Algoritmos y Estructuras de datos - 2023

Arboles y HEAP

Clase 1/2

Todos son **objetos** que intercambian mensajes Una **clase** tiene:

- Variables (referenciales y primitivas) //Estado
- Metodos //Comportamiento

Nombre del archivo: Contacto.java //Nombre de clase con mayuscula y CamelCase

```
package whatsapp; //A que paquete pertenece
import java.awt.Image; //importo clase Image

public class Contacto { //Clase publica Contacto
    //Variables de instancia
    private String nombre; // modificadorDeAcceo tipoDato nombreVar
    private Image imagen;
    private int id;

    //Constructor Vacio
    public Contacto(){
    }
    //Constructor con argumentos
    public Contacto(String nombre, Image imagen, int id){
        this.nombre = nombre;
        this.imagen = imagen;
        this.id = id;
    }
    //Sobrecarga de Constructores
    public Contacto(int id){
        this.id = id;
    }

    //Comportamientos
    public String getNombre() { //modificadorDeAcceo tipoDatoRetorno nombreMetodo
(parametros)
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    . . .
}
```

Palabras reservadas:

- modificadorDeAcceo = private, public
- tipoDatoRetorno = void (nada), int, String ...

Tipos de Datos:

- Primitivos (Se guarda en STACK)
  - Entero: byte, short, int (32 bits), long (64 bits)
  - Punto flotante: float y double
  - Un carácter: char
  - Lógico: boolean
- Referenciales (Clase) (Se guarda en HEAP)
  - NombreClase nomVar;
  - nomVar = new NombreClase();
  - NombreClase nomVar = new NombreClase();

Valor por defecto en variables de instancia:

- boolean = false;
- char = '\uoooo' (nulo);
- byte/short/int/long = 0;
- float/double = 0.0;
- objeto = null;

Clases wrappers

Clases de objetos con metodos por defecto ya creados dentro del paquete java.lang(ya disponible sin import):

- primitivo = Wrapper
- char = Character
- boolean = Boolean
- byte = Byte
- short = Short
- int = Integer
- long = Long
- float = Float
- double = Double

Autoboxing/Boxing-Unboxing:(cambio de tipo primitivo a wrapper)

Antes:

- Integer nro = new Integer(3);
- int num = nro.intValue();

Ahora:

- Integer nro = 3;
- int num = nro;

new():

Pasos al crear instancia de objeto:

1. Se aloca espacio para la variable
2. Se aloca espacio para el objeto en la HEAP y se inicializan los atributos con valores por defecto.
3. Se inicializan explícitamente los atributos del objeto.
4. Se ejecuta el constructor (parecido a un método que tienen el mismo nombre de la clase)
5. Se asigna la referencia del nuevo objeto a la variable.

Variables:

Variables locales a un metodo:

- Es necesario inicializar
- Se almacena en STACK

Variables de instancia

- Se almacena en HEAP

static

Variable static: es una unica variable (compartida) para todas las instancia de un objeto. Una sola referencia en memoria

```
private static int ultCont;
```

Metodo static: se utilizan cuando se necesita algún comportamiento que no depende de una instancia particular

```
public static int getUltCont{
    return ultCont;
}
```

Arreglos

- Un objeto que hace referencia a un conjunto
- Heterogeneos
- Se guardan en posiciones contiguas
- nombre.length cantidad de espacio reservado

Declaracion:

```
tipoDato[] nomArr = new tipoDato[cant]; // Se reserva cant posiciones de ese tipo
```

Inicializacion:

```
nomArr[0] = x;
```

Declaracion/Iniciacizacion:

```
tipoDato[] nomArr = {new tipoDato(), new tipoDato(),new tipoDato()}
String[] nomArr = {"example0", "example1",'example2'}
```

Recorridos

For Tradicional:

```
for (int i=0; i<a.length;i++)
    result = result + a[i];
```

For-each:

```
for (int valor: a) // para cada elemento elto de tipo int, en el arreglo a
    result = result + valor;
```

Matriz:

```
int[][] notas = new int[2][3];
```

Pasaje de parametros

En JAVA siempre se hace una copia de los parametros reales. Definicion de parametros:

- Parametros **formales**: Son los parametros en la definicion del metodo
- Parametros **reales**: Son los parametros en la invocacion al metodo

Tipos de parametros:

- Tipo primitivo: se hace una copia.
- Wrapper: crea una nueva instancia cuando le damos un valor.

Si pasamos un tipo de dato referencial (objeto) se pueden modificar el estado mediante los setters.

Clase 3

Herencia: Una clase hereda los atributos y comportamientos de otra clase

```
extends
    public class Camioneta extends Vehiculo { // La clase Camioneta es una
subclase de Vehiculo
        ...
    }
```

Sobreecritura de metodos (super) @Override

```
public String detalles() {
    return super.detalles() + " sigo agregando"; //Con super puedo llamar a
metodos de la super Clase
}
```

Importante: Habilitacion de metodos segun la jerarquia de herencia

Casting: Conversion de tipos

- Upcasting: casting hacia arriba en la jerarquia de herencia, es SEGURO.

  ```
  Vehiculo vc = new Camion(); // vc solo puede utilizar los metodos de
  Vehiculo y los metodos sobreescritos (binding dinamico) de Camion
  ```

- Downcasting: casting hacia abajo en la jerarquia de herencia.

  ```
  Integer x = (Integer)lista.elemento(2);
  ```

Clase object: Clase base de todas las clases sin extends

```
public boolean equals(Object obj){} // Se puede sobreescribir para que compare los
atributos
obj1.equals(obj2); //Compara si las dos referencias a memoria son iguales.

public String toString(){}, // Se puede sobreescribir para imprimir los atributos
ob1.toString(); //Devuelve el string
```

