A[j][0]
        C[A[j][1]]=C[A[j][1]]-1
    return B

```

```

def FormaArregloConClaves(B,numCar):
    Btmp=[]
    for i in range(len(B)):
        Btmp.append([B[i]]*2)
        A3=list(B[i])
        Btmp[i][1]=ord(A3[numCar-1])
    return Btmp

```

```

def radixSort(A):
    numCar=len(A[1])
    for i in range (numCar,0,-1):
        cc=FormaArregloConClaves(A,i)
        ordenado=CountingSort2(cc,122)
        A=obtenerElemSinClaves(ordenado)
        print (A)
    return A

```

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

```
#busqueda 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
```

```
#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
```



```
#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)
```



```
#crear arreglo indexado 0-tamaño
def formaArreglo(tamaño):
    Arr=[None]*tamaño
    return Arr
```

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:

```
# Convertir la llave a valor numérico
def obtenerLlaveNumerica(llave):
    hash=0
    for char in str(llave):
        hash+=ord(char)
    return hash
```

Como función hash se utiliza $h(x) = x \bmod m$, la función en Python es:

```
#Funcion Hash
def H(llaveN):
    return llaveN%5
```

```

# Función donde dada una llave, se genera la dirección o índice
# en la tabla llamada map de tamaño n donde se agrega un valor.
def agregar(llave, valor, map, tamaño):

    # Dada la llave obtener el lugar donde se colocara valor (dirección)
    llave_hash = H(obtenerLlaveNumerica(llave))
    # Datos a colocar
    ParllaveValor = [llave, valor]

    # Si la dirección o posición esta vacía colocar datos
    if map[llave_hash] is None:
        map[llave_hash] = list([ParllaveValor])
        return True
    else:
        # Si la llave dada ya fue agregada, colocar valor en el mismo sitio

        for par in map[llave_hash]:
            if par[0] == llave:
                par[1] = valor
                return True

        # Si la llave genera una dirección ya ocupada (colisión),
        # se busca otra dirección
        # manejo de colisión con hash lineal
        for j in range(tamaño):
            llaveh = (llave_hash + j) % 13
            # Si la tabla ya esta llena
            if (llaveh == len(map)):
                print("Tabla llena", llave_hash)
                break
            else:
                # Si ya se encuentra una dirección vacía, se coloca el valor
                if map[llaveh] is None:
                    map[llaveh] = list([ParllaveValor])
                    return True

```

```

#Función que localiza el valor dada una llave
def buscar(llave,tamaño):
    #Dada la llave obtener el lugar donde probablemente
    #se encuentre el valor buscado
    llave_hash=H(obtenerLlaveNumerica(llave))
    #Si la posición no esta vacía
    if map[llave_hash] is not None:

        for par in map[llave_hash]:
            #Si la llave está en la posición generada por la función
            #se retorna el valor buscado
            if par[0] == llave:
                return par[1]
            #Si la llave generó una entrada que no contiene lo buscado
            # ¡Colisión!.Buscar posición alternativa
            else:
                for j in range(tamaño):
                    llaveh=(llave_hash+j)%13
                    #Si ya se busco en toda la tabla
                    if (llaveh==len(map)):
                        break

                for par1 in map[llaveh]:
                    #Si ya se localizó la dirección donde esta lo buscado
                    #retornar valor
                    if par1[0]== llave:
                        return par1[1]

    return None

```

Analizando esta función podemos ver como la tabla es este ejemplo con 13 elementos la cual se encuentra

```

#Función que localiza el valor dada una llave
def buscar(llave,tamaño):
    #Dada la llave obtener el lugar donde probablemente
    #se encuentre el valor buscado
    llave_hash=H(obtenerLlaveNumerica(llave))
    #Si la posición no esta vacía
    if map[llave_hash] is not None:

        for par in map[llave_hash]:
            #Si la llave está en la posición generada por la función
            #se retorna el valor buscado
            if par[0] == llave:
                return par[1]
            #Si la llave generó una entrada que no contiene lo buscado
            # ¡Colisión!.Buscar posición alternativa
            else:
                for j in range(tamaño):
                    llaveh=(llave_hash+j)%13
                    #Si ya se busco en toda la tabla
                    if (llaveh==len(map)):
                        break

                for parl in map[llaveh]:
                    #Si ya se localizó la dirección donde esta lo buscado
                    #retornar valor
                    if parl[0]== llave:
                        return parl[1]

    return None

```

Finalmente con esta función se forma la tabla en este ejemplo con 13 elementos la cual se ve así:

```

#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()

```

```
def bfs(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'
```

```
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))
```

```
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
```

```

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

```

```

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

```



```
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)
```

```
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)
```

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)
```

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]);
        }
    }
}
```

```
#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);
    }
}
```

