Facultad:	Ingeniería
Escuela:	Computación
Asignatura:	Programación IV

Tema: **“Métodos de Ordenamiento parte III. HeapSort”**

Objetivos

- Identificar la estructura de algunos algoritmos de ordenamiento.
- Interpretar los algoritmos de ordenamiento en sintaxis de C#.
- Aplicar el algoritmo de ordenamiento HeapSort.

Introducción Teórica

Montículos (Heaps).

Un montículo posee la siguiente característica: “Para todo nodo del árbol se debe cumplir que su valor es mayor o igual al de cualquiera de sus hijos”.

Para representar un montículo en un arreglo lineal:

1. El nodo “k” se almacena en la posición k correspondiente del arreglo.
2. El hijo izquierdo del nodo “k”, se almacena en la posición 2 * k.
3. El hijo derecho del nodo “k”, se almacena en la posición 2 * k+1.

La estructura de datos Montículo es un arreglo de objetos que puede ser visto como un árbol binario con raíz, cuyos nodos pertenecen a un conjunto totalmente ordenado, y tal que cumple las siguientes dos propiedades:

a) Propiedad de orden: La raíz de cada subárbol es mayor o igual que cualquiera de sus nodos restantes.

b) Propiedad de forma: La longitud de toda rama es h o h−1, donde “h” es la altura del árbol. Además, si una rama termina en una hoja a la derecha de otra hoja, ésta última de altura h−1, la primera debe ser de altura h−1.

Fig 1: Representación de un Montículo

Utilizamos un array para representar un árbol binario.

