



# Facultad de Ingeniería UNAM



Maestro: Tista Garcia Edgar MI.

Asignatura: Estructura de Datos y Algoritmos II

Práctica 2 : Algoritmos de Ordenamiento parte 2

Alumno: Mejía Alba Israel Hipólito

Grupo:8

Objetivo: El estudiante identificará la estructura de los algoritmos de ordenamiento selection sort, insertion sort y heapsort.

Desarrollo:

Ejercicio 1:

a)

#### Descripción del Insertion Sort:

El método insertionSort recibe arr2 del cual le asignarla a la variable n el tamaño del mismo y en base a eso, repetir dentro de un ciclo que valla de 1 a n las siguientes instrucciones:

asignar una a key el valor actual del contador, después si no se sale el indice del arreglo y el valor anterior del elemento del mismo a comparar es mayor, se tiene que que cambiar de posición el elemento y se guarda el otro en la variable key. Esto se repite las veces necesarias para que el arreglo termine ordenado

#### Diferencias a lo visto en clase:

Todo es igual, el unico cambio es el uso de las variables i y j, este se invierte, pero fuera de ello es lo mismo.

b)

Descripción de las diferentes formas de llamar las funciones de insertion sort y selection sort :

Para el caso de InsertionSort, se llama mediante el método que manda un a arr2 para que se ordene. Por otro lado para realizar el SelectionSort, se crea el objeto selección que contiene al método selectionSort al cual se le manda el arr1 para que lo ordene.

#### c) Descripción del código de SelectionSort

Hay un ciclo que se refiere al número de interacciones que hará nuestro programa, que es igual al número de elementos que contiene el arreglo. Se le asigna el valor del mínimo por convención al primer elemento de la lista y este se va comparando

este con el siguiente elemento que tiene en el arreglo, si este último es menor, su índice se le asigna a la variable mínimo.

De esta manera si se realizó el cambio el valor de mínimo, en las próximas instrucciones se intercambiarán las posiciones de los elementos de la lista comparados, si no se realizó cambio en el valor del mínimo, no pasara nada.

### Diferencias a lo visto en clase

No fue necesario crear la variable n para usarla en el ciclo de iteraciones, hasta el momento es la única diferencia que encontré ya que la metodología del algoritmo es prácticamente la misma.

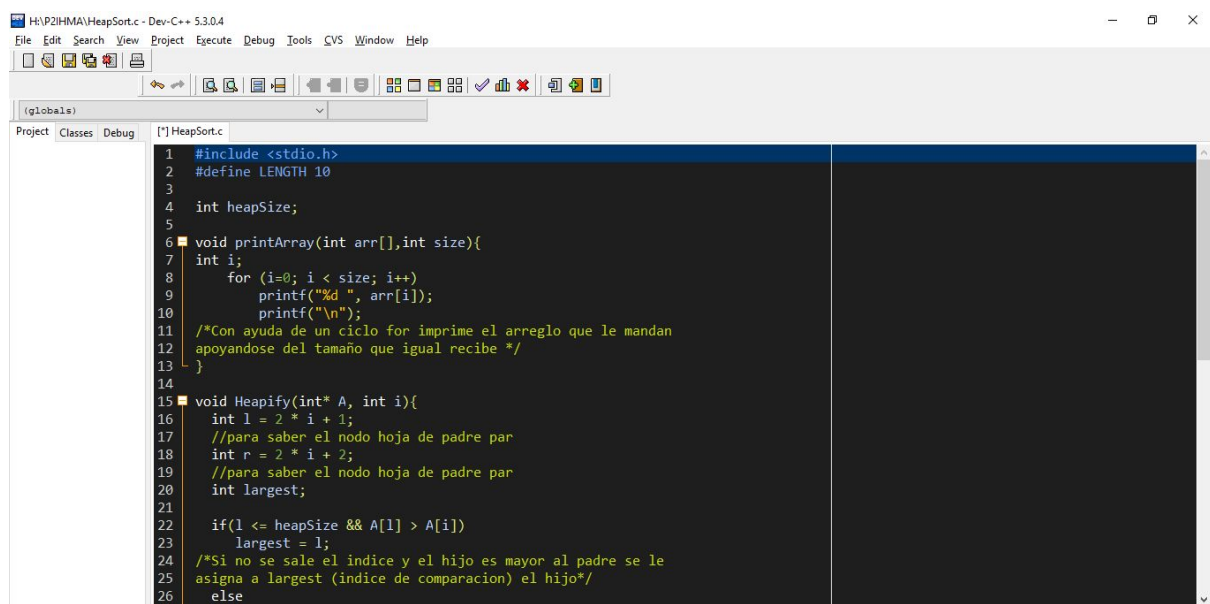
### d) Comentarios de lo que me llamaron la atención en JAVA:

La facilidad de conocer la longitud de un arreglo usando únicamente `.length`, el uso de clases públicas en lugar de archivos creo que facilita mucho el uso sus métodos en el programa principal, y por último algo que me llamó mucho la atención fue la línea 17 de la clase `SelectionSort`, donde viene `for(int j:arr)`, la verdad no se que haga, no vi alguna acción en la consola y no se porque se usa los dos puntos en este caso.

### Ejercicio 2:

#### a) Funciones comentadas de Bottom Up:

De una vez aproveche para comentar todas las funciones necesarias para crear el arreglo ordenado del `HeapSort`, cabe destacar que las funciones necesarias para el Bottom Up se encuentran en la función `Heapify`:



```
1 #include <stdio.h>
2 #define LENGTH 10
3
4 int heapSize;
5
6 void printArray(int arr[],int size){
7     int i;
8     for (i=0; i < size; i++)
9         printf("%d ", arr[i]);
10    printf("\n");
11    /*Con ayuda de un ciclo for imprime el arreglo que le mandan
12    apoyandose del tamaño que igual recibe */
13 }
14
15 void Heapify(int* A, int i){
16     int l = 2 * i + 1;
17     //para saber el nodo hoja de padre par
18     int r = 2 * i + 2;
19     //para saber el nodo hoja de padre par
20     int largest;
21
22     if(l <= heapSize && A[l] > A[i])
23         largest = l;
24     /*Si no se sale el indice y el hijo es mayor al padre se le
25     asigna a largest (indice de comparacion) el hijo*/
26     else
```

```
26     else
27         largest = i;
28         //caso contrario el padre conserva el indice de comparacion
29         if(r <= heapSize && A[r] > A[largest])
30             largest = r;
31         /*Si no se sale el indice y el hijo es mayor al padre se le
32         asigna a largest el hijo*/
33         if(largest != i)
34             //si el indice de comparacion no es par
35             {
36                 printArray(A, LENGTH);
37                 int temp = A[i];
38                 A[i] = A[largest];
39                 A[largest] = temp;
40                 /*manda a imprimir el arreglo e intercambia, para despues intercambiar la
41                 posicion el valor del ultimo indice de comparacion con el indice que sera igual
42                 a la ultima comparacion por lo que se entiende que lo que queda a la izquierda ya
43                 esta ordenado*/
44                 Heapify(A, largest);
45                 //se vuelve hacer lo anterior
46             }
47     }
48
49     void BuildHeap(int* A){
50         heapSize = LENGTH - 1;
51         int i;
52
53         for(i = (LENGTH - 1) / 2; i >= 0; i--){
54             Heapify(A, i);
55         }
56         /*Las iteraciones necesarias para el heap y las hace sobre heapify*/
57         printf("Termino de construir el HEAP \n\n");
58     }
59
60     void HeapSort(int* A){
61         BuildHeap(A);
62         /*manda a llamar la funcion BuildSort para recibir el heap*/
63         int i;
64         for(i = LENGTH - 1; i > 0; i--){
65             int temp = A[heapSize];
66             A[heapSize] = A[0];
67             A[0] = temp;
68             heapSize--;
69             printf("Iteracion HS: \n");
70             Heapify(A, 0);
71             /*Elimina la raiz asignandola al arreglo ordenado y lo demas lo manda a
72             que se vuelva a hacer el heap */
73         }
74     }
75
76     int main(){
77         int prueba[LENGTH] = {10,4,12,6,70,5,8,23,31,38};
78
79         HeapSort(prueba);
80         //manda a realizar el heapsort
81         int i;
82         for(i = 0; i < LENGTH; i++){
83             // for(i=LENGTH; i>0;i--)
84             /*Para cambia a orden descendente de la impresion del arreglo ordenado*/
85             printf("%d ",prueba[i]);
86         }
87         //imprime el arreglo ordenado
88         return 0;
89     }
```

