RESUMEN 1ER PARCIAL POO II

Type	Resumen
Autor	Juan Pablo Frascino
Reviewed	☐

CONTENIDO PREVIO; RESUMEN UML:

El UML(UNIFIED MODELING LANGUAGE) es un lenguaje de modelacion de diagrama de clases. Fue creado con el objetivo de estandarizar los planos de software en la programacion orientada a objetos. En el se representan las clases de un programa(con sus nombres, atributos y metodos) y sus relaciones entre ellas.

ESTRUCTURA DE UNA CLASE

CLASE
-atributo:Tipo
+metodo():Tipo

TIPOS DE RELACIONES EN EL UML:

- HERENCIA: Cuando una clase hereda de otra(es una flecha con la punta blanca)
- INTERFACE: Cuando una clase implementa una interface(es una flecha puntuada con punta blanca)
- ASOCIACION: no la usamos
- DEPENDENCIA: Cuando un objeto de otra clase aparece en los parametros de un metodo de otra clase, entonces la clase que lo tiene depende de la que tiene en los parametros.(flecha puntuada con la punta en forma de > apuntando a la cual depende)
- AGREGACION(PARTE DE): Cuando un objeto de una clase contiene a un objeto de otra clase en sus atributos. Tiene dos tipos AGRUPACION(rombo

blanco) si no es esencial o COMPOSICION(rombo negro) si es esencial(la parte no pude existir fuera de la composicion)

Las clases abstractas tienen en nombre en italica, y las interfaces se les pone interface.

PRUEBAS DE SOFTWARE

Una prueba consiste en una busqueda de errores, nunca podremos testear un software al 100% pero podemos tratar de minimizar sus errores, por eso, que la prueba no encuentre errores no significa que no los haya ya que un sistema mal probado puede contenerlos.

Una caracteristica importante de una buena prueba es la ESPECIFICIDAD, esto significa que la prueba se enfoca en un aspecto ESPECIFICO y PUNTUAL, ya que si este falla entonces sera facil el saber que es lo que fallo. Una prueba muy abarcativa es mala al detectar que es lo que esta fallando.

Durante el ciclo de vida del desarrollo de un software las pruebas son FUNDAMENTALES puesto que estas atraviesan TODAS LAS ETAPAS. Alrededor del 50% del tiempo es pruebas.

La etapa de pruebas la dividimos en 2:

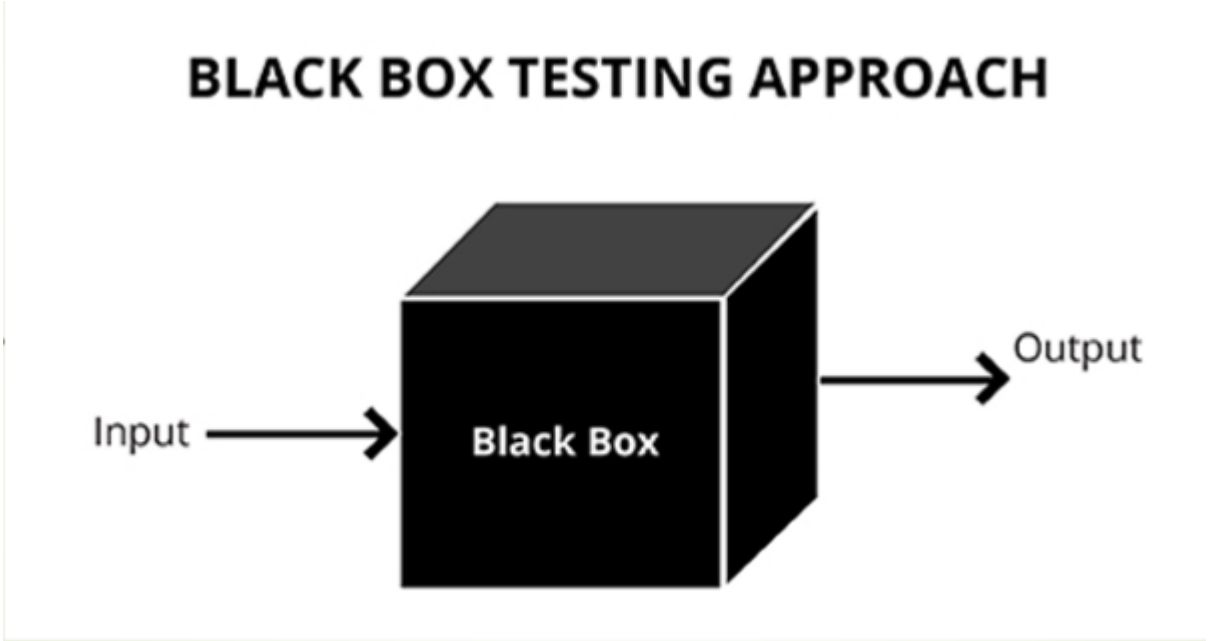
- PREPARACION DE LA PRUEBA: Despues del analisis y antes del diseño
- EJECUCION DE LA PRUEBA: Al final de todo y luego del desarrollo