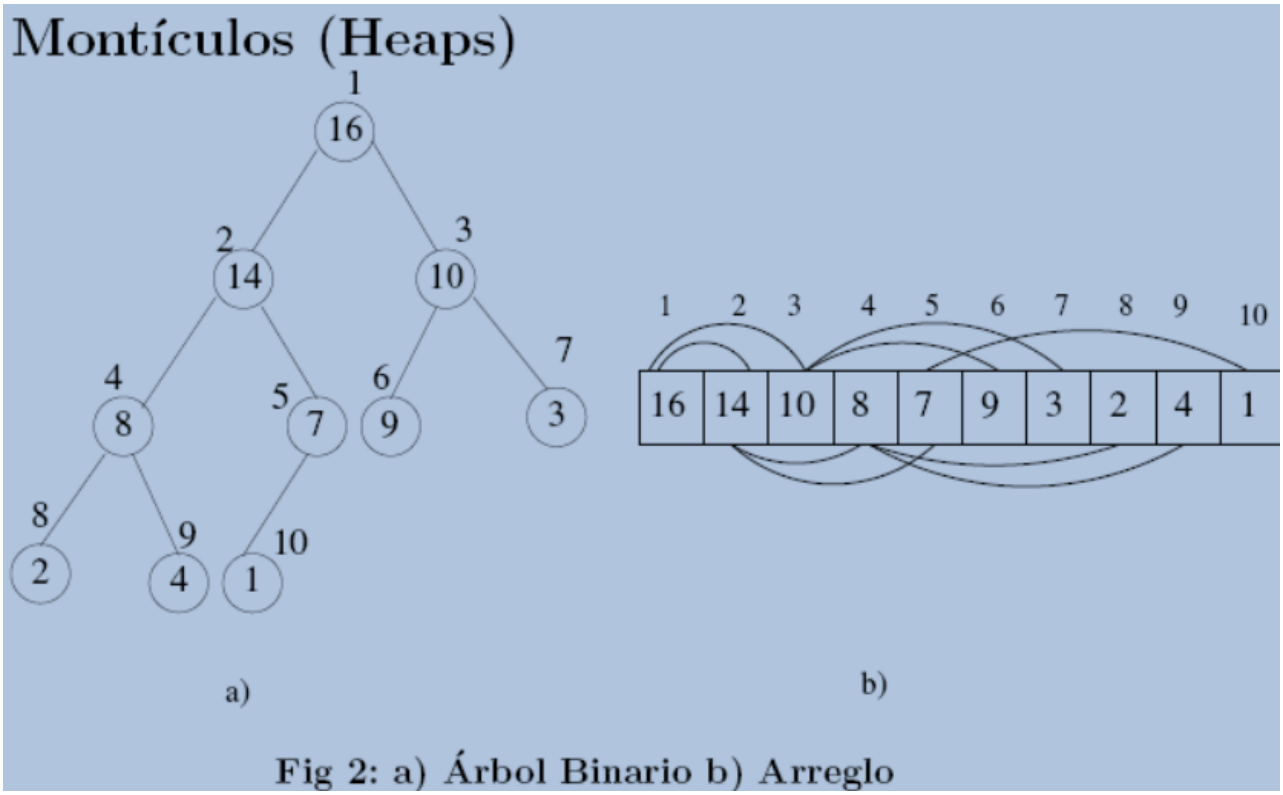


Fig 2: a) Árbol Binario b) Arreglo

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

- Length [A]: número de elementos en el arreglo.
- Heap-size [A]: número de elementos en el montículo almacenados dentro de “A”.

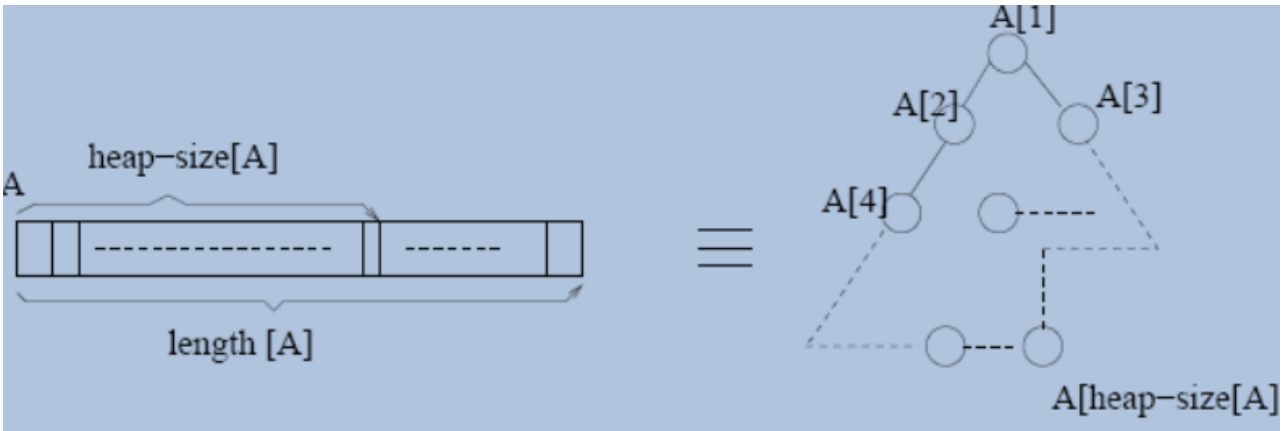


Fig 3: Representación de un Montículo

A [1….heap − size [A]] es el montículo representado.

Método de Ordenamiento HeapSort.

Se toman las mejores características de los dos algoritmos de ordenamiento basados en comparación (MergeSort e InsertionSort) para crear un nuevo algoritmo llamado HeapSort. Este algoritmo está basado en la estructura de montículo.

El método de ordenación HeapSort es también conocido con el nombre “montículo” en el mundo de habla hispana. Su nombre se debe a su autor J. W. J. Williams quien lo bautizó así en 1964. Es el más eficiente de los métodos de ordenación que trabajan con árboles.

La idea central de este algoritmo consiste en lo siguiente:
Construir un montículo.
Eliminar la raíz del montículo en forma repetida.

El método de ordenación se puede describir con los siguientes pasos:
1. Construir un montículo inicial con todos los elementos del vector A [1], A [2], …., A [n]
2. Intercambiar los valores de A [1] y A [n] (siempre queda el máximo en el extremo)
3. Reconstruir el montículo con los elementos A [1], A [2],……, A [n-1]
4. Intercambiar los valores de A [1] y A [n-1]
5. Reconstruir el montículo con los elementos A [1], A [2],……, A [n-2]

Este es un proceso iterativo que partiendo de un montículo inicial, repite intercambiar los extremos, decrementar en 1 la posición del extremo superior y reconstruir el montículo del nuevo vector.

