UADE

Trabajo Práctico Obligatorio

Programación II

Integrantes Grupo N°6

Esquivel, Candela - LU 1134110

Bari, Diego - LU 1133805

Bird, Maria Florencia - LU 1133832

Docentes

Wehbe, Ricardo Abraham

Barrera, Elizabeth Gabriela

Índice

Grupo N°6 Integrantes:

Esquivel, Candela - LU 1134110

Bari, Diego - LU 1133805

Bird, Maria Florencia - LU 1133832

Introducción

En el siguiente informe presentaremos estructuras de datos que nos permiten resolver ciertos tipo de problemas. Para facilitar la lectura lo dividimos en las siguientes secciones:

En la primera parte, explicaremos el heap, un tipo de estructura de datos en forma de árbol que tiene características especiales que pueden ser aprovechados para escenarios específicos, en particular cuando queremos acceder el mayor o menor elemento de un conjunto de datos.

En la segunda parte, utilizaremos el heap para implementar un tipo abstracto de datos (TDA) como es la cola con prioridad. Analizaremos el vínculo que los une y cuales son los beneficios de utilizar el heap en su implementación. En la parte del código, se observa la interfaz, los métodos internos y los externos solicitados, cada uno con su estimación de costos temporales.

En la última parte, desarrollamos un TDA denominado ListaEspecialTDA que se asemeja en comportamiento a las listas en Python. Se presentará la interfaz donde se explica cada método y en la implementación se podrá observar “desde adentro” cómo se obtiene un resultado similar al de Python utilizando una implementación dinámica de nodos doblemente enlazados.

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Heap

¿Qué es un heap?

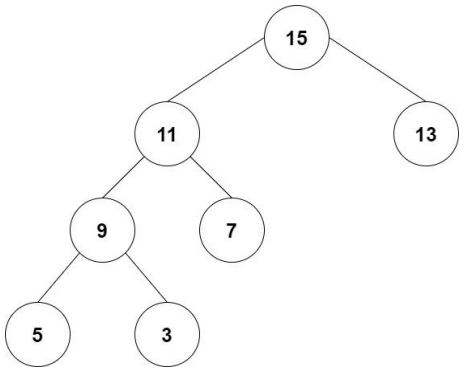
El heap es una estructura de datos basada en árboles binarios. El autor Mark Allen Weiss (2013) señala dos propiedades esenciales. La primera, la denomina propiedad estructural y se refiere a que el heap es un árbol binario completo con excepción del nivel inferior, es decir, no es necesario que posea todas las hojas por esta razón, se le suele decir árbol casi completo. En el caso de agregar nuevos nodos la inserción se realiza de arriba hacia abajo y de izquierda a derecha en la posición correspondiente. Un árbol completo garantiza una altura o profundidad que se calculará como:

$$h = log_2(n)$$

Donde h es la altura y n es la cantidad de nodos que tiene el heap. Más adelante se retomará esta ecuación para analizar los costos temporales de los métodos internos del heap.

En la Figura 1 se propone un ejemplo de un árbol binario que no cumple con la propiedad estructural.

Figura 1

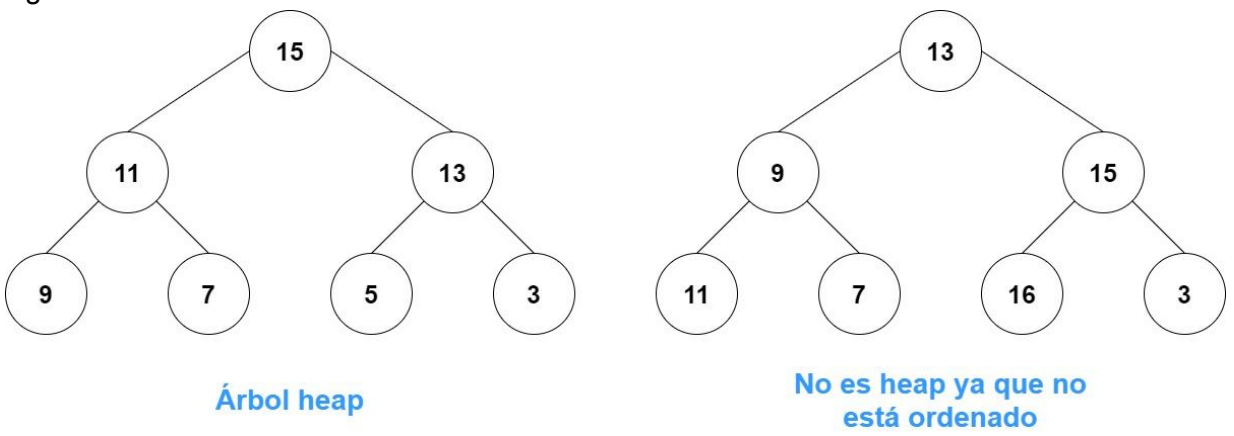


Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

La segunda propiedad, el autor, la denomina propiedad de ordenamiento. En la que se describe la relación que existe entre un nodo padre y su o sus nodos hijos. Para un heap min, el valor del nodo padre siempre es menor o igual que el valor de sus hijos y para un heap max, el valor del nodo padre siempre es mayor o igual al valor de sus hijos. Esta propiedad se debe conservar en todos los subárboles para que el heap sea consistente. De esta forma, el valor mínimo o máximo se ubicará siempre en la raíz según el heap que se decida utilizar.

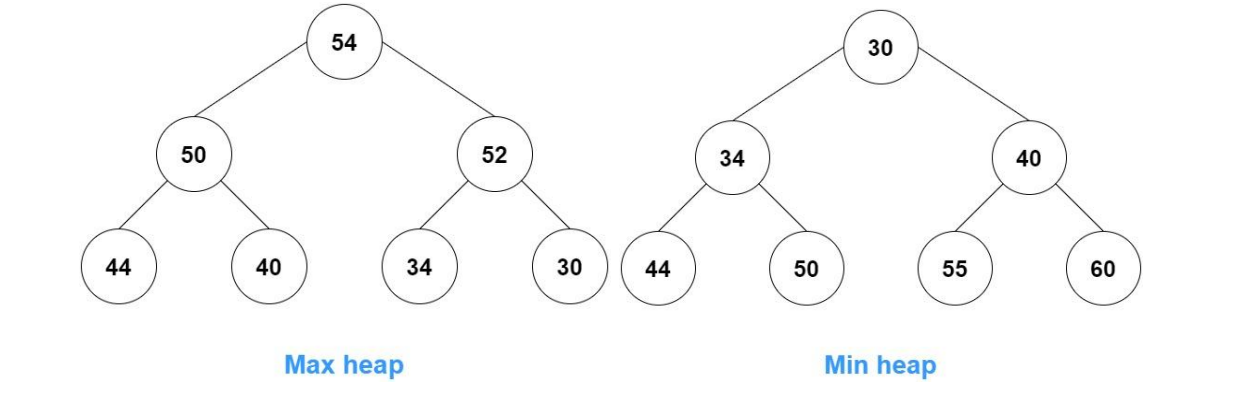
En la Figura 2 se observa la diferencia entre en heap y un árbol binario que no cumple con la propiedad de ordenamiento del heap.

Figura 2



En la Figura 3 se muestra un ejemplo de Max heap y de Min Heap, en el cual se observa el valor máximo o mínimo de la raíz según corresponda.

Figura 3



Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

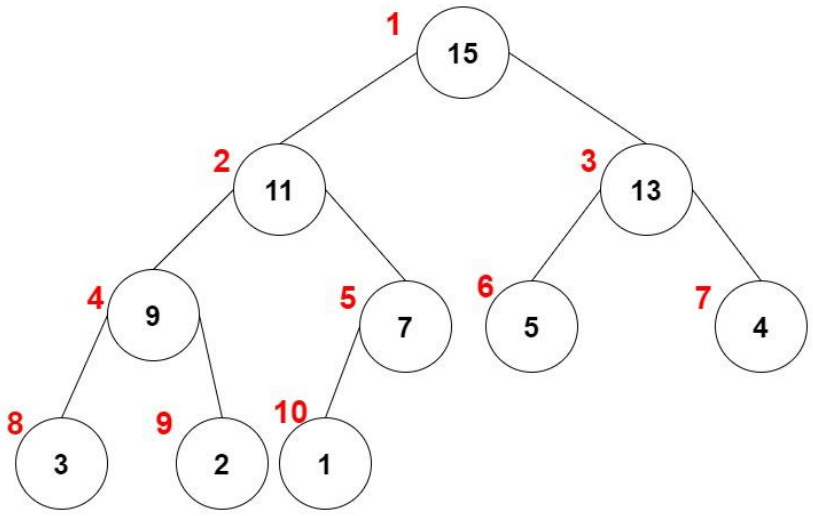
Conservación del heap

Debido a las propiedades mencionadas anteriormente sabemos que el heap nos provee el valor mínimo o máximo simplemente consultando la raíz. Esta característica resulta de relevancia al momento de implementar una cola con prioridad, ya que en este tipo abstracto de datos se requiere acceder al valor de mayor prioridad. Sin embargo, cabe preguntarse qué sucede cuando se agregan nuevos elementos al árbol, o cuando se eliminan elementos porque al modificar el heap se puede incurrir en una violacion de alguna o ambas propiedades. En consecuencia, al realizar alguna de estas operaciones, se debe realizar una verificación y adecuación del heap para conservar sus propiedades. En la sección de métodos, se describirá con mayor detalle cómo se realiza la adecuación luego de insertar o eliminar un elemento.