b) Modificaciones para que el algoritmo se haga de manera descendente:

```
14 void Heapify(int* A, int i){
15     int l = 2 * i + 1;
16     //para saber el nodo hoja de padre par
17     int r = 2 * i + 2;
18     //para saber el nodo hoja de padre par
19     int smaller;
20
21     if(l <= heapSize && A[l] < A[i])
22         smaller = l;
23     /*Si no se sale el indice y el hijo es MENOR al padre se le
24     asigna a smaller (indice de comparacion) el hijo*/
25     else
26         smaller = i;
27     //caso contrario el padre conserva el indice de comparacion
28     if(r <= heapSize && A[r] < A[smaller])
29         smaller = r;
30     /*Si no se sale el indice y el hijo es MENOR al padre se le
31     asigna a smaller el hijo*/
32     if(smaller != i)
33         //si el indice de comparacion no es par
34         {
35             printArray(A, LENGTH);
36             int temp = A[i];
37             A[i] = A[smaller];
38             A[smaller] = temp;
39             //manda a imprimir el arreglo e intercambia, para despues intercambiar la
40             posicion el valor del ultimo indice de comparacion con el indice que sera igual
41             a la ultima comparacion por lo que se entiende que lo que queda a la izquierda ya
42             esta ordenado*/
43             Heapify(A, smaller);
44             //se vuelve hacer lo anterior
45         }
46 }
47
```

```
10 4 12 6 70 5 8 23 31 38
10 4 12 6 38 5 8 23 31 70
10 4 5 6 38 12 8 23 31 70
4 10 5 6 38 12 8 23 31 70
Termino de construir el HEAP

Iteracion HS:
Iteracion HS:
Iteracion HS:
Iteracion HS:
Iteracion HS:
Iteracion HS:
Iteracion HS:
70 31 23 8 12 38 10 5 6 4
-----
Process exited after 0.1335 seconds with return value 0
Presione una tecla para continuar . . .
```

```
50 heapSize = LENGTH - 1;
51 int i;
52 for(i = (LENGTH - 1) / 2; i >= 0; i--)
53     Heapify(A, i);
54 /*Las iteraciones necesarias para el heap y las hace sobre heapify*/
55 printf("Termino de construir el HEAP \n\n");
56 }
57
58 void HeapSort(int* A){
59     BuildHeap(A);
60     /*manda a llamar la funcion BuildSort para recibir el heap*/
61     int i,d=0;
62     //for(i = LENGTH - 1; i > 0; i--){
63     for(i=0; i<LENGTH-1; i++){
64         int temp = A[d];
65         A[d] = A[0];
66         A[0] = temp;
67         heapSize--;
68         printf("Iteracion HS: \n");
69         Heapify(A, 0);
70     /*Elimina la raiz asignandola al arreglo ordenado y lo demas lo manda a
71     que se vuelva a hacer el heap */
72     }
73 }
74
75
76 int main(){
77     int prueba[LENGTH] = {10,4,12,6,70,5,8,23,31,38};
78     HeapSort(prueba);

```

Cambiando las comparaciones entre el nodo padre e hijo dentro de la función Heapify, se logró hacer que el ordenamiento fuera descendente, sin embargo, no encuentro la manera de que se impriman bien las iteraciones, se que esto último se hace dentro la función HeapSort, pero aun no encuentro la forma de hacerlo.

Conclusiones:

Me pareció adecuada la práctica ya que pude analizar y comprender los algoritmos de ordenamiento selection sort, insertion sort y heap sort, usando el método de bottom up en este último. De igual manera al analizar cada línea de código y comparándolas con lo visto en clase me ayudó bastante a reforzar mis conocimientos en estos algoritmos.