Entonces podemos ver que las pruebas se pueden comenzar a idear mucho antes de desarrollar la solucion. Este enfoque nos permite una planificacion y ejecucion mas simple, ordenada y rapida, ya que enmendar un error es mucho mas facil con el desarrollo ya terminado.

Cuando ideamos una prueba es importante tener en cuenta el RESULTADO ESPERADO para luego cuando pasamos a la etapa de ejecucion poder compararlo con el resultado obtenido y determinar si la prueba tuvo exito o no. (Hay que tener en cuenta que las pruebas no deben generar riesgo)

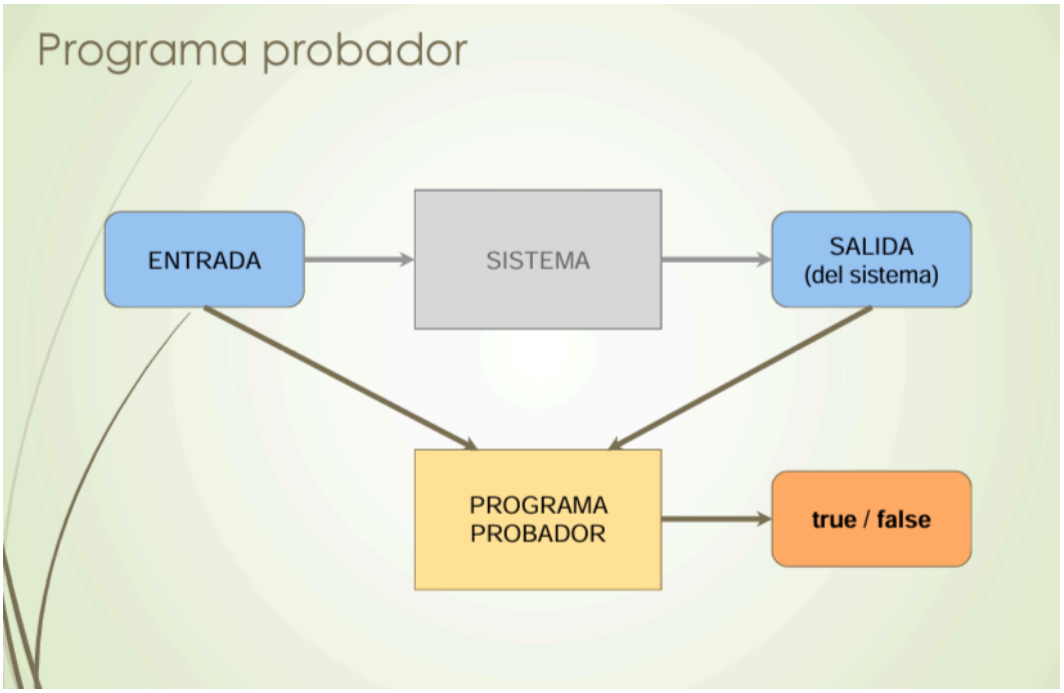
Luego a las pruebas las podemos dividir en 2 TIPOS:

- **CAJA NEGRA**(QUE): Se prueba desde el exteriror, sin tener detalles de que es lo que ocurre en el interior. Se compara dado una entrada que nos da de salida, pero no sabemos el mecanismo interno que produce esa salida, solo que tal entrada da tal salida.