¿Cómo se implementa?

Si bien un heap es un tipo de árbol binario y, a priori, uno supone que se implementa con nodos conectados; se suele implementar mediante un arreglo según Jay Wengrow (2017). Partiendo de la propiedad estructural (árbol completo) se puede establecer una asociación entre la posición de un elemento en el árbol con el índice de un arreglo. En consecuencia, un heap puede ser un tipo abstracto de datos que utiliza un arreglo para su implementación. Para ilustrar este punto, veamos la Figura 4. Se observa en el arreglo los valores de cada elemento, en una posición que sigue un patrón. La raíz se ubica en el índice 1 del arreglo, luego es posible acceder a los valores de los hijos partiendo del padre, o partiendo de un hijo conocer el valor del padre. El patrón está determinado de la siguiente manera: el hijo izquierdo de un padre (en la posición i) es el valor que se encuentra en la posición 2i en el arreglo. Del mismo modo, el hijo derecho del padre está en la posición 2i+1 del arreglo. Para encontrar el padre de cualquier nodo en el árbol, podemos simplemente usar la división entera. Dado un nodo en la posición n del arreglo, el padre estará en la posición n/2.

Figura 4



Árbol binario completo

	15	11	13	9	7	5	4	3	2	1
	1	2	3	4	5	6	7	8	9	10

¿Cuál es su interfaz?

```
public interface HeapTDA {
        /**
        * @Tarea_Nombre: Insertar_Push
        * @Tarea_Descripción: Inserta elementos en el heap
        * @Parámetros: Recibe un tipo de dato int.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: No existen precondiciones.
        * @Postcondiciones: El heap mantiene su orden al haber insertado un elemento.
        * @Excepción: No tiene excepción.
        */
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
   public void push(int x) ;            //COSTO: O(log(n))

        /**
        * @Tarea_Nombre: Eliminar_Pop
        * @Tarea_Descripción: Elimina el primer elemento del Heap.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: El Heap no debe estar vacío.
        * @Postcondiciones: El heap mantiene su orden al haber eliminado el primer elemento.
        * @Excepción: No tiene excepción.
        */

 public void pop() ;                  //COSTO: O(log(n)).
        /**
        * @Tarea_Nombre: Retorna_Peek
        * @Tarea_Descripción: Retorna el primer elemento del Heap.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: Retorna un tipo de dato int que refiere al primer elemento del Heap.
        * @Precondiciones: El Heap no debe estar vacío.
        * @Postcondiciones: No tiene postcondición.
        * @Excepción: No tiene excepción.
        */
   public int peek();                   // COSTO: O(1).
   /**
        * @Tarea_Nombre: HeapVacío_isEmpty.
        * @Tarea_Descripción: Verifica si el Heap está vacío.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: Retorna un booleano. False si el Heap no está vacío y true si tiene elementos.
        * @Precondiciones: No hay precondición.
        * @Postcondiciones: No tiene postcondición.
        * @Excepción: No tiene excepción.
        */
   public boolean isEmpty() ;           // COSTO: O(1).
}
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

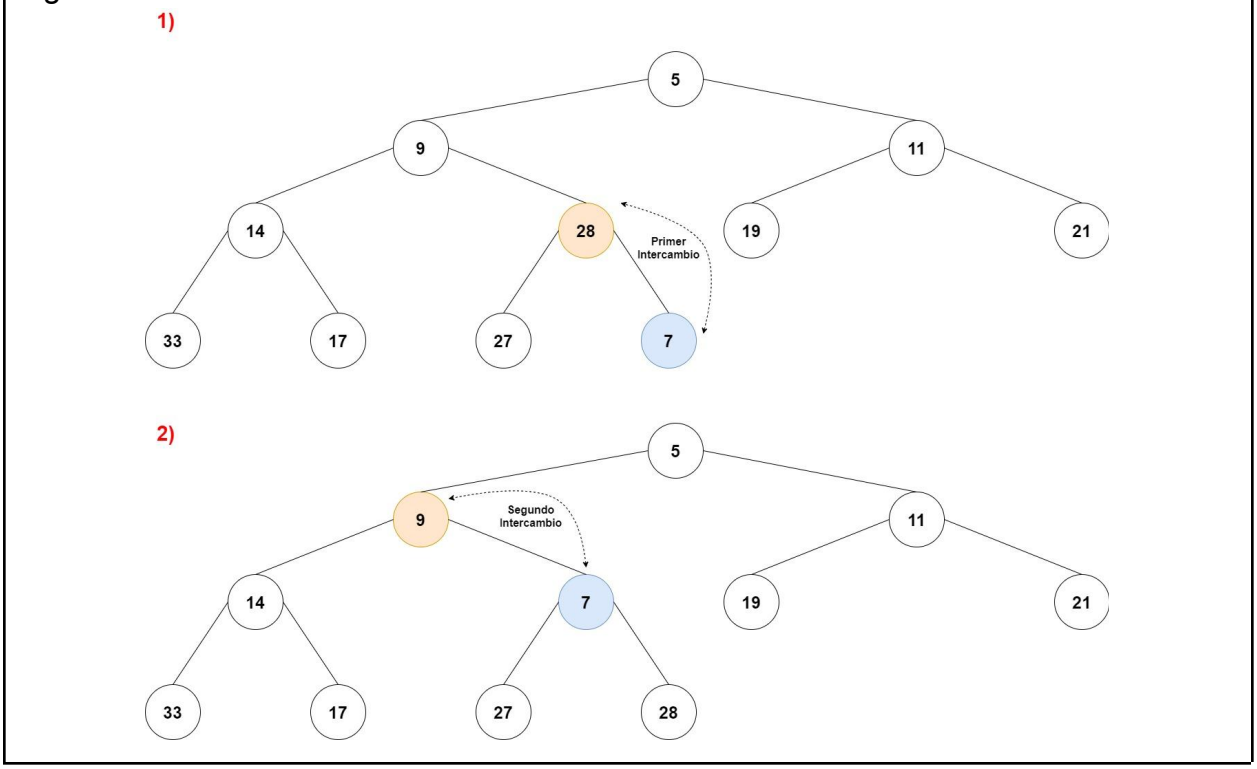
Implementación de heaps

Inserción de elementos en un heap:

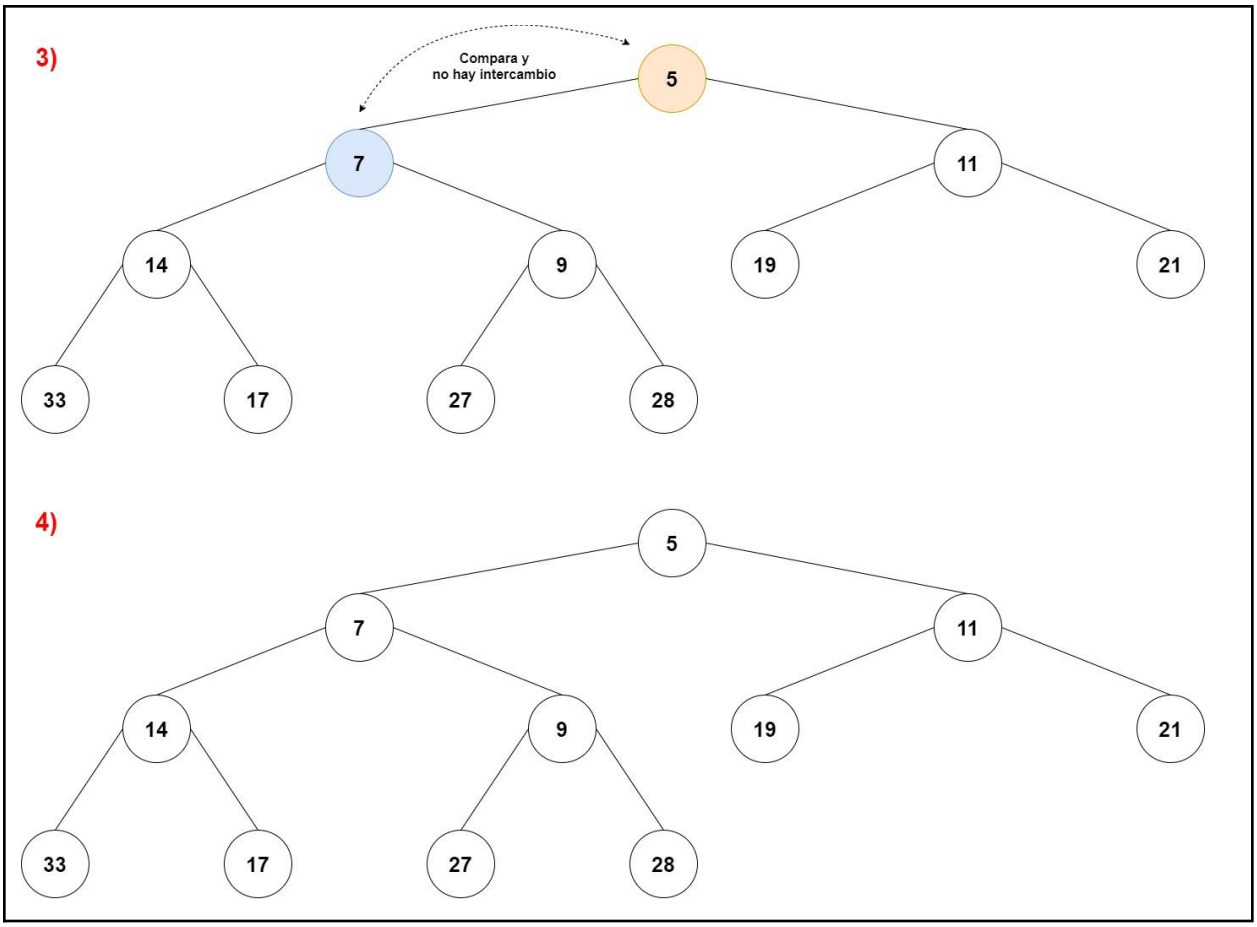
Para insertar un elemento a un heap min, contamos con un método denominado Push. El mismo consiste en agregar el elemento en un nuevo nodo al final del árbol. Como este nuevo nodo puede romper la consistencia del heap, es decir, el orden, cada vez que se inserta un nuevo elemento se deberá recorrer el heap desde la posición del elemento agregado hacia la raíz, En caso de que el elemento agregado sea menor que el padre, se los debe intercambiar y se debe realizar la comparación en los diferentes niveles del árbol hasta llegar a la raíz. En nuestra implementación este método lo denominamos heapifyUp().

En la Figura 5 se ilustra el procedimiento paso a paso de inserción de un nuevo elemento “7” y su ordenamiento posterior hasta recuperar la consistencia. Este método posee un costo temporal estimado en Log_2 (n), que recuerda a la ecuación mencionada para el cálculo de la altura del heap, ya que va a ser necesario (como máximo) realizar Log_2 (n) intercambios entre hijo-padre para mantener las propiedades del heap.

Figura 5



Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832



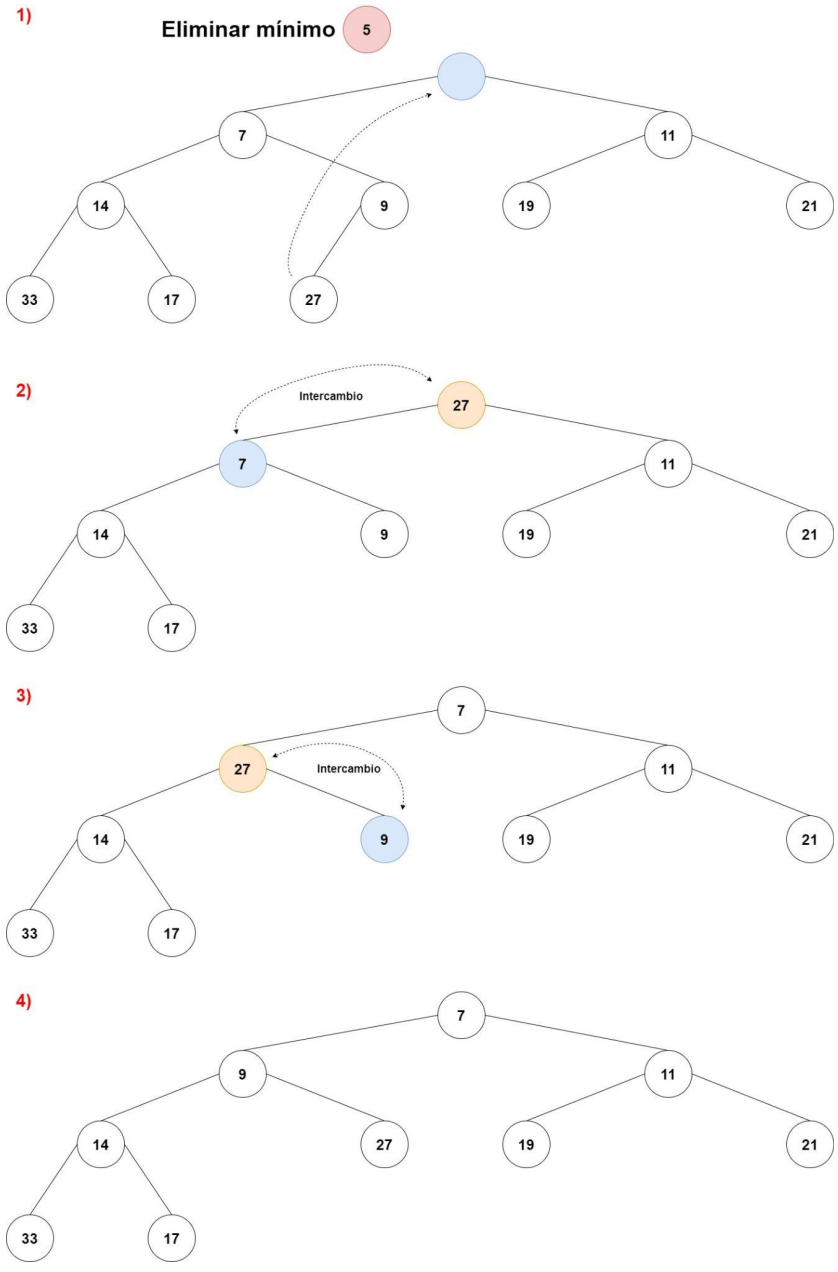
Eliminación de elementos de un heap:

En un heap, eliminamos el elemento que se encuentra en la raíz. Puede ser el máximo en un heap max o un mínimo en un heap min. En esta operación se elimina un elemento ubicado en la raíz y se lo reemplaza con el último elemento del árbol. Esta operación puede generar una inconsistencia en el heap, ya que puede romper el orden, como ocurre cuando se agrega un nuevo elemento. En este caso, como el problema se encuentra en la raíz, se debe comparar este elemento con sus hijos e intercambiarlo por el menor. Este recorrido se realiza desde arriba hacia abajo hasta que el heap recupera su consistencia. En nuestra implementación dentro del método Pop(), se encuentra el método heapifyDown() que es responsable de restablecer el orden del heap luego de la eliminación del valor que está raíz. Este método posee un costo temporal estimado en Log_2 (n), como el método de inserción, ya que va a ser necesario

(como máximo) realizar Log_2 (n) intercambios entre padre-hijo para mantener las propiedades del heap.

En la Figura 6 se observa como es el procedimiento que elimina el valor “5” y las sucesivas correcciones.

Figura 6



Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Implementación Max Heap:

```
import api.HeapTDA;

public class HeapMax implements HeapTDA {

   private int[] heap;
   private int size;

   public HeapMax(int capacidad) {
      heap = new int[capacidad+1];       //constante O(1)+
      heap[0] = Integer.MAX_VALUE;     //constante O(1)+
      size = 0;                                  //constante O(1)=
   }                                             //constante O(1)

   private void intercambio(int i, int j) {
      int tmp = heap[i];    //constante O(1) +
      heap[i] = heap[j];    //constante O(1) +
      heap[j] = tmp;        //constante O(1) =
   }                        //constante O(1)

   private void heapifyDown(int k) {
      int posMayor = k;                                  //constante O(1) +
      int posIzquierdo = 2*k;                            //constante O(1) +
      int posDerecho = 2*k + 1;                          //constante O(1) +

      if (posIzquierdo <= size && heap[posIzquierdo] > heap[posMayor]){  // (constante O(1) +
         posMayor = posIzquierdo;                              //constante O(1)) +
      }

      if (posDerecho <= size && heap[posDerecho] > heap[posMayor]) {   //(constante O(1) +
         posMayor = posDerecho;                                 //constante O(1)) +
      }

      if (posMayor != k) {                               //(constante O(1) +
         intercambio(k, posMayor);                       //constante O(1) +
         heapifyDown(posMayor);                          // logarítmico O(log(n)))=
      }
   }                                                     //logarítmico O(log(n))

   private void heapifyUp(int k) {
      while (heap[k] > heap[k/2] && (k/2)!=0) {         //n*(
         intercambio(k , k/2);                          //constante O(1) +
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
            k = k/2;                                        //constante O(1))=
        }
    }
    public void push(int x) {
        heap[++size] = x;       //constante O(1) +
        heapifyUp(size);        //logaritmico O(Log(n))=
    }                           //logaritmico O(Log(n))

    public void pop() {
        heap[1] = heap[size--];             //constante O(1) +
        heapifyDown(1);                     //logaritmico O(log(n))=

    }                                       //logaritmico O(log(n))

    public int peek() {
        return heap[1];       //constante O(1)
    }

    public boolean isEmpty() {
        return size == 0;     //constante O(1)
    }
}
```

Implementación Min Heap:

```
import api.HeapTDA;

public class HeapMin implements HeapTDA {

        private int[] heap;
        private int size;

        public HeapMin(int capacidad) {
                heap = new int[capacidad + 1];          //constante O(1)+
                heap[0] = Integer.MIN_VALUE;            //constante O(1)+
                size = 0;                               //constante O(1)=
        }                                              //constante O(1)

        private void intercambio(int i, int j) {
                int tmp = heap[i];                    //constante O(1)+
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
        heap[i] = heap[j];                  //constante O(1)+
        heap[j] = tmp;                      //constante O(1)=
    }                                       //constante O(1)

    private void heapifyDown(int k) {
        int posMenor = k;                   //constante O(1)+
        int posIzquierdo = 2 * k;           //constante O(1)+
        int posDerecho = 2 * k + 1;         //constante O(1)+

        if (posIzquierdo <= size && heap[posIzquierdo] < heap[posMenor]) {  //(constante O(1)+
            posMenor = posIzquierdo;                    //constante O(1))+
        }

        if (posDerecho <= size && heap[posDerecho] < heap[posMenor]) {  //(constante O(1)+
            posMenor = posDerecho;              //constante O(1))+
        }

        if (posMenor != k) {                                //(constante O(1)+
            intercambio(k, posMenor);                       //constante O(1)+
            heapifyDown(posMenor);                          //logaritmico O(log(n)))=
        }
    }                                                       //logaritmico O(log(n))

    private void heapifyUp(int k) {
        while (heap[k] < heap[k / 2] && (k / 2) != 0) {      //n*(
            intercambio(k, k / 2);                          //constante O(1)+
            k = k / 2;                                      //constante O(1))=
        }
    }                                                       //logaritmico O(log(n))

    public void push(int x) {
        heap[++size] = x;                       //constante O(1)+
        heapifyUp(size);                        //logaritmico O(log(n))=
    }                                           //logaritmico O(log(n))

    public void pop() {
        heap[1] = heap[size--];                 //constante O(1)+
        heapifyDown(1);                         //logaritmico O(log(n))=
    }                                           //logaritmico O(log(n))

    public int peek() {
        return heap[1];                     //constante O(1)
    }
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
        public boolean isEmpty() {
                return size == 0;                       //constante O(1)
        }
}
```

Identificación de un elemento dentro de un heap

Estrategia para un tipo Heap Min:

Se recibe un valor int por parámetro junto con el Heap y verificamos que el Heap no esté vacío. Luego comparamos el valor con el primer elemento del heap. Si son iguales, devolvemos true. Si son distintos, y mientras el valor sea menor al primer elemento, continuamos buscando. A medida que iteramos, apilamos en una pila los elementos que vamos eliminando del heap para no perderlos. Luego, volvemos a insertarlos en el Heap.

```
public static boolean valorPerteneceHeapMin(int valor, HeapTDA heap) {
        PilaTDA pila = new PilaEstatica();              //constante O(1)+
        pila.InicializarPila();                         //constante O(1)+
        boolean existe = false;                         //constante O(1)+

        while (!heap.isEmpty() && heap.peek() <= valor && existe != true ) {   //n*((constante O(1)+constante
O(1))
                if(heap.peek()==valor) {                        //constante O(1)+
                        existe=true;                            //constante O(1))+
                }else {
                        pila.Apilar(heap.peek());               //(constante O(1)+constante O(1))+
                        heap.pop();                             //logaritmico O(log(n)))
                }
        }
                while (!pila.PilaVacia()) {             //n*(constante O(1)+
                        heap.push(pila.Tope());         //(logaritmico O(log(n)) + constante O(1))+
                        pila.Desapilar();               //constante O(1)) =
                }
                return existe;
        }                                               //casi lineal O(n log(n))
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Estrategia para un tipo Heap Max:

Se recibe un valor int por parámetro junto con el Heap y verificamos que el Heap no esté vacío. Luego comparamos el valor con el primer elemento del heap. Si son iguales, devolvemos true. Si son distintos, y mientras el valor sea mayor al primer elemento, continuamos buscando. A medida que iteramos, apilamos en una pila los elementos que vamos eliminando del heap para no perderlos. Luego, volvemos a insertarlos en el Heap.

```
public static boolean valorPerteneceHeapMax(int valor, HeapTDA heap) {
        PilaTDA pila = new PilaEstatica();     //constante O(1)+
        pila.InicializarPila();                          //constante O(1)+
        boolean existe=false;                            //constante O(1)+

        while (!heap.isEmpty() && heap.peek() >= valor && existe !=true) {          //n*((constante
O(1)+constante O(1))
                if(heap.peek()==valor) {                                    //constante O(1)+
                        existe= true;                                       //constante O(1))+
                }else {
                        pila.Apilar(heap.peek());                           //(constante O(1)+constante O(1))+
                        heap.pop();                                         //logaritmico O(log(n))+
                }
        }

        while (!pila.PilaVacia()) {                             //n*(constante O(1)+
                heap.push(pila.Tope());                         //(logaritmico O(log(n)) + constante O(1))+
                pila.Desapilar();                               //constante O(1)) =
        }
        return existe;
}                                                               //casi lineal O(n log(n))
```

¿Es el heap una buena estructura para efectuar búsquedas?

Desde el punto de vista de la implementación del heap, buscar un elemento requiere que se recorra cada nodo del heap, ya que la condición de ordenamiento solo estipula reglas para la relación de padre-hijo pero no garantiza un ordenamiento como puede ser el del árbol AVL. Mientras que buscar un elemento en un heap el costo es lineal, en un árbol AVL el costo es del orden Log_2 (n). El heap no sería la mejor alternativa para realizar búsquedas. La “debilidad” de un heap para realizar búsquedas

tiene como ventaja de que solo ordena lo necesario, disminuyendo el costo al momento de insertar o eliminar elementos.

Respecto del usuario que solo puede acceder a la interfaz, deberá realizar una estrategia como la que planteamos anteriormente. El costo estimado es de $n.Log_2(n)$, probablemente para realizar una búsqueda le convenga utilizar otro tipo de estructura de datos más eficiente para su propósito.

Ordenar un arreglo utilizando heaps:

Estrategia para ordenar un arreglo:

Recibimos un arreglo y la cantidad de elementos del mismo por parámetro. Se inserta cada elemento del arreglo en el Heap y el mismo los va ordenando. Luego, mientras el Heap no quede vacío, pisamos los elementos del arreglo con los elementos que hemos ordenado previamente.

```
public static void ordenar (int[] arr , int n) {
        HeapTDA heap = new HeapMin(n);          //constante O(1)+

        for (int i=0 ; i<n; i++) {              //n*(
                heap.push(arr[i]);              //logaritmico O(log(n)))
        }

        int i=0;                                        //constante O(1)+

        while (!heap.isEmpty()) {                       //n*(constante O(1)+
                arr[i]=heap.peek();                     //constante O(1)+
                heap.pop();                             //logaritmico O(log(n))+
                i++;                                    //constante O(1))=
        }
}                                                       //casi lineal (n log(n))
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Determinar la equivalencia de dos heaps

Estrategia para determinar la equivalencia de dos heaps:

Se reciben dos heaps por parámetro. Mientras el heap no esté vacío, vamos comparando la raíz de ambos heaps. Si son distintas, determinamos que los heaps no son iguales, por ende devolvemos false. Si son iguales, apilamos dichos elementos en dos pilas respectivamente, eliminamos el primer elemento del heap y continuamos con la iteración hasta que un heap o ambos estén vacíos o encuentren elementos que sean distintos. Al final, en el caso de haber eliminado elementos del heap, se vuelve a insertar los elementos apilados en la pila a cada heap respectivamente.

```
public static boolean igualesHeap(HeapTDA heap , HeapTDA heap2) {
        PilaTDA pila1 = new PilaEstatica();                    //constante O(1)+
        pila1.InicializarPila();                               //constante O(1)+
        PilaTDA pila2 = new PilaEstatica();                    //constante O(1)+
        pila2.InicializarPila();                               //constante O(1)+
        boolean iguales=true;                                          //constante O(1)+

        while(!heap.isEmpty() && !heap2.isEmpty() && iguales==true ) {   //n*((constante O(1)+constante O(1))
                if(heap.peek()!=heap2.peek() )         {       //(constante O(1)+constante O(1))+
                        iguales= false;                        //constante O(1)+
                }else {
                        pila1.Apilar(heap.peek());             //(constante O(1)+constante O(1))+
                        heap.pop();                            //logaritmico O(log(n))+
                        pila2.Apilar(heap2.peek());            //(constante O(1)+constante O(1))+
                        heap2.pop();                           //logaritmico O(log(n)))
                }
        }
        if( (heap.isEmpty() && !heap2.isEmpty()) || (!heap.isEmpty() && heap2.isEmpty()) && iguales==true)
//constante O(1)+
                iguales= false;                                  //constante O(1)+
        while(!pila1.PilaVacia()) {                            //n*(constante O(1)+
                heap.push(pila1.Tope());                       //(logaritmico O(log(n))+constante O(1))+
                pila1.Desapilar();                             //constante O(1)+
                heap2.push(pila2.Tope());                      //(logaritmico O(log(n))+constante O(1))+
                pila2.Desapilar();                             //constante O(1))=
        }
        return iguales;
}                                                              //casi lineal (n log(n))
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

¿Son dos heaps equivalentes iguales (en el sentido de tener los mismos elementos en la misma secuencia)?