Lo expresamos en forma algorítmica así:

```
Ordenación Heapsort (Vector, N)
Debe construirse un montículo inicial (Vector, N)
desde k = N hasta 2 hacer
        intercambio (Vector [1], Vector [k])
        construir montículo (Vector, 1, k-1)
fin desde
```

Entonces de acuerdo al algoritmo debemos considerar dos problemas:

- Construir el montículo inicial.
- Cómo restablecer los montículos intermedios

Para restablecer el montículo hay dos posibilidades:

- A [1] es menor o igual que los valores de sus hijos, entonces la propiedad del montículo no se ha roto.
- En otro caso, el elemento mínimo que necesitamos poner en A [1] es o su hijo izquierdo o su hijo derecho (A [2], A [3] respectivamente): por lo que se determina el menor de sus hijos y éste se intercambia con A [1]. El proceso continúa repitiendo las mismas comparaciones entre el nodo intercambiado y sus nodos hijos; así hasta llegar a un nodo en el que no se viole la propiedad de montículo, o estemos en un nodo hoja.

Rutina Heapify.

Heapify es una importante subrutina para manipular montículos.

{Pre: subárbol de raíz Izq (i) y subárbol de raíz Der (i) son montículos}.

{Post: subárbol de raíz i es un montículo}.

```
Heapify (A, i)
        izq = Izq ( i ) // La función Izq ( i ) = 2 * i
        der = Der ( i ) // La función Der ( i ) = 2 * i + 1
```

```
if izq ≤ heap-size [A] and A [izq] > A [ i ] then
        pos-max = izq
else
        pos-max = i
if der ≤ heap-size [A] and A [der] > A [pos-max] then
        pos-max = der
if pos-max ≠ i then
        intercambiar (A [ i ], A [pos-max])
        Heapify (A, pos-max)
```

Construcción de un Montículo.

Utilizando Heapify:

```
Build-Heap (A)
    Heap-size [A] = length [A]
    for I = [length [A]/2] down to 1
        do Heapify (A, I)
```

Algoritmo HeapSort.

```
HeapSort (A)
1  Build-Heap (A)
2  For j = length [A] down to 2
3      do intercambio (A [1], A [ j ])
4  heap-size [A] = heap-size [A] − 1
5  Heapify (A, 1)
```

El significado de heap en ciencia computacional es el de una cola de prioridades (priority queue).

Tiene las siguientes características:

- ✓ Un heap es un arreglo de “n” posiciones ocupado por los elementos de la cola. (Nota: se utiliza un arreglo que inicia en la posición 1 y no en cero, de tal manera que al implementarla en C# se tienen n+1 posiciones en el arreglo.)

- ✓ Se mapea un árbol binario de tal manera en el arreglo que el nodo en la posición “i” es el padre de los nodos en las posiciones (2 * i) y (2 * i + 1).
- ✓ El valor en un nodo es mayor o igual a los valores de sus hijos. Por consiguiente, el nodo padre tiene el mayor valor de todo su subárbol.

HeapSort consiste esencialmente en:

- ➢ Convertir el arreglo en un heap.
- ➢ Construir un arreglo ordenado de atrás hacia adelante (mayor a menor) repitiendo los siguientes pasos:
  - o Sacar el valor máximo en el heap (el de la posición 1).
  - o Poner ese valor en el arreglo ordenado.
  - o Reconstruir el heap con un elemento menos.
- ➢ Utilizar el mismo arreglo para el heap y el arreglo ordenado.