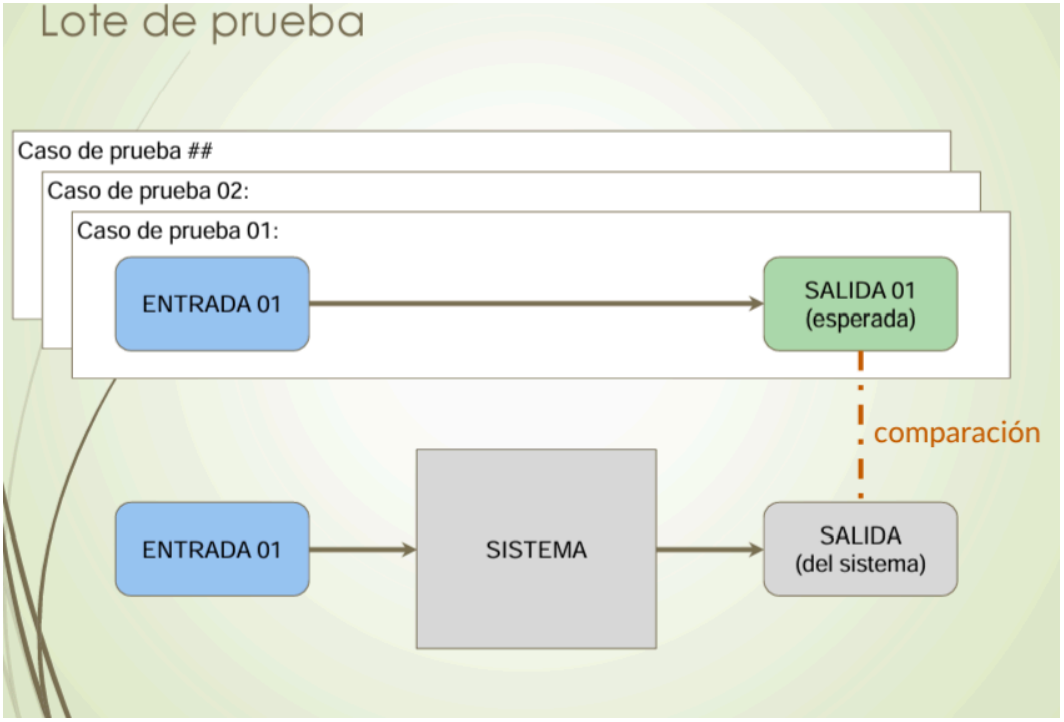


EJ:

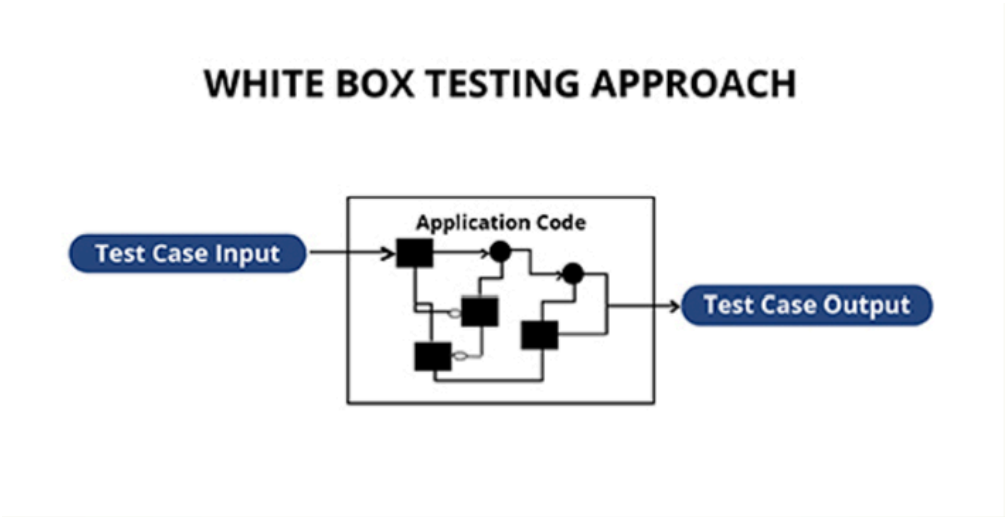
- Pruebas funcionales(testeo funcionalidades)
- Pruebas de estres(testeo limites)
- **Programa probador**: Este se utiliza cuando tenemos un programa con **multiples salidas**, entonces hacemos un mini programa que compara la entrada que se le da al sistema y la salida que el sistema nos da, entonces este nos devuelve si la prueba fallo o no, PERO hay que tener en cuenta que este programa probador debe tener una **logica simple** para asi minimizar sus errores.