Partiendo de la definición dada en la consigna del TPO "... dos heaps son equivalentes si contienen los mismos elementos." Si bien se puede determinar la equivalencia. No es posible garantizar que tengan la misma secuencia, es decir, que ambos árboles sean idénticos. Por más que los elementos que vamos extrayendo del heap sean iguales, el ordenamiento interno puede ser distinto según se hayan insertado los elementos. En este sentido, habría que cambiar el foco y observar cuál fue la secuencia de inserción de elementos, en el caso de que eso sea posible. Además, como precondición deberíamos solicitar que el heap esté vacío, ya que si posee elementos nos encontraríamos en la situación original.

¿Para qué se utilizan los heaps?

Habiendo explicado el funcionamiento y las características del heap se puede comprender porque son útiles para la implementación de las colas con prioridad. La necesidad de acceso inmediato al valor prioritario aprovecha la naturaleza del heap ya que siempre este valor estará en la raíz del heap. Además, una vez que se haya cumplido con tal tarea y se decida acceder a la siguiente (es decir, se elimine dicha prioridad del heap), la siguiente tarea subirá hasta colocarse en la raíz. Como se indicó en la parte de costos temporales del heap, tanto la inserción como la eliminación de un elemento tienen un costo estimado logarítmico en función de la cantidad de elementos del heap. En la comparación propuesta por Wengrow entre los costos de implementar una cola con prioridad con un heap o un arreglo ordenado se observa los siguiente:

Operación	**Arreglo ordenado**	**Heap**
Insertar	O(n)	O(Log(n))
Eliminar	O(1)	O(Log(n))

Si bien puede parecer que comparando en ambas operaciones se compensan el en arreglo ordenado, es más conveniente utilizar una estructura de datos que siempre mantenga su costo como es el heap, en comparación con el arreglo ordenado que para

una operación es rápido y lento para la otra. No obstante queda a discreción del programador si encuentra razones válidas para utilizar el arreglo ordenado.

Interfaz de Cola con Prioridad:

```
public interface ColaPrioridadTDA {

        /**
        * @Tarea_Nombre: Inicializa_InicializarCola.
        * @Tarea_Descripción: Inicializa la cola para comenzar a trabajarla.
        * @Parámetros: Recibe un int con el tamaño de la cola.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: No tiene precondiciones.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: No tiene excepciones.
        */
        public void inicializarCola(int x);        //COSTO: O(1).

        /**
        * @Tarea_Nombre: Agregar_AcolarPrioridad.
        * @Tarea_Descripción: Agrega un elemento con su prioridad a una cola.
        * @Parámetros: Recibe un int con el valor y un int con la prioridad.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: La cola debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: No tiene excepciones.
        */
        public void acolarPrioridad(int x, int prioridad);   //COSTO: O(n).

        /**
        * @Tarea_Nombre: Eliminar_Desacolar.
        * @Tarea_Descripción: Elimina el elemento de mayor prioridad de una cola.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: La cola debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: Si la cola está vacía devuelve la excepción tipo RuntimeException con el mensaje
"La cola está vacía".
        */
        public void desacolar();      //COSTO: O(1).

        /**
```

```
        * @Tarea_Nombre: EstaVacía_colaVacia.
        * @Tarea_Descripción: Permite saber si una cola con prioridad está vacía.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: Retorna un booleano. Devuelve False si la cola no está vacía y True si lo está.
        * @Precondiciones: La cola debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: No tiene excepciones.
        */
        public boolean colaVacia(); //COSTO: O(1).

        /**
        * @Tarea_Nombre: RetornaPrimero_Primero.
        * @Tarea_Descripción: Permite conocer el elemento de mayor prioridad de una cola.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: Retorna un int con el valor del elemento de mayor prioridad.
        * @Precondiciones: La cola debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: Si la cola está vacía devuelve la excepción tipo RuntimeException con el mensaje
"La cola está vacía".
        */
        public int primero(); //COSTO: O(1).

        /**
        * @Tarea_Nombre: RetornaPrioridad_Prioridad.
        * @Tarea_Descripción: Permite conocer la prioridad del elemento de mayor prioridad.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: Retorna un int con la prioridad.
        * @Precondiciones: La cola debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: Si la cola está vacía devuelve la excepción tipo RuntimeException con el mensaje
"La cola está vacía".
        */
        public int prioridad(); //COSTO: O(1).
}
```

Implementación de Cola con Prioridad utilizando la estructura de heaps:

```
import api.ColaPrioridadTDA;

public class ColaPrioridad implements ColaPrioridadTDA {

        int [][] Matriz; // matriz que contiene prioridad en la posición 0 y valor en la posición 1.

        int size;

        public void inicializarCola(int capacidad) {
                Matriz= new int [capacidad+1][2];              //constante O(1)+
                size=0;                                        //constante O(1)=
        }                                                      //constante O(1)

          private void swap(int i, int j) {             // el valor y prioridad se modifican juntos
                        int tmp = Matriz[i][0];        //constante O(1)+
                        int tmp2 = Matriz[i][1];         //constante O(1)+

                        Matriz[i][0] = Matriz[j][0];       //constante O(1)+
                        Matriz[i][1] = Matriz[j][1];       //constante O(1)+

                        Matriz[j][0] = tmp;                //constante O(1)+
                        Matriz[j][1] = tmp2;               //constante O(1)=
           }                                               //constante O(1)

          private void heapifyUp(int k) {
              while (Matriz[k][0] > Matriz[k/2][0] && (k/2)!=0 ) {        //n*(
                 swap(k , k/2);                                          //constante O(1)+
                 k = k/2;                                                //constante O(1))=
              }
           }                                                           //logarítmico O(log(n))

        public void acolarPrioridad(int valor, int prioridad) {
                Matriz[++size][0]=prioridad;                   //constante O(1)+
                Matriz[size][1]=valor;                         //constante O(1)+
                heapifyUp(size);                               //logaritmico O(log(n))=
        }                                                      //logaritmico O(log(n))

         private void heapifyDown(int k) {
              int posMayor = k;                                                  //constante O(1) +
              int posIzquierdo = 2*k;                                            //constante O(1) +
              int posDerecho = 2*k + 1;                                          //constante O(1) +
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
        if (posIzquierdo <= size && Matriz[posIzquierdo][0] > Matriz[posMayor][0] ){ // constante O(1) +
                posMayor = posIzquierdo;                //constante O(1) +
        }
        if (posDerecho <= size && Matriz[posDerecho][0] > Matriz[posMayor][0] ) { //constante O(1) +
                posMayor = posDerecho;                  //constante O(1) +
        }
        if (posMayor != k ) {                           //constante O(1) +
                swap(k, posMayor);                      //constante O(1) +
                heapifyDown(posMayor);                  //logaritmico O(log(n))=
        }

    }                                                   //logaritmico O(log(n))

    public void desacolar() {
            if(colaVacia()) {                                                   //constante O(1) +
                    throw new RuntimeException("La cola está vacía");           //constante O(1) +
            }
            Matriz[1][0] = Matriz[size][0];                                     //constante O(1) +
            Matriz[1][1]=Matriz[size][1];                                       //constante O(1) +
            size--;                                                             //constante O(1) +
            heapifyDown(1);                                                     //logaritmico O(log(n))=
    }                                                                           //logaritmico O(log(n))

    public boolean colaVacia() {
            return size==0 ;                                //constante O(1)
    }

    public int primero() {
            if(colaVacia()) {                                                   //constante O(1) +
                    throw new RuntimeException("La cola está vacía");           //constante O(1)=
            }else {
                    return Matriz[1][1];
            }                                                                   //constante O(1)
    }

    public int prioridad() {

            if(colaVacia()) {                                                   //constante O(1) +
                    throw new RuntimeException("La cola está vacía");           //constante O(1)=
            }else {
                    return Matriz[1][0];
            }
    }                                                                           //constante O(1)
}
```