Materiales y Equipo

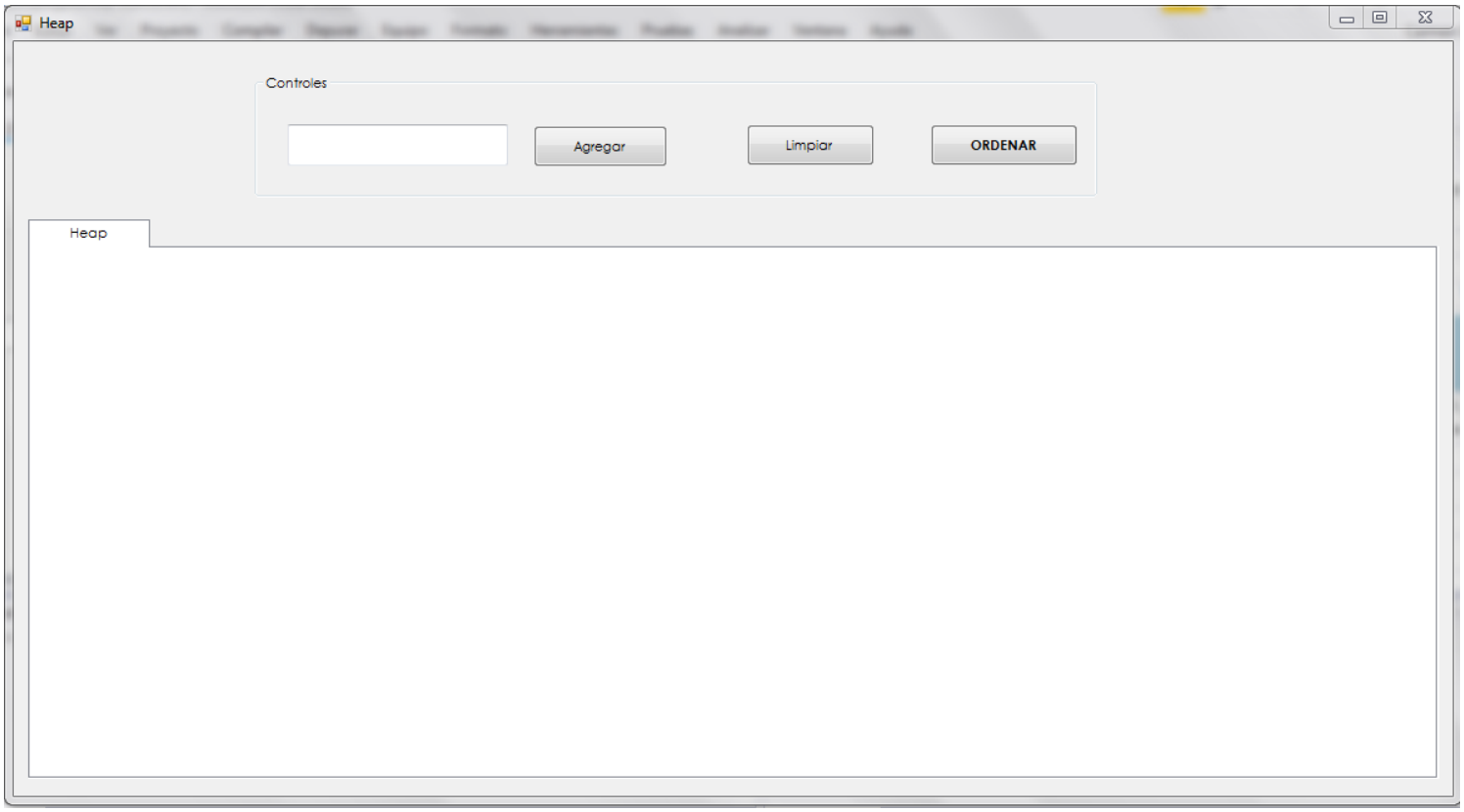
- Guía de Laboratorio Nº 5
- Computadora con programa: Visual Studio C#
- Dispositivo de Almacenamiento (USB).

Procedimiento

Ejemplo 1

Realizaremos una simulación de un montículo en entorno gráfico, para ello siga los pasos que se detallan:

Crear un nuevo proyecto en Visual C# (Windows Form), siga la pantalla sugerida, puede realizar los cambios que considere más favorables pero deberá aplicarlos también en el código.



Para realizar la pantalla anterior se han utilizado las siguientes herramientas

Elemento	Nombre	Observación
3 buttons	Agregar Limpiar Ordenar	Cambiar el nombre de cada botón según corresponda
1 textBox		Dejar en blanco para ingresar valores
1 groupBox		Agrupar los botones y el textBox ahí
1 tabControl		Solamente dejará 1 tabPage (el tabPage ya viene con el tabControl, tiene 2 por defecto. Borramos uno y nos quedamos con 1)