- **Lote de pruebas**: Es un conjunto de casos, donde dada una entrada busca validar la salida del sistemas con una salida esperada confeccionada desde antes. Cada caso de prueba contiene Nombre significativo del caso, descripcion, entrada y salida esperada.



- **CAJA BLANCA**(COMO): Se prueba teniendo conocimiento de lo que SUCEDE ADENTRO, se conoce el mecanismo que transforma nuestras entradas en salidas(codigo, algoritmos, estructuras de datos).



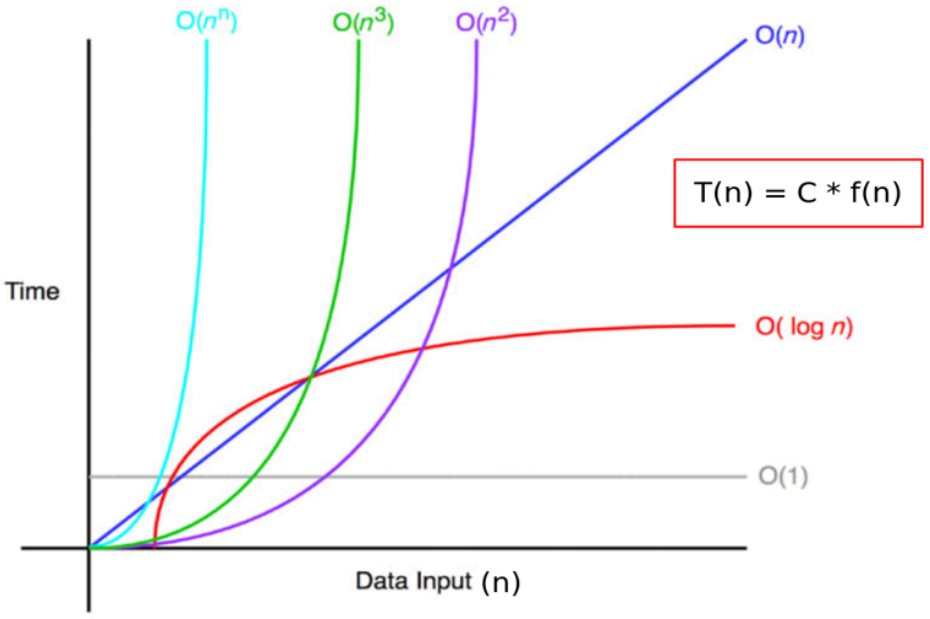
EJ:

- Inspeccion de codigo
- Pruebas unitarias
- Pruebas de integracion
- Pruebas de estres
- etc

Los metodos de prueba que se utilizan en esta materia son los ya explicados Lote de prueba y Programa Probador.

	Lote de Prueba	Programa Probador
Salidas únicas	✓	✓
Salidas múltiples	✗	✓
Identificación de errores	Si falla un caso, se sabe cuál podría ser el problema.	Si falla, no nos brinda información del problema.
Cálculo de salidas esperadas	Manual	Automático (programado)
Complejidad del problema original	Funciona para todo tipo de problemas	Debe tener una complejidad mucho menor

COMPLEJIDAD COMPUTACIONAL



La teoria de la complejidad computacional busca relacionar el tiempo con la cantidad de datos, esta estudia unicamente la forma de la curva.

Sigue la siguiente formula: T = k * n / La T representa el tiempo, la k una constante que es la velocidad de la computadora(mientras mas pequeña mas rapida) y la n representa la cantidad de datos.

Para estudiar la complejidad de un programa no hace falta tener todo el codigo ya desarrollado, con saber mas o menos que algoritmos vamos a implementar podemos determinarla, veamos las diferentes curvas de complejidad una por una:

- O(1): COMPLEJIDAD CONSTANTE, el tiempo es siempre el mismo independientemente de la cantidad de datos que estemos manejando(n).
- O(n): COMPLEJIDAD LINEAL, el tiempo aumenta en proporcion de la cantidad de datos que estamos manejando, ej busqueda simple,un for().
- O(n^2), O(n^3), O(n^n): COMPLEJIDAD EXPONENCIAL, el tiempo aumenta al cuadrado, cubo o elevado al exponente que este elevado la cantidad de dato sque estamos manejando, ej ordenamiento burbuja o seleccion, o cuando tenemos varios for anidados unos adentro de otros.
- O(log n): COMPLEJIDAD LOGARITMICA, el tiempo tiende a reducirse cuando se hacen busquedas mas grandes, y estos debido a que en este caso nos encontramos con una busqueda binaria, la cantidad de datos a recorrer se parten a la mitad en cada busqueda asi bajando exponencialmente la cantidad de elementos e iteraciones necesarias del algoritmo. Ej la ya mencionada busqueda binaria, cuando parto de a mitades el n
- O(n*log n): QUICKSORT, este es un caso de algoritmo que es el utilizado por la mayoria de algoritmo de ordenamiento actuales. Se encontraria en el medio entre O(n) y O(n^2)