Clases abstractas: clase que sirve para modelar un concepto abastracto,agrupar clases y no se puede instanciar.

```
public abstract class FiguraGeometrica {
    //atributos
    //metodos
    public abstract void dibujar(); //Metodo delegado a la subclase
}
```

Listas

Operaciones:

- elemento(int pos): retorna el elemento de la posición indicada
- incluye(Object elem): retorna true si elem está en la lista, false en caso contrario
- agregarEn(Object elem, int pos): agrega el elemento elem en la posición pos
- agregarInicio(Object elem): agrega al inicio de la lista
- agregarFinal(Object elem): agrega al final de la lista
- eliminarEn(int pos): elimina el elemento de la posición pos
- eliminar(Object elem): elimina, si existe, el elemento elem
- esVacia(): retorna true si la lista está vacía, false en caso contrario
- tamanio(): retorna la cantidad de elementos de la lista
- comenzar(): se prepara para iterar los elementos de la lista
- proximo(): retorna el elemento y avanza al próximo elemento de la lista.
- fin(): determina si llegó o no al final de la lista, retorna true si no hay mas elementos, false en caso contrario

Tipo especifico no hay que hacer casting, y necesitamos una clase por cada tipo a almacenar
ListaDeEnteros(clase abstracta):

- ListaDeEnterosConArreglos
- ListaDeEnterosEnlazada

Tipo Object: estructura generica, pero hay que hacer Downcasting

- Lista heterogeneas de objetos
- Para recuperar un objeto de la lista hay que hacer Downcasting

  ```
  Integer x = (Integer)lista.elemento(2);
  ```

Tipos genericos: permiten abstraerse de los tipos Definicion:

```
public class ListaEnlazadaGenerica<T> extends ListaGenerica<T>{
    private NodoGenerico<T> inicio;
```

```
    ...
}
```

Instanciacion:

```
ListaEnlazadaGenerica<Integer> lista = new ListaEnlazadaGenerica<Integer>();
lista.agregarFinal(new Integer(50));
ListaEnlazadaGenerica<Alumno> lista = new ListaEnlazadaGenerica<Alumno>();
lista.agregarFinal(new Alumno(“Peres, Juan”, 3459));
```

Clase 4

Definiciones

- *Arbol*: coleccion de nodos
- *Camino*: cantidad de nodos por debajo de n
- *Profundidad*: es la longitud del unico camino desde la raiz hasta el nodo - 1
- *Grado*: numero de hijos del nodo
- *Altura*: la longitud del camino mas largo de n a su ultima hoja
- *Ancestro/Descendiente* : todos los nodos que esten por debajo del nodo
- *Arbol binario lleno*: si tiene todos los nodos con 2 hojas y su altura maxima
- *Cantidad de nodos en un arbol binario lleno*: altura = h, 2^(h+1) - 1
- *Cantidad de nodos en un arbol binario completo*: altura = h, min(2^h) entre max(2^(h+1) - 1)

Representacion : *hijo izquierdo - hijo derecho Recorridos*:

- Preorden: (recursivo)(abajo) se procesa la raiz, luego el hijo izquierdo y despues el hijo derecho public void preorden(){ imprimir (dato); si (tiene hijo_izquierdo) hijoIzquierdo.preorden(); si (tiene hijo_derecho) hijoDerecho.preorden(); }
- Postorden: (recursivo)(arriba) se procesa los hijos, izquierdo y derecha, y luego la raiz
- Inorden: (recursivo) se procesa el hijo izquierdo, luego la raiz y despues el hijo derecho
- Por niveles: primero la raiz, luego los hijos, los hijos de estos, etc
- ```
  public void porNiveles(){
      encolar(raiz)
      mientras(cola no se vacie){
          desencolar(v);
          imprimir (dato de v);
          si (tiene hijo_izquierdo)
              encolar(hijo_izquierdo);
          si (tiene hijo_derecho)
              encolar(hijo_derecho);
      }
  }
  ```
  ```

Para saber el recorrido de acuerdo con solamente la secuencia de nodos, primero hay que ver donde esta la raiz.

Arboles de expresion (logicas, aritmeticas)

- Nodos internos son operadores
- Nodos externos son operandos
- Lo utilizan los compiladores para anlizar

Traducir expresiones a notaciones:

- *Postfija/Sufija* (Postorden) (no necesita uso de parentesis) Algoritmo:

```
tomo un caracter
    mientras(exista caracter)
        si es un operado: creo un nodo y lo apilo
        si es un operador:
            creo un nodo R
            desapilo y lo agrego como hijo derecho
            desapilo y lo agrego como hijo izquierod de r
            apilo r
        tomo otro caracter
```

- *Prefija* (Preorden) (no necesita uso de parentesis) (de raiz hacia abajo/ recursivo)(primero operadores luego operandos) Algoritmo:

```
ArbolExpresion(A: ArbolBin,exp: string)
si exp nulo: nada
si es un operador:
    creo nodo raiz
    ArbolExpresion(subArbIzq de R, exp(sin 1 caracter))
    ArbolExpresion(subArbDer de R, exp(sin 1 caracter))
si es un operando:
    creo un nodo(hoja)
```

- *Infija* (Inorden)(hiper-parentisada por niveles/ ambigua)

Clase 5

Diagrama UML ArbolBinario<T> (Homogeneos respecto al tipo de dato)

- T dato;
- ArbolBinario<T> hijoDerecho
- ArbolBinario<T> hijoIzquierdo

Metodos:

- getDato();
- setDato();
- getHijoDerecho();
- getHijoIzquierdo();
- agregarHijoDerecho(ArbolBinario<T> hijo);
- agregarHijoIzquierdo(ArbolBinario<T> hijo);
- eliminarHijoDerecho(arbolBinario<T> hijo);
- agregarHijoIzquierdo(ArbolBinario<T> hijo);
- esVacio();
- esHoja();
- tieneHijoDerecho();
- tieneHijoIzquierdo();

Creacion:

- De abajo hacia arriba, primero ultimas hojas y por ultimo la raiz
- Creo padres y luego hijos

Recorridos:

- *Preorden*: Se procesa primero la raíz y luego sus hijos, izquierdo y derecho.
  - Algoritmo:
    - En la misma clase:

```
      public void printPreorden(){
          System.out.println(this.getDato()); //Se puede agregar a
una lista tambien, en vez de imprimir
          if (this.tieneHijoIzquierdo){
              this.getHijoIzquierdo().printPreorden();
          }
          if (this.tieneHijoDerecho){
              this.getHijoDerecho().printPreorden();
          }
      }
```

    - En otra clase:

```
      public void preorden(ArbolBinario<T> arbol){
          System.out.println(arbol.getDato());
          if (arbol.tieneHijoIzquierdo){
              this.preorden(arbol.getHijoIzquierdo);
          }
          if (arbol.tieneHijoDerecho){
              this.preorden(arbol.getHijoDerecho);
```

```
            }
        }
```

- *Inorden*: Se procesa el hijo izquierdo, luego la raíz y último el hijo derecho
- *Postorden*: Se procesan primero los hijos, izquierdo y derecho, y luego la raíz
- *Por niveles*: Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc. Se utiliza una cola, se encolan los elemntos y a medida que se pasa de niveles se van encolando null
    - Algoritmo:

```
    public void recorridoProNiveles(){
        ArbolBinario<T> arbol = null;'
        ColaGenerica<ArbolBinario<T>> cola = new
  ColaGenerica<ArbolBinario<T>>();
        cola.encolar(this);
        cola.encolar(null);
        while (!cola.esVacia()){
            arbol = cola.desencolar();
            if (arbol != nul){
                System.out.print(arbol.getDato());
                if (arbol.tieneHijoIzquierdo){
                    cola.encolar(arbol.getHijoIzquierdo);}
                if (arbol.tieneHijoDerecho){
                    cola.encolar(arbol.getHijoDerecho); }
            } else if (!cola.esVacia()){
                System.out.println();
                cola.encolar(null);
            }
        }
    }
```

Algoritmo para saber si esta lleno el arbol:

- Algoritmo:

```
  public boolean lleno() {
      ArbolBinario<T> arbol = null;
      ColaGenerica<ArbolBinario<T>> cola = new ColaGenerica<ArbolBinario<T>>
();
      boolean lleno = true;
      int cant_nodos=0;
      int nivel= 0;
      cola.encolar(this);
      cola.encolar(null);
      while (!cola.esVacia() && lleno) {
```

```
            arbol = cola.desencolar();
            if (arbol != null) {
                System.out.print(arbol.getDatoRaiz());
                if (!arbol.getHijoIzquierdo().esvacio()) {
                    cola.encolar(arbol.getHijoIzquierdo());
                    cant_nodos++;
                }
                if (!(arbol.getHijoDerecho().esvacio())) {
                    cola.encolar(arbol.getHijoDerecho());
                    cant_nodos++;
                }
            } else if (!cola.esVacia()) {
                if (cant_nodos == Math.pow(2, ++nivel)){ // ++nivel Primero
incrementa y luego retorna el valor incrementado
                    cola.encolar(null);
                    cant_nodos=0;
                    System.out.println();
                } else {
                    lleno=false;
                    }
            }
        }
        return lleno;
    }
```

Arbol de Expresion:

- Convertir expresion *posfija en arbol de expresion*:
    - Idea: apila los elementos que no sean operadores, al encontrarse con un operador agrega los operadores como hojas de los operandos
    - Algoritmo:

```
    public ArbolBinario<Character> convertirPostfija(String exp) {
    Character c = null;
    ArbolBinario<Character> result;
    PilaGenerica<ArbolBinario<Character>> p = new
  PilaGenerica<ArbolBinario<Character>>();

    for (int i = 0; i < exp.length(); i++) {
        c = exp.charAt(i);
        result = new ArbolBinario<Character>(c);
        if ((c == '+') || (c == '-') || (c == '/') || (c == '*')) {
            // Es operador
            result.agregarHijoDerecho(p.desapilar());
            result.agregarHijoIzquierdo(p.desapilar ());
            }
        p.apilar(result);
        }
```

```
    return (p.desapilar());
    }
```

- Convertir expresion *prefija en arbol de expresion*:
    - Idea: tomamos el primer elemento del string, ir agregando izquierda, luego derecha del string-1
    - Nota: al momento de hacer exp.delete(0,1) borra el elemento y lo modifica dentro del objeto - el caso base son los operandos, y el recursivo los operadores
    - Algoritmo:

```
    public ArbolBinario<Character> convertirPrefija(StringBuffer exp) {
    Character c = exp.charAt(0);
    ArbolBinario<Character> result = new ArbolBinario<Character>(c);
    if ((c == '+') || (c == '-') || (c == '/') || c == '*') {
        // es operador

result.agregarHijoIzquierdo(this.convertirPrefija(exp.delete(0,1)));
//Aca borras un elemento de exp

result.agregarHijoDerecho(this.convertirPrefija(exp.delete(0,1))); //
Aca borras otro mas, al momento de hacer delete() borras y queda
guardado sin ese caracter
    }
    // es operando
    return result;
    }
```

- Evaluacion de un arbol de expresion:
    - Idea: si no es operador, retorno integer. Si es operador retorno la operacion entre los dos operadores.
    - Nota: recursivo
    - Algoritmo:

```
    public Integer evaluar(ArbolBinario<Character> arbol) {
        Character c = arbol.getDato();
        if ((c == '+') || (c == '-') || (c == '/') || c == '*') {
            // es operador
            int operador_1 = evaluar(arbol.getHijoIzquierdo());
            int operador_2 = evaluar(arbol.getHijoDerecho());
            switch (c) {
                case '+':
                    return operador_1 + operador_2;
                case '-':
                    return operador_1 - operador_2;
                case '*':
```

```
                    return operador_1 * operador_2;
                case '/':
                    return operador_1 / operador_2;
            }
        }
        // es operando
        return Integer.parseInt(c.toString());
    }
```

Clase 6

Construccion de un arbol de expresion A partir de una expresion:

- Postija (pila):
    - Algoritmo:

```
  tomo un carácter de la expresión
  mientras ( existe carácter ) hacer
      si es un operando →
          creo un nodo y lo apilo.
      si es un operador (lo tomo como la raíz de los dos últimos nodos
creados)
          → creo un nodo R,
          desapilo y lo agrego como hijo derecho de R
          desapilo y lo agrego como hijo izquierdo de R
          apilo R.
      tomo otro carácter
  fin
```

- Prefija (recursiva):
    - Algoritmo:

```
  ArbolExpresión (A: ArbolBin, exp: string )
      si exp nulo
          → nada.
      si es un operador →
          creo un nodo raíz R
          ArbolExpresión (subArbIzq de R, exp(sin 1 carácter) )
          ArbolExpresión ( subArbDer de R, exp(sin 1 carácter) )
      si es un operando →
          creo un nodo (hoja)
```

- Infija (Pasar a postfija - armar arbol postija - recorrer en preorden):

```
  a) si es un operando → se coloca en la salida.
  b) si es un operador → se maneja una pila según la prioridad del operador
en relación al tope de la pila
      -operador con > prioridad que el tope se apila
      -operador con <= prioridad que el tope se desapila elemento
colocándolo en la salida.
  c) si es un "(" se apila, si es ")" se desapila todo hasta el "(",
incluído éste
  d) cuando se llega al final de la expresión, se desapilan todos los
elementos llevándolos a la salida, hasta que la pila quede vacía
```

Arboles Generales

- Grado: es el número de hijos del nodo
- Grado del árbol: es el grado del nodo con mayor grado
- Altura: es la longitud del camino más largo desde la raiz hasta una hoja.
- Profundidad / Nivel: es la longitud del único camino desde la raíz hasta el nodo

Recorridos:

- Preorden: Se procesa primero la raíz y luego los hijos (arriba hacia abajo)
    - Algoritmo:

```
public void preOrden() {
    imprimir (dato);
    obtener lista de hijos;
    mientras (lista tenga datos) {
        hijo  obtenerHijo;
        hijo.preOrden();
    }
}
```

- Inorden: Se procesa el primer hijo, luego la raíz y por último los restantes hijos (abajo hacia arriba)
- Postorden: Se procesan primero los hijos y luego la raíz (abajo hacia arriba)
    - Algoritmo:

```
public void postOrden() {
obtener lista de hijos;
mientras (lista tenga datos) {
    hijo obtenerHijo;
    hijo.postOrden();
    }
imprimir (dato);
}
```

- Por niveles: Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.
    - Algoritmo:

```
public void porNiveles() {
    encolar(raíz);
    mientras cola no se vacíe {
        v desencolar();
        imprimir (dato de v);
        para cada hijo de v
        encolar(hijo);
    }
}
```

Definiciones:

- Arbol lleno: es lleno si cada nodo interno tiene el mismo grado y todas las hojas estan en el mismo nivel Recursivamente:
    1. T es un nodo simple o
    2. T es de altura h y todos sus subarbolers son llemos de altura h-1
- Cantidad de nodos: Sea T un arbol lleno de grado k y altura h, la cantidad de nodos N es:

```
N = (k^(h+1) - 1) / (k-1)
```

- Arbol completo: es completo si es lleno de altura h-1 y el nivel h se completa de izquierda a deracha

Representacion:

- Lista de hijos
    - Cada nodo tiene:
        - Informacion propia del nodo
        - Lista de todos sus hijos
        - Cada nodo tiene:
            - Información propia del nodo
            - Referencia al hijo más izquierdo (nivel inferior)
            - Referencia al hermano derecho (mismo nivel)

Preguntas de parcial

1. ¿Cuántos niveles tiene el árbol?

```
Seudocódigo Ejerc1-Niveles {
    q: cola de vértices;
    encolar raíz R en q; encolar ?? en q;
    mientras (cola no se vacíe) {
        desencolar v de q;
        si (dato de v no es ??) {
            imprimir (dato de v);
            para cada hijo w de v
                encolar w en q; }
        sino
            si (q no está vacía)
                encolar ?? en q;
    }
}
```

2. ¿Cuántos nodos hay en cada nivel del árbol?

```
Seudocódigo Ejerc2-Niveles {
    q: cola de vértices;
    encolar raíz R en q; encolar ?? en q;
    mientras (cola no se vacíe) {
        desencolar v de q;
        si (dato de v no es ??) {
            imprimir (dato de v);
            para cada hijo w de v
                encolar w en q; }
        sino
            si (q no está vacía)
                encolar ?? en q;
    }
}
```

3. ¿Cuántos nodos hay en el nivel k del árbol?