1 Form	Se recomienda que el Form tenga medidas de 1030, 708.

1. Una vez creado el formulario ingrese al código del Form e inicializará elementos como se muestra en la imagen (SOLAMENTE OCUPAREMOS EL FORM.CS, no hay más clases)

```
1   using System;
2   using System.Drawing;
3   using System.Threading;
4   using System.Windows.Forms;
5
6   namespace ejemploguiaheap
7   {
8       public partial class Form1 : Form
9       {
10          //INICIALIZACIÓN DE VARIABLES
11          int xo, yo, tam; //variables para valor inicial de x, de y y de tamaño
12          bool ec = false; //bandera booleana en falso
13          bool estado = false; //estado inicializado en falso
14          int n = 0, i = 1;  //inicialización de variables
15          int[] Arreglo_numeros; //arreglo de números ingresados
16          Button[] Arreglo; //arreglo de botones para simular valores ingresados
17
18          //INICIALIZACIÓN DEL FORMULARIO
19          public Form1()
20          {
21              InitializeComponent();
22              tam = tabPage1.Width / 2;  //tam será la mitad del ancho del tabpage
23              xo = tam; //el valor inicial de x será la mitad del ancho del tabpage
24              yo = 20; //el valor inicial en y será de 20
25              txtNumero.Focus(); //cursor en textbox
26          }
```

2. Activamos el evento click del botón de Agregar y el código que irá en el evento es

```
//EVENTO CLICK DE AGREGAR
private void btnAgregar_Click(object sender, EventArgs e)
{
    try
    {
        int num = int.Parse(txtNumero.Text); //capturamos el valor ingresado

        Array.Resize<int>(ref Arreglo_numeros, i + 1); //incrementamos el arreglo en base al nuevo valor ingresado
        Arreglo_numeros[i] = num; //asignamos ese valor a la posición i del arreglo
        Array.Resize<Button>(ref Arreglo, i + 1); //incrementamos el arreglo de botones
        Arreglo[i] = new Button(); //creamos un nuevo botón i
        Arreglo[i].Text = Arreglo_numeros[i].ToString(); //texto del botón será el valor ingresado de posición i
        Arreglo[i].Height = 50; //alto de botón 50
        Arreglo[i].Width = 50; //ancho de botón 50
        Arreglo[i].BackColor = Color.GreenYellow; //color GreenYellow para botón
        Arreglo[i].Location = new Point(xo, yo) + new Size(-20, 0); //punto de ubicación

        //para poder dibujar el árbol y crear los niveles

        if ((i + 1) == Math.Pow(2, n + 1)) //para saber nivel en base a los nodos (si tenemos que cambiar de nivel)
        {
            n++; //incrementamos n
            tam = tam / 2;  //dividimos de nuevo tam
            xo = tam; // el valor inicial de xo será el nuevo tam
            yo += 60;  //incrementamos el y en 60 para que el siguiente nivel se dibuje 60 espacios más abajo en y
        }
        else
        {
            xo += (2 * tam);  //si no hay que cambiar de nivel solo movemos el valor de x
        }

        i++; //incrementamos i
        estado = true; //pasamos estado a true
        ec = false;
        tabPage1.Refresh();  //refrescamos el tabpage
        txtNumero.Clear(); //limpiamos el textbox
        txtNumero.Focus(); //cursor nuevamente ahí
    }
    catch { MessageBox.Show("Valor no valido"); } //error
}
```

3. Ahora hay que programar el evento click del botón Limpiar

```
69          //EVENTO CLICK DE LIMPIAR
70          private void btnLimpiar_Click(object sender, EventArgs e)
71          {
72              //limpiamos pantalla y reinicializamos a valores iniciales
73              n = 0;
74              i = 1;
75              tam = tabPage1.Width / 2;
76              xo = tam;
77              yo = 20;
78              tabPage1.Controls.Clear();
79              tabPage1.Refresh();
80              Array.Resize<int>(ref Arreglo_numeros, 1);
81              Array.Resize<Button>(ref Arreglo, 1);
82          }
```