Pero en un codigo puede ser que se encuentren varias de estas curvas dependiendo la parte que analicemos, entonces como hacemos para calcular la complejidad total? Aqui es donde aplican 2 reglas fundamentales:

- LA REGLA DE LA SUMA: Se dice que tenemos una suma de complejidades cuando tenemos varias complejidades en un codigo una abajo de la otra(ej un for y luego un while y luego otro for, etc) aqui lo que sucede es que la PEOR CURVA DE COMPLEJIDAD PREDOMINA al resto y es la que define la complejidad del codigo entero(ej tengo una busqueda binaria y luego un burbujeo, entonces la complejidad es de O(n^2) ya que esta es la peor.
- LA REGLA DEL PRODUCTO: Se dice que tenemos un producto de complejidades cuando tenemos varias complejidades unas adentro de las otras, entonces aqui lo que sucede es que SE MULTIPLICAN las

complejidades(ej un for anidado con otro for es n*n, y un for con una busqueda binaria adentro es n*log n)

(nota: la recursividad es ineficiente computacionalmente, solo simplifica el codigo)

METODOS DE ORDENAMIENTO

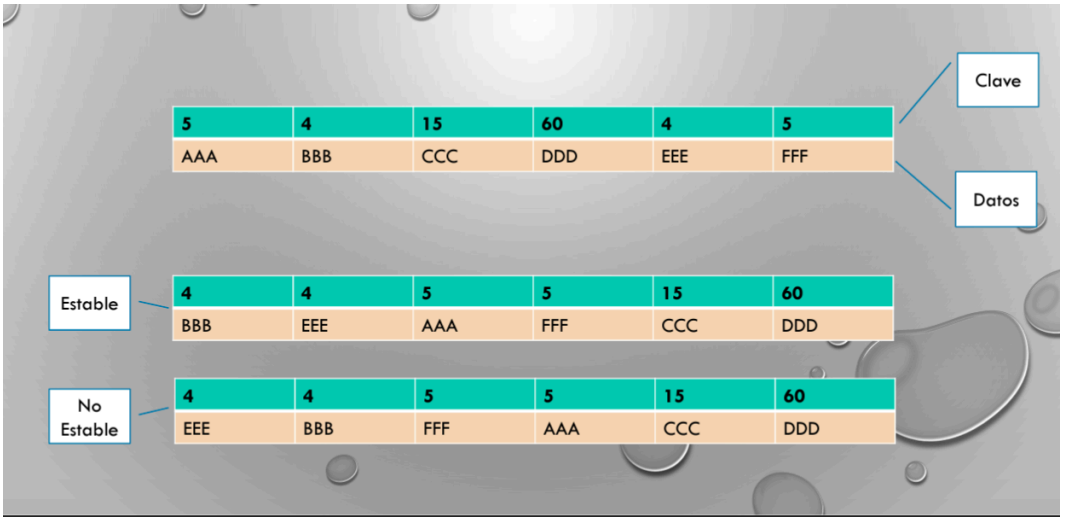
Cuando hablamos de metodos de ordenamiento, evidentemente nos referimos al ordenamiento de datos, pero cuando hablamos de datos debemos tener en cuenta un aspecto importante, estos estan formados por una Clave y Datos,

Datos: Todos aquellos datos que contiene un objeto/registro, por ejemplo en una persona contiene nombre, apellido, dni, edad, etc.

Clave: La clave es el dato que se utiliza para diferenciar al elemento que integra todos los datos dentro suyo, por ejemplo en el caso de persona, una posible clave es el dni.

Teniendo eso en cuenta podemos dar paso a los metodos de ordenamiento. Hay muchos metodos de ordenamiento pero todos tienen ciertas caracteristicas en comun:

- ESTABILIDAD: Los algoritmos de ordenación estables mantienen el orden relativo de los elementos con valores iguales o claves. Los algoritmos de ordenación inestables no mantienen el orden relativo de los elementos con valores/claves iguales.