Lista especial TDA

Interfaz de ListaEspecialTDA

```
public interface ListaEspecialTDA {

	public static class NodoDoble {
		public int info;
		public NodoDoble anterior;
		public NodoDoble siguiente;
	}

	/**
	* @Tarea_Nombre: Inicializa_InicializarLista.
	* @Tarea_Descripción: Inicializa la lista para comenzar a trabajarla.
	* @Parámetros: No recibe parámetros.
	* @Devuelve: No retorna ya que es de tipo void.
	* @Precondiciones: No tiene precondición.
	* @Postcondiciones: No tiene postcondiciones.
	* @Excepción: No tiene excepciones.
	*/
	public void inicializarLista();			//COSTO: O(1).

	/**
	* @Tarea_Nombre: Tamaño_Size.
	* @Tarea_Descripción: Devuelve el tamaño de la lista, es decir, la cantidad de
elementos que tiene la lista.
	* @Parámetros: No recibe parámetros.
	* @Devuelve: Devuelve un int con la cantidad de elementos de la lista.
	* @Precondiciones: La lista debe estar inicializada.
	* @Postcondiciones: No tiene postcondiciones.
	* @Excepción: No tiene excepciones.
	*/
	public int size();				//COSTO: O(1).

	/**
	* @Tarea_Nombre: Agregar_Append.
	* @Tarea_Descripción: Agrega elementos al final de la lista.
	* @Parámetros: Recibe un int con el valor a agregar.
	* @Devuelve: No retorna ya que es de tipo void.
	* @Precondiciones: La lista debe estar inicializada.
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
* @Postcondiciones: Incrementa uno el tamaño de la lista.
* @Excepción: No tiene excepciones.
*/
public abstract void append(int valor);          //COSTO: O(1).


/**
* @Tarea_Nombre: VacíaLista_Clear.
* @Tarea_Descripción: Elimina todos los elementos de la lista.
* @Parámetros: No recibe parámetros.
* @Devuelve: No retorna ya que es de tipo void.
* @Precondiciones: La lista debe estar inicializada.
* @Postcondiciones: El tamaño de la lista es igual a 0.
* @Excepción: No tiene excepciones.
*/
public abstract void clear();                    //COSTO: O(1).

/**
* @Tarea_Nombre: Copiar_Copy.
* @Tarea_Descripción: Copia los elementos de una lista en una lista nueva.
* @Parámetros: No recibe parámetros.
* @Devuelve: Retorna la copia de la lista.
* @Precondiciones: No tiene precondiciones.
* @Postcondiciones: No hay postcondiciones.
* @Excepción: No tiene excepciones.
*/
public abstract ListaEspecialTDA copy();          //COSTO: O(n)

/**
* @Tarea_Nombre: Contar_Count.
* @Tarea_Descripción: Cuenta la cantidad de apariciones que hay de un valor en la lista.
* @Parámetros: Recibe un int con el valor.
* @Devuelve: Retorna un int con la cantidad de apariciones de ese valor en la lista.
* @Precondiciones: La lista debe estar inicializada.
* @Postcondiciones: No hay postcondiciones.
* @Excepción: No tiene excepciones.
*/
public abstract int count(int valor);             //COSTO: O(n).

/**
* @Tarea_Nombre: AgregaLista_Extend.
* @Tarea_Descripción: Agrega la lista recibida por parámetro al final de la lista original.
* @Parámetros: Recibe una lista de tipo ListaEspecialTDA.
* @Devuelve: No retorna ya que es de tipo void.
```

```
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: Incrementa el tamaño de la lista original.
        * @Excepción: No tiene excepciones.
        */
        public abstract void extend(ListaEspecialTDA lista);        //COSTO: O(n).

        /**
        * @Tarea_Nombre: Indice_Index.
        * @Tarea_Descripción: Busca en qué posición de la lista se encuentra el elemento valor;
        * si aparece varias veces, devuelve la primera.
        * @Parámetros: Recibe un int con el valor a buscar.
        * @Devuelve: Retorna la posición del valor en la lista.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: No tiene postcondición.
        * @Excepción: Cuando no existe el elemento lanza una excepción de tipo
RuntimeException con el mensaje "No existe en la lista".
        */
        public abstract int index(int valor);                    //COSTO: O(n).

        /**
        * @Tarea_Nombre: EliminaIndice_Pop.
        * @Tarea_Descripción: Elimina el elemento de la posición enviada por parámetro.
Permite la utilización de índices negativos.
        * @Parámetros: Recibe un int con la posición del valor a eliminar.
        * @Devuelve: Devuelve el valor eliminado.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: Decrementa uno el tamaño de la lista.
        * @Excepción: En caso de sobrepasar el tamaño de la lista, lanza una excepción de tipo
RuntimeException con el mensaje "No existe en la lista ese índice".
        */
        public abstract int pop(int index);                //COSTO: O(n).

        /**
        * @Tarea_Nombre: EliminaValor_Remove.
        * @Tarea_Descripción: Elimina el elemento valor de la lista; si aparece varias veces,
elimina la primera.
        * @Parámetros: Recibe un int con el valor a eliminar.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: Decrementa uno el tamaño de la lista.
        * @Excepción: En caso de no existir el valor a eliminar, lanza una excepción de tipo
RuntimeException con el mensaje "No existe en la lista ese valor".
        */
        public abstract void remove(int valor);            //COSTO: O(n).
```

```
        /**
        * @Tarea_Nombre: Invertir_Reverse.
        * @Tarea_Descripción: Invierte la lista.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: La lista está invertida.
        * @Excepción: No tiene excepciones.
        */
        public abstract void reverse();                    //COSTO: O(n).

        /**
        * @Tarea_Nombre: Ordenar_Sort.
        * @Tarea_Descripción: Ordena los elementos de la lista de menor a mayor según el
valor de dichos elementos.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: La lista se encuentra ordenada.
        * @Excepción: No tiene excepciones.
        */
        public abstract void sort();                    //COSTO: O(n^2).

        /**
        * @Tarea_Nombre: EstaOrdenado_isSorted.
        * @Tarea_Descripción: Valida si la lista está ordenada tanto de forma creciente como
decreciente.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: No retorna ya que es de tipo void.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: No tiene excepciones.
        */
        public abstract boolean isSorted();             //COSTO: O(n).

        /**
        * @Tarea_Nombre: Buscar_binarySearch.
        * @Tarea_Descripción: Busca el valor ingresado mediante una búsqueda binaria.
        * @Parámetros: Recibe un int con el valor a buscar.
        * @Devuelve: Retorna un boolean. False si no encontró el elemento y True si lo
encontró.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
```

```
        * @Excepción: Si la lista no está ordenada (creciente o decrecientemente), lanza una
excepción de tipo RuntimeException con el mensaje "Lista no ordenada".
        */
        public abstract boolean binarySearch(int valor);          //COSTO: O(n^2)

        /**
        * @Tarea_Nombre: Imprime_Print.
        * @Tarea_Descripción: Imprime por pantalla los elementos de la lista.
        * @Parámetros: No recibe parámetros.
        * @Devuelve:No retorna ya que es de tipo void.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: No tiene excepciones.
        */
        public void print();                    //COSTO: O(n).

        /**
        * @Tarea_Nombre: DevuelveValor_Get.
        * @Tarea_Descripción: Devuelve el valor del índice pasado por parámetro.
        * @Parámetros: Recibe un int con el índice.
        * @Devuelve:Retorna un int con el valor.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: No tiene excepciones.
        */
        public int get(int index);              //COSTO: O(n).

        /**
        * @Tarea_Nombre:DevuelvePrimero_Primero.
        * @Tarea_Descripción: Devuelve el primer elemento de la lista.
        * @Parámetros: No recibe parámetros.
        * @Devuelve: Retorna un elemento tipo NodoDoble.
        * @Precondiciones: La lista debe estar inicializada.
        * @Postcondiciones: No tiene postcondiciones.
        * @Excepción: No tiene excepciones.
        */
        public NodoDoble primero();             //COSTO: O(1).
}
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Implementación de ListaEspecialTDA

```
import api.ListaEspecialTDA;


public class ListaEspecial implements ListaEspecialTDA {

	int pos;
	NodoDoble inicio;
	NodoDoble ultimo;

	public void inicializarLista() {
		inicio = null;		//constante O(1)+
		ultimo = null;		//constante O(1)+
		pos = 0;			//constante O(1)=
	}					//constante O(1)

	private NodoDoble crearNodo(int x, NodoDoble anterior, NodoDoble siguiente) {
		NodoDoble aux = new NodoDoble();		//constante O(1)+
		aux.info = x;					//constante O(1)+
		aux.anterior = anterior;			//constante O(1)+
		aux.siguiente = siguiente;			//constante O(1)=
		return aux;
	}						//constante O(1)

	public void print() {
		if (pos != 0) {					//constante O(1)+
			NodoDoble aux = new NodoDoble();	//constante O(1)+
			aux = inicio;				//constante O(1)+
			System.out.print("[");			//constante O(1)+
			while (aux != null) {			//n*(
				System.out.print(aux.info);	//constante O(1)+
				if (aux.siguiente != null) {	//constante O(1)+
					System.out.print(" , ");//constante O(1)+
				}
				aux = aux.siguiente;		//constante O(1))+
			}
			System.out.print("]");			//constante O(1)+
		}else {
			System.out.print("[ ]");			//constante O(1)+
		}
		System.out.println();				//constante O(1)=
	}						//lineal O(n)
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
public int size() {
        return pos;                     //constante O(1)
}

public void append(int valor) {
        if (pos == 0) {                                                 //constante O(1)+
                inicio = ultimo = crearNodo(valor, null, null);         //constante O(1)+
        } else {
                NodoDoble aux = new NodoDoble();                        //constante O(1)+
                aux = crearNodo(valor, ultimo, null);                   //constante O(1)+
                ultimo.siguiente = aux;                                 //constante O(1)+
                ultimo = aux;                                           //constante O(1)+
        }
        pos++;                                                          //constante O(1)=
}                                                                       //constante O(1)

public void clear() {
        ultimo = inicio = null;                 //constante O(1)+
        pos = 0;                                //constante O(1)=
}                                               //constante O(1)

public ListaEspecialTDA copy() {

        ListaEspecialTDA listaCopy = new ListaEspecial();               //constante O(1)+
        listaCopy.inicializarLista();                                   //constante O(1)+