4. Programando el evento click de Ordenar

```
84          //EVENTO CLICK DE ORDENAR
85          private void btnOrdenar_Click(object sender, EventArgs e)
86          {
87              if (i == 1)
88                  MessageBox.Show("No hay elementos que ordenar");
89              else
90              {
91                  btnAgregar.Enabled = false; //mientras se ordena deshabilitamos botones para que no interfiera
92                  btnLimpiar.Enabled = false; //con la simulación
93                  btnOrdenar.Enabled = false;
94                  this.Cursor = Cursors.WaitCursor; //hacemos que el cursor espere y mostramos que está procesando
95
96                  if (!ec)  {
97                      Heap_Num();  //llamamos el heap num
98                  }
99                  else {
100                     HPN(); //llamamos hpn
101                 }
102
103                 btnAgregar.Enabled = true; //habilitamos todo de nuevo
104                 btnLimpiar.Enabled = true;
105                 btnOrdenar.Enabled = true;
106                 this.Cursor = Cursors.Default;  //regresamos el cursor a su forma normal
107             }
108         }
```

5. Hay que programar un método que permita intercambiar los botones que contienen a los valores que deben reorganizarse para el heap. Este método es llamado a la hora de hacer el ordenamiento.

```
110         //método para intercambio
111         public void intercambio(ref Button[] botones, int a, int b)
112         {
113             string temp = botones[a].Text;  //dejamos valores en un temporal
114 
115             Point pa = botones[a].Location;  // sacamos ubicación de a
116             Point pb = botones[b].Location;  // sacamos ubicación de b
117 
118             int diferenciaX = Math.Abs(pa.X - pb.X);  //sacamos la distancia entre sus x
119             int diferenciaY = Math.Abs(pa.Y - pb.Y);  //sacamos la distancia entre sus y
120 
121             int x = 10;
122             int y = 10;
123             int t = 70;
124             while (y != diferenciaY + 10) //mientras no llegue a la posición esperada en y
125             {
126                 Thread.Sleep(t);  //suspendemos durante 70 milisegundos
127                 botones[a].Location += new Size(0, -10);  //movemos a -10
128                 botones[b].Location += new Size(0, +10);  //movemos b +10
129                 y += 10;
130             }
131             while (x != diferenciaX - diferenciaX % 10 + 10)  //mientras no llegue a la posición esperada en x
132             {
133                 if (pa.X < pb.X) //si X de a es menor que X de b
134                 {
135                     Thread.Sleep(t);  //dormir durante 70 milisegundos
136                     botones[a].Location += new Size(+10, 0);  //movemos +10 a
137                     botones[b].Location += new Size(-10, 0);   //movemos -10 a b
138                     x += 10;
139                 }
140                 else
141                 {
142                     Thread.Sleep(t); //dormir durante 70 milisegundos
143                     botones[a].Location += new Size(-10, 0); //movemos a -10 en x
144                     botones[b].Location += new Size(+10, 0);  //movemos b +10 en x
145                     x += 10;
146                 }
147             }
148 
149             botones[a].Text = botones[b].Text;  //valor de b se muestra en a
150             botones[b].Text = temp;  //el valor temporal almacenado se mostrará en b
151             botones[b].Location = pb; // nuevo pb, se almacenará ubicación
152             botones[a].Location = pa; //nuevo pa, se almacenará ubicación
153             estado = true; //estado en true
154 
155             tabPage1.Refresh(); //se refresca tabpage
156         }
```

6. El siguiente paso que se realiza es activar el evento Paint del tabPage por lo que nos iremos al entorno gráfico y en el listado de eventos seleccionamos Paint, damos click a dicho evento y nos llevará al código; ahora codificaremos lo siguiente:

```
158         //Evento paint del tabpage (activarlo desde el diseño
159         private void tabPage1_Paint(object sender, PaintEventArgs e)
160         {
161             if (estado) //si estado es verdadero
162             {
163                 try
164                 {
165                     Dibujar_Arreglo(ref Arreglo, ref tabPage1);  //dibujar arreglo
166                     dibujar_Ramas(ref Arreglo, ref tabPage1, e); //dibujar ramas
167                 }
168                 catch
169                 { }
170                 estado = false;  //pasar estado a falso
171             }
172         }
```

7. Ahora haremos el método para dibujar el arreglo, con el código siguiente:

```
174         //método para dibujar arreglo
175         public void Dibujar_Arreglo(ref Button[] botones, ref TabPage tb)
176         {
177             for (int j = 1; j < botones.Length; j++)
178             {
179                 tb.Controls.Add(this.Arreglo[j]);
180             }
181         }
```