- SENSIBILIDAD: Un algoritmo puede ser SENSIBLE a la ENTRADA, esto significa que el tiempo del algoritmo varia si la entrada esta mas o menos ordenada, si no es sensible siempre va a tardar el mismo tiempo independientemente de la entrada dada, pero en cambio si es sensible este va a tardar menos si la entrada esta parcial o completamente ordenada.

Ahora vayamos con 3 algoritmos de ordenamiento:

SELECCION

- Selecciona el elemento más pequeño actual y lo cambia de lugar.

1. Encuentra el elemento más pequeño en el arreglo y lo intercambia con el primer elemento.
2. Encuentra el segundo elemento más pequeño y lo intercambia con el segundo elemento del arreglo.
3. Encuentra el tercer elemento más pequeño y lo intercambia con el tercer elemento del arreglo.
4. Repite el proceso de encontrar el siguiente elemento más pequeño y cambiarlo a la posición correcta hasta que se ordene todo el arreglo.

Propiedades

- Complejidad del tiempo: **O(n2)**
- Sensible a la entrada: **No**
- Estable: Si

```
public void selectionsort(int array[])
{
   int n = array.length;           // método para encontrar la longitud del arreglo
   for (int i = 0; i < n-1; i++)
   {
      int index = i;
      int min = array[i];              // tomando el elemento min como i-ésimo ele
      for (int j = i+1; j < n; j++)
      {
         if (array[j] < array[index])
         {
            index = j;
            min = array[j];
         }
      }
      int t = array[index];        //Intercambiar los lugares de los elementos
      array[index] = array[i];
      array[i] = t;
   }
}
```

BURBUJEO

- El algoritmo atraviesa una lista y compara valores adyacentes, intercambiandolos si no están en el orden correcto, esto lo hace multiples veces y al final queda la lista ordenada.

Propiedades

- Complejidad del tiempo: **O(n2)**
- Sensible a la entrada: **Si**
- Estable: **Si**

```
public class BubbleSort {
   static void sort(int[] arr) {
      int n = arr.length;
      int temp = 0;
```

```
        for(int i=0; i < n; i++){
                for(int x=1; x < (n-i); x++){
                        if(arr[x-1] > arr[x]){
                                temp = arr[x-1];
                                arr[x-1] = arr[x];
                                arr[x] = temp;
                        }

                }
        }
```

INSERCION

- Su funcionamiento consiste en el recorrido por la lista seleccionando en cada iteración un valor como clave y compararlo con el resto insertándolo en el lugar correspondiente. Comienza con el indie 1(posicion 2 del array) y lo compara con todos los numeros a la izquierda, luego se mueve al indice 2 y asi sucesivamente realizando las comparaciones e inserciones hasta que queda ordenada.

Propiedades

- Complejidad del tiempo: **O(n2)**
- Sensible a la entrada: **Si**
- Estable: **Si**

```
public int[] insertionSort(int[] arr)
      for (j = 1; j < arr.length; j++) {
        int key = arr[j]
        int i = j - 1
        while (i > 0 and arr[i] > key) {
          arr[i+1] = arr[i]
          i -= 1
        }
        arr[i+1] = key
      }
      return arr;
```

PILAS, COLAS Y ARBOLES

Para comenzar hay que entender que las pilas, colas y arboles son TIPO DE DATOS ABSTRACTOS(TDA). Un Tipo de Dato Abstracto (TDA) es un modelo que define **valores** y las **operaciones** que se pueden realizan sobre ellos. Y se denomina **abstracto** ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la **representación interna** o bien el **cómo** están implementadas las operaciones. Los que nos importa es como utilizarlos, asi que veamoslos:

(nota: la complejidad no es características de la estructura, si no de la implementación.)

PILA

La PILA o tambien llamada STACK es una estructura del tipo LIFO(LAST IN, FIRST OUT), como solo podemos acceder por arriba de la pila para añadir o quitar un elemento, entonces el valor que se encuentra encima de todos se le llama TOPE DE PILA(TOS), este valor es el unico visible y accesible.

La pila tiene 3 metodos fundamentales:

- .peek(): Nos devuelve el tope de la pila
- .push(): Añade un elemento a la pila(apila)
- .pop(): Quita un elemento a la pila(desapila)

Como el unico elemento accesible es el tope se puede afirmar que la pila tiene una COMPLEJIDAD COMPUTACIONAL CONSTANTE de O(1), ya que el tiempo de la operacion no depende de cantidad de elementos.

