Guía Práctica de Estudio 2

Algoritmos de ordenamiento parte 2

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaño

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 lista.

Para su descripción, la secuencia a ordenar está representada por un arreglo lineal o unidimensional, aunque también se puede utilizar una lista ligada.

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.

$\leq x$			x			$>x$		
32	25	45	50	82	60	78	94	92

Parte inferior de la lista — Parte superior de la lista

Figura 2.1

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

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

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

p	q	r
$\leq x$	x	$> x$

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 las sub-listas son parte del arreglo A y cada una ya está ordenada no es necesario combinarlas, el arreglo A[p, …,r] ya está ordenado.

Un algoritmo general 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].

```
QuickSort(A,p,r)
Inicio
  Si p < r entonces   // Si la lista tiene más de un elemento
          q =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 el sub-arreglo para que de un lado del elemento pivote queden todos los menores o iguales y del otro todos los mayores a él.

En la función *Particionar()* en pseudocódigo [1] mostrada abajo, primero se define el elemento pivote x como el último elemento de la sublista ($x = A[r]$) 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 rvisar 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]<=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 al 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.

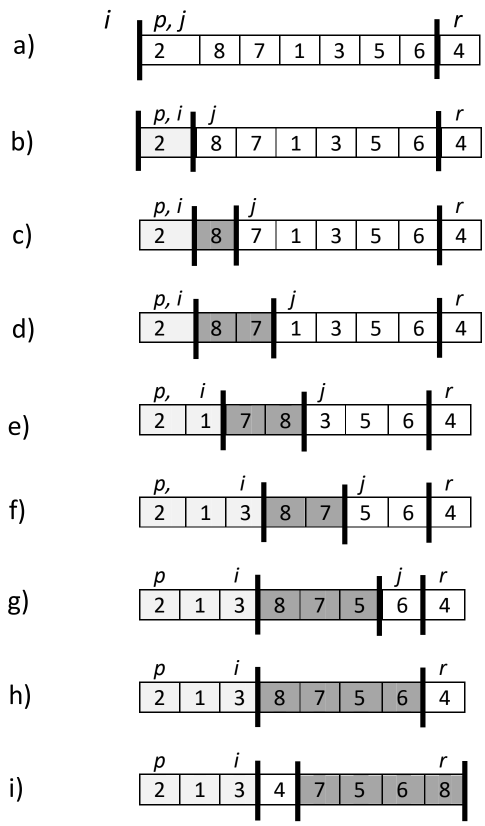

Figura 2.3 [1]

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.

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.

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.

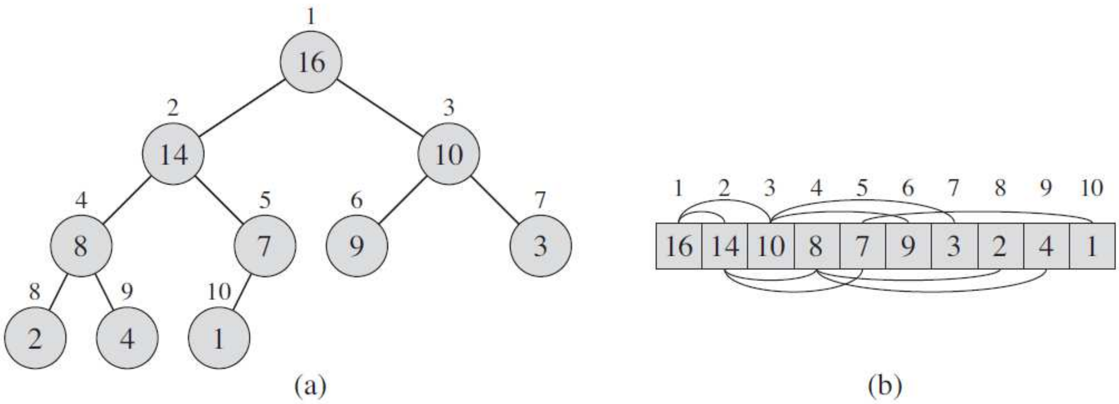

Figura 2.4 [1]

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
$0 \leq TamañoHeapA \leq \ longtudDeA.$

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:

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

Padre(i)
Inicio
retorna $\lfloor \ i/2 \ \rfloor$
Fin

hIzq(i)
Inicio
retorna $2 * i$
Fin

IDer(i)
Inicio
retorna $2 * i$+1
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 en 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 de 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 $A[1], A[2], A[3], \ldots., A[n]$. 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 $A[1], A[2], A[3], \ldots., A[n-1]$ y se vuelven a intercambiar los valores $A[1]$ y $A[n-1]$ para reconstruir nuevamente el montículo con los elementos $A[1], A[2], A[3], \ldots., A[n-2]$. El proceso se realizar 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 l*ongitudDeA* y el número de elementos en el ***heap*** *TamañoHeapA*.

El algoritmo queda [1]:

```
OrdenacionHeapSort( A)
Inicio
        construirHeapMaxIni( A)
        Para i=longiudDeA hasta 2 hacer
                Intercambia(A[1], A[i])
                TamañoHeapA= TamañoHeapA-1;
                MaxHeapify (A,1,TamañoHeap)
Fin
```

Donde la función *constuirHeapMaxIni( )* 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 <TamañoHeapA y A[L]>A[i]
             posMax=L
     En otro caso
              posMax = i
     Fin Si
     Si R<TamañoHeapA y A[R]> A[posMax] entonces
              posMax =R
     Fin Si
     Si posMax ≠ i entonces
            Intercambia(A[i], A[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 intercambió 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 pseudocodigo queda *[1]*.

```
construirHeapMaxIni( A )
Inicio
TamañoHeapA=longiudDeA
    Para i=⌊longiudDeA/2⌋ 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.

Nota: Considerar que en la lista creada el elemento con índice 0 no se toma en cuenta para el ordenamiento, por ejemplo, si la lista es {9,21,4,40,10,35} en Python se debe definir como {0,9,21,4,40,10,35}. Esto porque en el algoritmo descrito el índice de la lista inicia en 1.

Las funciones en Python del algoritmo Quicksort descrito en la guía son las siguientes:

```
#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)
```

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

¿Qué modificaciones se tienen que hacer para ordenar la lista en orden inverso? 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.

Nota: Considerar que en la lista creada el elemento con índice 0 no se toma en cuenta para el ordenamiento, por ejemplo, si la lista es {9,21,4,40,10,35} en Python se debe definir como {0,9,21,4,40,10,35}. Esto porque en el algoritmo descrito el índice de la lista inicia en 1.

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

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

```
#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 (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)
```

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