```
Seudocódigo Ejerc3-Niveles (int k) {
    q: cola de vértices; nroNivel=0; cantNodos=0;
    encolar raíz R en q; encolar ?? en q;
    mientras (cola no se vacíe) {
        desencolar v de q;
        si (dato de v no es ??){
            si (nroNivel==k){
                mientras (dato de v no es ??)
                    {cantNodos++;
                    desencolar v de q;}
            }
            sino para cada hijo w de v
```

```
                    encolar w en q;
            }sino
            si (q no está vacía){
                encolar ?? en q;
                nroNivel++; }
        }
        return cantNodos;
    }
```

Clase 7

Interfaces

Es una coleccion de definiciones de metodos sin implementacion/cuerpo y de declaraciones de variables de clase cosntantes, agrupadas bajo un nombre. Debido a que en java la herencia de clases es simple, la herencia de interfaces es multiple. **Definicion**:

```
    package nomPaquete;
    public interface UnaInter extends SuperInter, SuperInter2...{
        Declaracion de metodos: implicitamente public y abstact
        Declaracion de constantes implicitamente public, static y final
    }
    public interface Volador{
        (public static final) long UN_SEGUNDO=1000; //Dentro de los espacios no
puede ir, y ese es valor por defecto si es que no esta definido
        (public abstract) String despegar();
    }
```

Herencia Multiple: Permite crear una clase derivada de varias clases bases, debido a que la implemetacion de metodos se define en la clase que se exitende no causa problema al convinar varias interfaces

Upcasting: El mecanismo de upcasting no tiene en cuenta si es una clase concreta, abstracta o una interface. Funciona de la misma manera **Polimorfismo**: Debido a que cada metodo es diferente en cada una de las clases que heredan de la interfaz, hay polimorfismo. *Ejemplo*:

```
public static void main(String[] args){
    Volador[]m = new Volador[3];
    m[0] = new Avion();
    m[1] = new Helicoptero();
    m[2] = new Pajaro();
    for (int i=0; i<m.length;i++){
        partida(m[i]);
    }
}
```

Interfaces vs Clases Abstractas:

- Las interfaces y las clases abstractas proveen una interface comun
- Las interfaces son completamente abstractas, no tienen ninguna implementacion
- Con interfaces no hay herencia de metodos, las clases asbtractas si
- No es posible crear instancias de clases abstractas ni de interfaces
- Una clase puede externder solo una clase abstracta, pero puede implementar multiples interfaces

Uso de interfaces vs Uso de clases abstractas

- Para crear una clase base con metodos sin implementacion y sin variables de instancia, es preferible usar interfaces
- Si estamos forzados a tener implementacion o definir atributos, entonces usamos clases abstractas
- Como Java no soporta herencia multiple de clases, si se quiere que una clase sea ademas del tipo de su superclase de otro dtipo diferete, entonces es necesario usar interfaces

Arrays Arrays es una clase del paquete java util, la cual sirbe para manipular arreglos, provee mecanismos de *BUSQUEDA y ORDENACION*

```
import java.util.*
Arrays.sort(nombreArray); // El arreglo queda ordenado, si los objetos son de tipo
comparable
```

Queda ordenado por el orden de magnitud, comenzando del mas pequenio al mas grande Si el arreglo que le pasamos no son objetos de tipo comparable da error en copilacion

```
Interface java.lang.Comparable
```

Para poder ordenar objetos que no son de tipo comparable. Hay que hacer una interface Comparable<T>.

```
public interface Comparable<T>{
    public int CompareTo(T o);
}
```

Este metodo retorna:

- 0 si es igual
- >0 si el objeto receptor es mayor que el pasado por parametro
- <0 si el objeto receptor es menor que el pasado por parametro

```
import java.util.*
public class Persona implements Comparable <Persona>{
```

```
        ...
        public int compareTo(Persona o){
            return this.edad - o.edad; //Establece de acuerdo a que generar el
    orden
        }
      }
```

En el caso de tener un array de Personas podre utilizar el Array.sort(Personas) ya que el elemento a comparar son las edades *Ejemplo*:

- Arbol binario de busqueda utilizando la interfaz CompareTo<ArbolBinarioDeBusqueda<T>>

```
  public class ArbolBinarioDeBusqueda<T extends Comparable<T>>{
          ...
      public void agregar(T x){
          if (dato == nul){
              dato = X;
          }else{
              this.agregar(x, this);
          }
      }
      private void agregar(T x, ArbolBinarioDeBusqueda<T> t){
          if(x.compareTo(t.getDato()) < 0){ //Mas pequenio
              if (!t.tieneHijoIzquierdo()){
                  t.setHijoIzquerdo(new ArbolBinarioDeBusqueda<T>(x));
              }else{
                  this.agregar(x,t.getHijoIzquierdo());
              }
          } else if (x.compareTo(t.getDato()) > 0){ // Mas grande
              if (!t.getHijoDerecho()){
                  t.setHijoDerecho(new ArbolBinarioDeBusqueda<T>(x));
              }else{
                  this.agregar(x,t.getHijoDerecho());
              }
          }
      }
  }
```

Clase 8

rboles Generales

- Atributos:
  - dato:T
  - hijos: ListaGenerica<ArbolGeneral<T>>
- Metodos:
  - ArbolGeneral(T)

- ArbolGeneral(T,ListaGenerica<ArbolGeneral<T>>)
- getDato();
- setDato():void
- setHijos(ListaGenerica<ArbolGeneral<T>>):void
- getHijos():ListaGenerica<ArbolGeneral<T>>
- agregarHijo(ArbolGeneral<T>):void
- esVacio():boolean
- esHoja():boolean
- tieneHijos():boolean
- eliminarHijo(ArbolGeneral<T>):void
- altura():integer
- nivel(T):integer
- ancho():integer

Recorridos: metodos de ArbolGeneral que retornan listas

- preOrden:

```
  public ListaEnlazadaGenerica\<T> preOrden(){
      ListaEnlazadaGenerica\<T> lis = new ListaEnlazadaGenerica\<T>();
      this.preOrden(lis);
      return lis;
  }
  private void preOrden(ListaGenerica\<T> l){
      l.agregarFinal(this.getDato());
      ListaGenerica<ArbolGeneral\<T>> lHijos = this.getHijos();
      lHijos.comenzar();
      while (!lHijos.fin()){
          (lHijos.proximo()).preOrden(l)
      }
  }
```

- Por niveles:

```
  public void porNiveles(){
      encolar(raiz);
      mientras cola no se vacie {
          desencolar()
          imprimir (dato de v);
          para cada hijo de v
              encolar(hijo)
      }
  }
  public ListaGenerica<T> porNiveles(ArbolGeneral<T> arbol){
      ListaGenerica<T> result = new ListaEnlazadaGenerica<T>();
      ColaGenerica<ArbolGeneral<T>> cola = new ColaGenerica<ArbolGeneral<T>>
();
```

```
      ArbolGeneral<T> arbol_aux;
      cola.encolar(arbol);
      while (!cola.esVacia()){
          arbol_aux = cola.desencolar();
          result.agregarFinal(arbol_aux.getDato());
          if (arbol_aux.tieneHijos()){
              ListaGenerica<ArbolGeneral<T>> hijos = arbol_aux.getHijos();
              hijos.comenzar();
              while (!hijos.fin()){
                  cola.encolar(hijos.proximo());
              }
          }
      }
  return result;
```

} **Ejercicios Parcial**

```
  //Retorna lista con elementos de tipo imagen en un nivel en partiuclar
  public ListaGenerica<Recurso> getImagenes(ArbolGeneral<Recurso> ag,int
nivel_pedido){
      ListaGenerica<Recurso> result = new ListaEnlazadaGenerica<Recurso>();
      ColaGenerica<ArbolGeneral<Recurso>> cola = new
ColaGenerica<ArbolGeneral<Recurso>>();
      ArbolGeneral<Recurso> arbol_aux;
      cola.encolar(ag); cola.encolar(null);
      boolean proceso_nivel_pedido = false;
      int nivel = 0; //La raiz comienza en nivel 0
      while (!cola.EsVacia() && proceso_nivel_pedido == false){
          arbol_aux = cola.desencolar();
          if (arbol_aux != null){
              if (nivel == nivel_pedido && arbol_aux.getDato().esImagen() )
                  result.agregarFinal(arbol_aux.getDato())
              if (arbol_aux.tieneHJijos()){
                  ListaGenerica<ArbolGeneral<Recurso>> hijos =
arbol_aux.getHijos();
                  hijos.comezar();
                  while (!hijos.fin()){
                      cola.encolar(hijos.proximo())
                  }
              }
          } else{
              if (!cola.esVacia()){
                  nivel++;
                  cola.encolar(null);
                  if (nivel>nivel_pedido)
                      proceso_nivel_pedido = true;
              }
          }
      }
```

```
        return result;
    }

    //Retorna camino que llega a princesa sin pasar por dragon
    private void encontrarPrincesa(ArbolGeneral<Personaje>
arbol,ListaGenerica<Personaje> lista, ListaGenerica<Personaje> camino){
        Personaje p = arbol.getrDato();
        if (p.esPrincesa()){
            clonar(lista, camino);
        }
        if (camino.esVacia()){
            ListaGenerica<ArbolGeneral<Personaje>> lHijos = arbol.getHijos();
            lHijos.comenzar():
            while (!lHijos.fin() && camino.esVacia()){
                ArbolGeneral<Personaje> aux = lHijos.proximo();
                if (!aux.getDato().esDragon()){
                    lista.agregarFinal(aux.getDato());
                    encontrarPrincesa(aux,lista,camino);
                    lista.eliminarEn(lista.tamnio);
                }
            }
        }
    }

    //Uso de backtracking, va agregando a medida que vuelva a sus llamadas
recursivas
    public ListaEnlazadaGenerica<Personaje>
encontrarPrincesa(arbolGeneral<Personaje> arbol){
        ListaEnlazadaGenerica<Personaje> lista = new
ListaEnlazadaGenerica<Personaje>();
        if (arbol.getDato().esPrincesa() || arbol.getDato().esDragon() ||
arbol.esHoja()){
            if (arbol.getDato().esPrincesa()){
                Personaje p = arbol.getDato();
                lista.agregarInicio(p);
            }
            return lista;
        }
        ListaGenerica<ArbolGeneral<Personaje>> lHijos = arbol.getHijos();
        lHijos.comenzar();
        while (!lHijos.fin() && lista.esVacia()){
            lista = encontrarPrincesa(lHijos.proximo());
            if(!lista.esVacia()){
                lista.agregarInicio(arbol.getDato());