        NodoDoble aux = new NodoDoble();                                //constante O(1)+

        aux = inicio;                                                   //constante O(1)+
        while (aux != null) {                                           //n*(
                int valor = aux.info;                                   //constante O(1)+
                listaCopy.append(valor);                                //constante O(1)+
                aux = aux.siguiente;                                    //constante O(1)=
        }
        return listaCopy;
}                                                                       //lineal O(n)

public int count(int valor) {
        int contador = 0;                               //constante O(1)+
        NodoDoble aux = new NodoDoble();                //constante O(1)+
        aux = inicio;                                   //constante O(1)+
        while (aux != null) {                           //n*(
                int valorAux = aux.info;                //constante O(1)+
                if (valorAux == valor)                  //constante O(1)+
```

```
                        contador++;                         //constante O(1)+
                    aux = aux.siguiente;                    //constante O(1)=
                }
                return contador;
        }                                                   //lineal O(n)

        public void extend(ListaEspecialTDA lista) {
                pos += lista.size();                        //constante O(1)+
                ultimo.siguiente = lista.primero();         //constante O(1)+
                lista.primero().anterior = ultimo;          //constante O(1)+

                NodoDoble aux = new NodoDoble();            //constante O(1)+
                aux = ultimo;                               //constante O(1)+

                //Se puede hacer que este método sea constante agregando un método Ultimo() en la
interfaz
                while (aux.siguiente != null) {             //n*(
                        aux=aux.siguiente;                  //constante O(1))+
                }
                ultimo=aux;                                 //constante O(1)=
        }                                                   //lineal O(n)

        public int index(int valor) {
                int posicion = -1;                          //constante O(1)+
                int contador = 0;                           //constante O(1)+

                NodoDoble aux = new NodoDoble();            //constante O(1)+
                aux = inicio;                               //constante O(1)+

                while (aux != null) {                               //n*(
                        int valorAux = aux.info;                    //constante O(1)+
                        if (valorAux == valor) {                    //constante O(1)+
                                posicion = contador;                //constante O(1)+
                                return posicion;                    //constante O(1)+
                        }
                        contador++;                                 //constante O(1)+
                        aux = aux.siguiente;                        //constante O(1))+
                }

                if (posicion == -1)                                 //constante O(1)+
                        throw new RuntimeException("No existe en la lista");   //constante O(1)=

                return posicion;
        }                                                   //lineal O(n)
```

```
public int pop(int index) {
        int valor = 0;                                          //constante O(1)+

        // primera posicion
        if (index == 0 || index == (size() * -1)) {             //constante O(1)+
                valor = inicio.info;                            //constante O(1)+
                inicio = inicio.siguiente;                      //constante O(1)+
                inicio.anterior = null;                         //constante O(1)+
                pos--;                                          //constante O(1)+

        // ultima posicion
        } else if (index == (size() - 1) || index == -1) {      //constante O(1)+
                valor = ultimo.info;                            //constante O(1)+
                ultimo = ultimo.anterior;                       //constante O(1)+
                ultimo.siguiente = null;                        //constante O(1)+
                pos--;                                          //constante O(1)+

        // indice positivo
        } else if (0 < index && index < size() - 1) {           //constante O(1)+
                NodoDoble aux = new NodoDoble();                //constante O(1)+
                aux = inicio.siguiente;                         //constante O(1)+
                int contadorPositivo = 1;                       //constante O(1)+
                while (aux != ultimo) {                         //n*(
                        if (index == contadorPositivo) {                //constante O(1)+
                                valor = aux.info;                       //constante O(1)+
                                aux.anterior.siguiente = aux.siguiente;  //constante O(1)+
                                aux.siguiente.anterior = aux.anterior;   //constante O(1)+
                        }
                        contadorPositivo++;                             //constante O(1)+
                        aux = aux.siguiente;                            //constante O(1))+
        }
        pos--;                                                  //constante O(1)+

        // indice negativo
        } else if ((size() * -1) < index && index < -1) {               //constante O(1)+
                NodoDoble aux = new NodoDoble();                        //constante O(1)+
                aux = ultimo.anterior;                                  //constante O(1)+
                int contadorNegativo = -2;                              //constante O(1)+
                while (aux != inicio) {                                 //n*(
                        if (index == contadorNegativo) {                //constante O(1)+
                                valor = aux.info;                       //constante O(1)+
                                aux.anterior.siguiente = aux.siguiente;  //constante O(1)+
                                aux.siguiente.anterior = aux.anterior;   //constante O(1)+
                        }
```

```
                    contadorNegativo--;                         //constante O(1)+
                    aux = aux.anterior;                         //constante O(1))+
                }
                pos--;                                          //constante O(1)+
            } else {
                throw new RuntimeException("No existe en la lista ese índice"); //constante O(1)=
            }
            return valor;
        }                                                       //lineal O(n)


        public void remove(int valor) {

            boolean removio=false;                              //constante O(1)+

            if(inicio.info==valor) {                            //constante O(1)+
                inicio=inicio.siguiente;                        //constante O(1)+
                inicio.anterior=null;                           //constante O(1)+
                removio=true;                                   //constante O(1)+
            }else if (ultimo.info==valor) {                     //constante O(1)+
                ultimo=ultimo.anterior;                         //constante O(1)+
                ultimo.siguiente=null;                          //constante O(1)+
                removio=true;                                   //constante O(1)+
            }
            NodoDoble aux = new NodoDoble();                    //constante O(1)+
            aux=inicio.siguiente;                               //constante O(1)+

            while(aux!=ultimo && removio==false) {              //n*(
                if (valor==aux.info) {                          //constante O(1)+
                    aux.anterior.siguiente=aux.siguiente;       //constante O(1)+
                    aux.siguiente.anterior=aux.anterior;        //constante O(1)+
                    removio=true;                               //constante O(1)+
                        }
                aux=aux.siguiente;                              //constante O(1))+
                 }

            if(removio==false) {                                //constante O(1)+
                throw new RuntimeException("No existe en la lista ese valor");  //constante O(1)=
            }

        }                                                       //lineal O(n)

        public void reverse() {
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
        NodoDoble aux = new NodoDoble();                        //constante O(1)+
        aux=ultimo;                                             //constante O(1)+

        NodoDoble inicio2 = new NodoDoble();                    //constante O(1)+
        inicio2=crearNodo(ultimo.info,null,null);               //constante O(1)+

        NodoDoble aux2 = new NodoDoble();                       //constante O(1)+
        aux2=inicio2;                                           //constante O(1)+

        while(aux.anterior!=null) {                             //n*(
            aux=aux.anterior;                                   //constante O(1)+
            aux2.siguiente=crearNodo(aux.info,aux2,null);       //constante O(1)+
            aux2=aux2.siguiente;                                //constante O(1))+
        }
        inicio=inicio2;                                         //constante O(1)+
    }                                                           //lineal O(n)


    public void sort() {
        NodoDoble aux = new NodoDoble();                        //constante O(1)+
        NodoDoble auxInicio = new NodoDoble();                  //constante O(1)+

        aux = auxInicio = inicio;                               //constante O(1)+

        if (pos != 0) {                                         //constante O(1)+
            while (auxInicio.siguiente != null) { // revisar esto       //n*(
                while (aux.siguiente != null) {                             //n*(
                    if (aux.info > aux.siguiente.info) {        //constante O(1)+
                        int temp = aux.info;                    //constante O(1)+
                        aux.info = aux.siguiente.info;                  //constante O(1)+
                        aux.siguiente.info = temp;                      //constante O(1)+
                    }
                    aux = aux.siguiente;                        //constante O(1)+
                }
                auxInicio = auxInicio.siguiente;                //constante O(1)+
                aux = inicio;                                   //constante O(1)=
            }
        }
    }                                                           //cuadrático O(n^2)


    public boolean isSorted() {
        if (pos == 0)                                           //constante O(1)+
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
                    throw new RuntimeException("Lista vacía");          //constante O(1)+
                else if (pos == 1)                                      //constante O(1)+
                    return true;
                else {
                    NodoDoble aux = new NodoDoble();                    //constante O(1)+
                    aux = inicio;                                       //constante O(1)+
                    if (aux.info <= aux.siguiente.info) {               //constante O(1)+
                        while ((aux.info <= aux.siguiente.info) && (aux.siguiente.siguiente !=
null)) {    //n*(
                            aux = aux.siguiente;                        //constante O(1))+

                            }
                        if ((aux.info <= aux.siguiente.info) && (aux.siguiente.siguiente == null))
{  //constante O(1)+
                            return true;
                        }
                    } else if (aux.info >= aux.siguiente.info) {        //constante O(1)+
                        while ((aux.info >= aux.siguiente.info) && (aux.siguiente.siguiente !=
null)) {        //n*(
                            aux = aux.siguiente;                        //constante O(1))+
                        }
                        if ((aux.info >= aux.siguiente.info) && (aux.siguiente.siguiente == null))
{              //constante O(1)=
                            return true;
                        }
                    }
                }
                return false;
            }                                                           //lineal O(n)

            public boolean binarySearch(int valor) {
                if (isSorted()) {                                               //lineal O(n)+
                    NodoDoble inf = new NodoDoble();                    //constante O(1)+
                    NodoDoble sup = new NodoDoble();                    //constante O(1)+
                    NodoDoble aux = new NodoDoble();                    //constante O(1)+
                    int posicion = (size()) / 2;                        //constante O(1)+
                    inf = inicio;                                       //constante O(1)+
                    sup = ultimo;                                       //constante O(1)+
                    aux = get_nodo(posicion);                           //lineal O(n)+

                    if (inf.info == valor || sup.info == valor)         //constante O(1)+
                        return true;
                    while (inf.info <= valor && valor <= sup.info) {            //log n*(
                        if (valor < aux.info) {                         //constante O(1)+
```