8. Se dibujan las ramas que unirán los botones (los nodos) del árbol

```
//método para dibujar ramas
public void dibujar_Ramas(ref Button[] botones, ref TabPage tb, PaintEventArgs e)
{
    Pen lapiz = new Pen(Color.Gray, 1.5f);
    Graphics g;
    g = e.Graphics;

    for (int j = 1; j < Arreglo.Length; j++) //para todos los elementos del arreglo
    {
        if (Arreglo[(2 * j)] != null) //mientras el arreglo no esté vacio
            g.DrawLine(lapiz, Arreglo[j].Location.X, Arreglo[j].Location.Y + 20, Arreglo[(2 * j)].Location.X + 20, Arreglo[(2 * j)].Location.Y);
        if (Arreglo[(2 * j) + 1] != null) //mientras no haya solo un elemento
            g.DrawLine(lapiz, Arreglo[j].Location.X + 40, Arreglo[j].Location.Y + 20, Arreglo[(2 * j) + 1].Location.X + 20, Arreglo[(2 * j) + 1].Location.Y);
    }
}
```

9. Método heap

```
199     //método heap máximizante
200     public void Heap_Num()
201     {
202         ec = true; //pasamos bandera a true
203
204         int x = Arreglo.Length;  //tomamos la longitud del arreglo
205
206         for (int i = (x) / 2; i > 0; i--)  //desde la mitad de la longitud decrementamos
207             Max_Num(Arreglo_numeros, x, i, ref Arreglo);
208     }
```

10. Método HPN o heap de números (llamado en el ordenamiento)

```
210     //método para intercambiar valores de heap de números
211     public void HPN()
212     {
213         int temp;  //variable temporal
214         int x = Arreglo_numeros.Length; //longitud de arreglo valores ingresados
215
216         for (int i = x - 1; i >= 1; i--)  //desde un valor menos de la longitud total decrementamos
217         {
218             intercambio(ref Arreglo, i, 1); // intercambio
219             temp = Arreglo_numeros[i]; // el elemento i del arreglo a temporal
220             Arreglo_numeros[i] = Arreglo_numeros[1]; //el elemento 1 a la posición i
221             Arreglo_numeros[1] = temp;//el que estaba en temporal a la posición 1
222             x--;
223         }
224     }
```

11. Para número máximo en heap, nos basamos en la fórmula para saber hijo izquierdo y derecho por la posición de los números en el array. Como nuestro arreglo comienza con i = 1 entonces usamos fórmula de hijo izquierdo en la posición 2*i y el hijo derecho en posición 2*i + 1.

    Esta sería la última parte del código digitado

```
226         //para número máximo en heap
227         public void Max_Num(int[] a, int x, int indice, ref Button[] botones)
228         {
229             int izquierdo = (indice) * 2;
230             int derecho = (indice) * 2 + 1;
231             int mayor = 0;
232 
233             if (izquierdo < x && a[izquierdo] > a[indice])
234             {
235                 mayor = izquierdo;
236             }
237             else
238                 mayor = indice;
239 
240             if (derecho < x && a[derecho] > a[mayor])
241             {
242                 mayor = derecho;
243             }
244 
245             if (mayor != indice) //si mayor es distinto de indice
246             {
247                 int temp = a[indice];  //valor con indice a temporal
248                 a[indice] = a[mayor];  //el mayor se almacena en posición del indice
249                 a[mayor] = temp; //el temporal se almacena en mayor
250 
251                 intercambio(ref Arreglo, mayor, indice); //se llama a intercambio
252                 Max_Num(a, x, mayor, ref botones); //llamada recursiva a Max_Num
253             }
254         }
```

Análisis de Resultados

Ejercicio 1:

Basados en el ejercicio proporcionado en el procedimiento se le pide modificarlo de forma que pueda elegir entre realizar un heap maximizante (como se hace en el ejemplo) y también se pueda hace un heap minimizante (padres menores que sus hijos, valor menor en la raíz del montículo) para elaborar un arreglo ascendente

Investigación Complementaria

1. Investigar cómo se puede incorporar la librería Visual Basic Power Pack en C# y qué utilidad tendría para nosotros.

2. En el ejercicio trabajado en esta guía ordenamiento ascendente y descendente ya sea por medio de heap maximizante o heap minimizante.