            }
        }
    return lista
    }

    //Retorna la cantidad de veces que hay un recorrido que suma valor
```

```
public static int contadorGematria(ArbolGeneral<Integer> ag, int valor){
    int resta = valor - ag.getDato();
    if (ag.esHoja() && resta == 0)
        return 1
    else if (resta > 0){
        int cont = 0
        ListaGenerica<arbolGeneral<Integer>> lista = ag.getHijos();
        lista.comenzar();
        while (!lista.fin()){
            ArbolGeneral<Integer> arbol = lista.proximo();
            cont = cont + contadorGematria(arbol,resta);
        }
        return cont;
    }
}
```

Clase 9

Cola de prioridad

Es una estructura de datos que permite al menos dos **operaciones**:

- Insert: insertar un elemento
- DeleteMin: Encuentra, recupera y elimina el elemento minimo (el elemento con mas prioridad)
  Implementaciones:
- Lista ordenada: Insert tiene O(n) operaciones / DeleteMin tiene O(1) operaciones
- Lista no ordenada: Insert tiene O(1) operaciones / DelteMin tiene O(n) operaciones
- Arbol Binario de Busqueda: Insert y DeleteMin tiene en promedio O(log N) operaciones
- **Heap** (monticulo) binario: no usa punteros y ambas operaciones estan en O(log N) operaciones en el peor caso
    - **Propiedades**:
        - Propiedad *estructural*: Una heap es un arbol binario completo(lleno hasta h-1 y se completa el ultimo nivel de izquierda a derecha)
            - El numeros de nodos n de un arbol binario completo de altura h: 2^h <= n <= ((2^(h+1))-1)
            - La raiz esta almacenada en la posicion 1
            - Para un elemento que esta en la posicion i:
                - El hijo izquierdo esta en la posicion 2*i
                - El hijo derecho esta en la posicion 2*i +1
                - El padre esta en la posicion i/2
        - Propiedad de *orden*:
            - MinHeap:
                - El elemento minimo esta almacenado en la raiz
                - El dato almacenado en cada nodo es menor o igual al de sus hijos
            - MaxHeap:
                - Se usa la propiedad inversa

Ventajas de Heap:

- No se necesitan punteros
- Facil implementacion de las operaciones

Operaciones:

- *Insert*: El dato se inserta como ultimo item en la heap (ultimo elemento del arreglo) La propiedad de la heap puede ser violada
    - Precolate Up: Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden (se cambia el padre por el hijo, si el hijo es mas grande) Idea general:

```
insert(Heap h, Comparable x){
    h.tamanio = h.tamanio +1;
    n = h.tamanio;
    while (n>2 0 && h.dato[n/2] > x) {
        h.dato[n] = h.dato[n/2];
        n=n/2;
    }
    h.dato[n]=x; //ubicacion corecta de 'x'
}
```

Mejor reoslucion:

```
precolate_up(Heap h, Integer i){
    temp = h.dato[i];
    while (i>2 0 && h.dato[i/2] > temp) {
        h.dato[i] = h.dato[i/2];
        i=i/2;
    }
    h.dato[i]=temp; //ubicacion corecta de 'x'
}
insert(Heap h, Comparable x){
    h.tamanio = h.tamanio +1; //Agrega un lugar
    h.dato[h.tamanio] = x; //Pone el dato en el ultimo lugar
    precolate_up(h,h.tamanio); //Filtra para que quede ordenado
}
```

- *DeleteMin*:
    - Guardo el dato de la raiz
    - Elimino el ultimo elemento y lo almaceno en la raiz Percolte Down: Se debe hacer un filtrado hacia abajo para restaurar la propiedad (intercambia el dato de la raiz hacia abajo a lo largo del camino que contiene los hijos minimos) Idea general:

```
    delete_min ( Heap h, Comparable e) {
    if (not esVacía(h) ) {
        e := h.dato[1];
        candidato := h.dato[h.tamaño];
        h.tamaño := h.tamaño - 1;
        p := 1;
        stop_perc := false;
        while ( 2* p <= h.tamaño ) and ( not stop_perc) {
            h_min := 2 * p; // buscar el hijo con clave menor
            if h_min <> h.tamaño{ //como existe el hijo derecho comparo a
  ambos
                if ( h.dato[h_min +1] < h.dato[h_min] )
                    h_min := h_min + 1
            }
            if candidato > h.dato [h_min] { // percolate_down
                h.dato [p] := h.dato[ h_min ];
                p := h_min;
            } else
                stop_perc := true;
        }
        h.dato[p] := candidato;
        }
    } // end del delete_min
```

Percolate_down:

```
percolate_down ( Heap h, int p) {
    candidato := h.dato[p]
    stop_perc := false;
    while ( 2* p <= h.tamaño ) and ( not stop_perc) {
        h_min := 2 * p; // buscar el hijo con clave menor
        if h_min <> h.tamaño then
        if ( h.dato[h_min +1] < h.dato[h_min] )
            h_min := h_min + 1
        if candidato > h.dato [h_min] { // percolate_down
            h.dato [p] := h.dato[ h_min ]
            p := h_min;
        }
        else stop_perc := true;
    } // end { while }
    h.dato[p] := candidato;
} // end {percolate_down }

delete_min ( Heap h; Comparable e) {
    if (h.tamaño > 0 ) { // la heap no está vacía
        e := h.dato[1] ;
        h.dato[1] := h.dato[h.tamaño] ;
        h.tamaño := h.tamaño - 1;
```