```
                    sup = aux.anterior;                         //constante O(1)+
                    aux = get_nodo(posicion / 2);               //lineal O(n)+
                } else if (valor > aux.info) {                  //constante O(1)+
                    inf = aux.siguiente;                        //constante O(1)+
                    aux = get_nodo(posicion + posicion / 2);    //lineal O(n)+
                } else if (valor == aux.info) {                 //constante O(1))+
                    return true;
                }
            }
            return false;
        } else {
            throw new RuntimeException("Lista no ordenada");    //constante O(1)=
        }
    }                                                           //casi lineal O(n log(n))
    private NodoDoble get_nodo(int posicion) {
        NodoDoble aux = new NodoDoble();                //constante O(1)+
        aux = inicio;                                   //constante O(1)+
        while (posicion != 0) {                         //n*(
            posicion--;                                 //constante O(1)+
            aux = aux.siguiente;                        //constante O(1)=
        }
        return aux;
    }                                                   //lineal O(n)
    public int get(int posicion) {
        if(posicion>=size()) {                          //constante O(1)+
            throw new RuntimeException("Índice fuera de rango");    //constante O(1)=
        }else {

            NodoDoble aux = new NodoDoble();            //constante O(1)+
            aux = inicio;                               //constante O(1)+
            while (posicion != 0) {                     //n*(
                posicion--;                             //constante O(1)+
                aux = aux.siguiente;                    //constante O(1)=
            }
            return aux.info;
        }
    }                                                   //lineal O(n)

    public  NodoDoble primero() {
        return inicio;                                  //constante O(1)
    }
}
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Validar si una ListaEspecialTDA es capicúa:

Estrategia para determinar si una ListaEspecialTDA es capicúa:

Recorremos la lista recibida por parámetro utilizando dos índices. El primero recorre la lista desde la posición 0 y el segundo recorre la lista desde la última posición. En cada iteración se van comparando ambos valores y si son diferentes, deja de comparar y retorna False. Si no, retorna true. Ambos recorridos de los índices se realizan hasta la mitad de la lista.

```
public static boolean esCapicua(ListaEspecialTDA lista) {
        int i = 0;
                                        //constante O(1)+
        int j = lista.size() - 1;               //constante O(1)+
        while (lista.get(i) == lista.get(j) && i < (lista.size() / 2)) {  //n*((lineal O(n)+lineal (n))+
                i++;                                                    //constante O(1)+
                j--;                                                    //constante O(1))+
        }
        if (i == (lista.size() / 2)) {                                  //constante O(1)=
                return true;
        }

        return false;
    }                                                                   //cuadrático O(n^2)
```

Eliminar valores repetidos de una ListaEspecialTDA:

Estrategia para eliminar valores repetidos:

A medida que recorremos la lista pasada por parámetro, consultamos si cada uno de sus elementos pertenece a un conjunto. Si pertenece, eliminamos el elemento de la lista. Si no pertenece, lo agregamos al conjunto.

```
public static void limpiarRepetidos(ListaEspecialTDA lista) {
        ConjuntoTDA repetidos = new ConjuntoDin();              //constante O(1)+
        repetidos.InicializarConjunto();                        //constante O(1)+
        int i = 0;                                              //constante O(1)+
        while (i < lista.size()) {                      //n*(constante O(1)+
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

```
                if (!repetidos.Pertenece(lista.get(i))) {     //(lineal O(n)+lineal O(n))+
                        repetidos.Agregar(lista.get(i));  //(lineal O(n)+lineal O(n))+
                        i++;                                        //constante O(1))+
                    } else {
                        lista.pop(i);                               //constante O(1)=
                    }
            }
    }                                                               //cuadrático O(n^2)
```

Normalizar valores de una ListaEspecialTDA:

Estrategia para normalizar valores de una ListaEspecialTDA:

Se recorre la lista enviada por parámetro y vamos sumando sus elementos hasta tener el total. Dividimos cada valor de la lista por la suma total y obtenemos el valor normalizado. Luego retornamos un arreglo con dichos valores normalizados.

```
public static float[] normalizar(ListaEspecialTDA lista) {

        int cont = 0;                                       //constante O(1)+
        float sum = 0;                                  //constante O(1)+
        float[] arreglo = new float[lista.size()];          //constante O(1)+

        while (cont < lista.size()) {                       //n*(constante O(1)+
            sum = sum + lista.get(cont);            //lineal O(n)+
            cont++;                                         //constante O(1))+
        }
        cont = 0;                                           //constante O(1)+

        while (cont < lista.size()) {                       //n*(constante O(1)+
            float valor = lista.get(cont) / sum;            //lineal O(n)+
            arreglo[cont] = valor;                  //constante O(1)+
            cont++;                                         //constante O(1))=
        }
        return arreglo;
    }                                                       //cuadrático O(n^2)
```

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Conclusiones

Para realizar este trabajo se profundizó en los temas indicados en las consignas del TPO, para luego elaborar las respectivas interfaces e implementaciones para el heap, ColaConPrioridad y ListaEspecial.

El heap se implementó un heap max y un heap min, se tomó la decisión de presentar ambos casos para que el usuario opte según su necesidad. Se mencionó las propiedades que lo tornan atractivo en ciertos escenarios debido a su coste logarítmico para insertar y eliminar elementos. Asimismo, mediante la utilización de arreglos simulamos el comportamiento de un árbol binario casi completo que es la forma más frecuente de realizar la implementación en comparación con una estructura de nodos enlazados.

En el caso de ColaConPrioridad se utilizó la estructura de datos del heap para realizar la implementación. Se aprovechó sus ventajas para agregar los elementos “prioridad” junto con su etiqueta o elemento asociado y que el heap max se ocupe de conservar el orden. En consecuencia, se utilizó una matriz donde la primera columna refiere a los elementos prioridad y la segunda tiene el elemento asociado. Esta resolución fue de utilidad para reflejar los movimientos de la prioridad junto a su elemento asociado. De este modo, se construyó una sola estructura (la matriz) para almacenar los valores y al mismo tiempo utilizar la lógica del heap max.

Para la ListaEspecialTDA se realizó una implementación dinámica. Si bien en la mayoría de los métodos de la interfaz que se han desarrollado no surgieron inconvenientes, para el caso del método público extend() se agregó el método público primero() que rompe la premisa de hacer una interfaz que no dependa de la implementación ya que exhibe al usuario la clase NodoDoble que se utilizó para realizar la implementación. La dificultad surgió a partir de que en una implementación dinámica requiere conocer el primer nodo para recorrer la lista.No obstante, se decidió realizarlo de forma dinámica ya que el usuario puede ingresar elementos sin “límite” mientras que en una implementación estática, se debe determinar el tamaño del arreglo previamente. En tal sentido, se establece un límite al usuario para ingresar un máximo de elementos tan grande como el tamaño del arreglo.

Por último, se adjunta a este trabajo, el proyecto en Java el cual contiene cada una de las partes mencionadas y se añadieron las pruebas para facilitar la visualización del comportamiento de las implementaciones realizadas.

Grupo N°6 Integrantes:
Esquivel, Candela - LU 1134110
Bari, Diego - LU 1133805
Bird, Maria Florencia - LU 1133832

Bibliografía

Wengrow, J. (2017). A Common-Sense Guide to Data Structures and Algorithms: Level Up Your Core Programming Skills (2.ª ed., pp. 281–302). Raleigh, North Carolina: Pragmatic Bookshelf.

Weiss, M. A. (2013). Estructura de datos en Java (4.ª ed., pp. 797–828). Madrid: Pearson Education S.A. Madrid: Pearson Education S.A.

Conrado Martínez. (2011). Algorítmica: Heaps y heapsort. 2021, noviembre 8, de algorithmics Recuperado de http://algorithmics.lsi.upc.edu/docs/pqueues.pdf

HelloKoding. (21 Octubre 2020). Heap Data Structure Tutorial with Examples. 2021, noviembre 8, de hellokoding Recuperado de https://hellokoding.com/heap-data-structure/

Recuperado de https://runestone.academy/runestone/static/pythoned/Trees/ImplementacionDeUnMonticuloBinario.html

YO ANDROIDE. (2021). HEAPS – MAX HEAP Y MIN HEAP ¿Cómo funcionan? Código en Java. 2021, noviembre 8, de YO ANDROIDE Recuperado de https://yoandroide.xyz/heaps-max-heap-y-min-heap-como-funcionan/

Grupo N°6 Integrantes:

Esquivel, Candela - LU 1134110

Bari, Diego - LU 1133805

Bird, Maria Florencia - LU 1133832