```
            percolate_down ( h ; 1);
    }
    } // end del delete_min
```

- *DecreaseKey(x,y, H)*: Decremenentar la clave que esta en la posicion x de la heap H, en una cantidad y
- *IncreaseKey(x,y, H)*: Incrementa la clave que esta en la posicion x de la heap H, en una cantidad y
- *DeleteKey(x)*: Elimina la clave que esta en la posicion X Puede realizarse:
    - DecreaseKey()
    - DeleteMin(H)

Clase 10

Heap:

Arbol binario completo (lleno hasta la altura h-1 y se completa de izquierda a derecha) que cumple propiedad de orden.

- MinHeap: Para cada nodo, el valor que contenga es menor que los hijos.

Operaciones:

- Insertar:
    - Inserto en el ultimo lugar(respetando propiedad de orden)
    - Filtra hacia arriba
- BorrarMin:
    - Se guarda el valor de la primer posicion
    - Se toma el ultimo nodo y se pone como raiz del arbol
    - Se filtra (reacomodo) hacia abajo para que cumpla la propiedad de orden
- DecreaseKey(pos,cant,heap) //Ej: Subir prioridad a una tarea (S.O)
    - Decrementa la clave que esta en pos de la heap, en una cantidad cant
- IncreaseKey(pos,cant,heap) //Ej: Bakar prioridad a una tarea (S.O)
    - Incrementar la clave que esta en la pos de la heap, en una cantidad cant
- DeleteKey(pos)
    - Elimina la clave que esta en la posicion pos Puede realizarse:
        - DecreaseKey(x,(infinito),H)
        - DeleteMin(H)

Como construir una heap apartir de una lista de elementos

- Se puede insertar los elementos de a uno
    - Se realiza (n log n) operaciones en total

BuildHeap: (Mas eficiente) Se puede usar un algoritmo de orden lineal, es decir, proporcional a los n elementos

1. Insertar los elementos desordenados en un arbol binario completo

2. Filtrar hacia abajo cada uno de los elementos
    - Se empieza filtrando desde el elemento que esta en la posicion (tamanio/2) // Esto es debido a que de la mitad hacia delante son todas hojas
        - Se filtran los nodos que tienen hijos < (tamanio/2)
        - El resto de los nodos son hojas > (tamanio/2)
    - Se elije el menor de los hijos
    - Se compara el menor de los hijos con el padre

Cantidad de operaciones requeridas:

- En el filtrado de cada nodo recorremos su altura
- Para acotar la cantidad de operaciones del Algoritmo BuildHeap, debemos calcular la suma de las alturas de todos los nodos

Teorema: En un arbol binario lleno de altura h que contiene 2^(h+1) - 1. La suma de las alturas de los nodos es: 2(^h+1) - 1 - (h+1)

- Demostracion: Un arbol tiene 2^i nodos de altura h-1 S = 2^0 (h-0) + 2(h-1) + 4(h-2) + 8(h-3) + ... 2^(h-1) (1) (2S - S) + 2 = (2^(h+1) - (h + 1))
    - Un arbol binario completo tiene entre 2^h y 2^(h+1) - 1 nodos, el teorema implica que la suma es de O(n) donde n es el numero de nodos

Ordenacion de vectores usando Heap Algoritmo que usa una heap y requiere una cantidad aproximada de (n log n) operaciones. Construir una MinHeap, realizar n DeleteMin operaciones e ir guardando los elementos extraidos en otro arreglo *Desventaja*: requiere el doble espacio

HeapSort: Construir una MaxHeap() con los elementos que se desean ordenar, intercambiar el ultimo elemento con el primero, decrementar el tamanio de la heap y filtrar hacia abajo.

- Este algoritmo usa la misma heap y no se necesita otro arreglo.

Algoritmo:

```
Armar MaxHeap (queda ordenado de forma creciente)
Mientras queden posiciones del arrelgo (tamanio) sin operar:
    Intercambia el primer elemento con el ultimo
    Decrementa el tamanio
    Filtra todo la heap de arriba hacia abajo poniendo el mas grande en el padre
```

Clase 12

Analisis de algoritmos

Nos permite comparar algoritmos en forma independiente de una plataforma en particular Mide la eficiencia de un algoritmo, dependiendo del tamanio de entrada Pasos:

- Caracterizar los datos de entrada del algoritmo
- Identificar las operaciones abstractas, sobre las que se basa el algoritmo

- Realizar un analisis matematico, para encontrar los valores de las cantidades anteriores

Introduccion al concepto de T(n)

- Priorizamos el tiempo de ejecucion por el espacio en memoria, con un enfoque teorico (no empirico)
- n = tamanio de la entrada de los datos
- Se calcula de forma teorica
- Siempre el tiempo en el peor de los casos
- Debemos enfocarnos en cuan rapido crece una funcion T(n) respecto al tamanio de la entrada. A esto lo llamamos la tasa o velocidad de crecimiento del tiempo de ejecucion

Ordenes de ejecucion de los algoritmos:

- Logaritmico: log2(n)
- Lineal: n
- Cuadratico: n^2

Funciones para describir tasas de crecimiento de los algoritmos: exponencial, cubica, cuadratica, lineal ritmica, lineal, logartimica, constante

Ejemplo: Un algoritmo requiere f(n) operaciones para resolver un problema y la computadora procesa 100 operaciones por segundo. Si f(n) es log10n determine el tiempo en segundos requerido por el algoritmo para resolver un problema de tamanio n=10000.

- TE = (log10(10000)) / 100 segundos

Siempre vamos a precisar tres parametros:

- T(n) cantidad de operaciones del algoritmo
- Cantidad de operaciones que puede hacer la computadora
- Cantidad de datos a aplicar el algoritmo

Calculo del tiempo de Ejecicion:

- Cada operacion de asignacion y operacion elementar toma un paso **Estructuras de control**:
    - Secuencia:
    - Condicional:
        - if/else: Se suma el tiempo que se tarda en evaluar la condicion y el tiempo maximo entre la cantiad de operaciones del if y el else
            - T(condicion) + max(T(statement1)
        - switch:
            - T(condicion) + max(T(con todos los casos)
    - Iteracion:
        - for:
          Big O: O(n) Serie aritmetica: con i=1 to n de la funcion i = n/2 * (n+1)
            - T(n) = constatesAntes + sumatoria(cant de rep) * constantesDentro = constantesAntes + (n * (constantesDentro))

- while:

-do-while