La pila puede ser implementada usando arrays o listas enlazadas. Es utilizada normalmente por los sistemas operativos para el manejo de inturrupciones entre otra cosas.

COLA

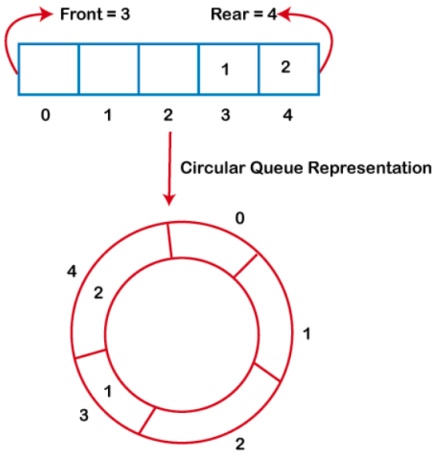
La COLA o tambien llamada QUEUE es una estructura del tipo FIFO(FIRST IN, FIRST OUT), las colas tienen un fondo/back por donde ingresan los datos y un frente/front por donde salen los datos(se puede decir que los datos ingresan

por la izquierda y salen por la derecha respetando el orden en el cual entraron) y solo es visible el frente de la cola.

La cola tiene 3 metodos fundamentales:

- .peek(): Nos devuelve el frente de la cola
- .enqueue(): Añade un elemento a la cola
- .dequeue(): Saca un elemento de la cola

La cola tiene diferentes tipos de implementaciones(punteros, listas enlazadas, etc), en su implementacion mas basica tiene una COMPLEJIDAD COMPUTACIONAL LINEAR de O(n) ya que se dezplaza n elementos en cada operacion. Pero existe una cola mas OPTIMIZADA, en esta lo que hacemos es que implementamos un vector donde el ultimo elemento del vector apunta al primero(llega al final y vuelve al comienzo), de esta manera logramos alcanzar una COMPLEJIDAD LINEAR CONSTANTE de O(1).



COLA DE PRIORIDAD

Una COLA DE PRIORIDAD es una cola como la ya mencionada anteriormente, salvo que entes caso cada elemento tiene asociado una prioridad. Se insertan elementos como el cualquier otra cola, pero a la hora de la extraccion se extraen respetando las prioridades(si hay 2 de la misma prioridad entonces si se respeta fifo).}

La COMPLEJIDAD COMPUTACIONAL de la cola de prioridad varia dependiendo si:

- COLA DE PRIORIDAD DESORDENADA: La cola se encuentra desordenada por lo que insercion es de O(1) como la cola normal, pero a la hora de la

extraccion pasa a ser de O(n) ya que debe buscar uno por uno cual es el elemento de mayor prioridad.

- COLA DE PRIORIDAD ORDENADA: La cola se encuentra ordena por lo que la insercion es de O(n) ya que cada vez que entra un elemento lo ordena teniendo en cuenta las prioridades, y luego la extraccion es de O(1) ya que sale el que esta primero en la cola.

ARBOLES

El ARBOL es una estructura del tipo JERARQUICA, cada arbol nace de un nodo o raiz principal y desciende en otros nodos que pueden descender en mas o no descender en nada y quedar como hojas. Los arboles pueden ser del tipo BINARIO, aqui cada arbol nace de una raiz y luego tiene un subarbol a su izquierda y un subarbol a su derecha(y asi sucesivamente), no obstante tambien existen arboles mas elementales como que tenga un solo subarbol de alguno de los lados o que haya un solo nodo y el arbol este vacio.

Una implementacion del arbol binario en java seria:

```
public class Nodo {
    private Info informacionDelNodo;
    private Nodo nodoIzquierdo;
    private Nodo nodoDerecho;
}
```

Los arboles binarios se pueden recorrer de 3 maneras:

- PRE-ORDEN(RID): RAIZ-IZQUIERDA-DERECHA
- POST-ORDEN(IDR): IZQUIERDA-DERECHA-RAIZ
- IN-ORDEN(IRD): IZQUIERDA-RAIZ-DERECHA

El post-orden es interesante porque es como se organizan las operaciones aritmeticas para procesarlas en la cpu. El in-orden es interesante porque al pasar a un arbol binario de busqueda a este recorrido nos lo devuelve ordenado.

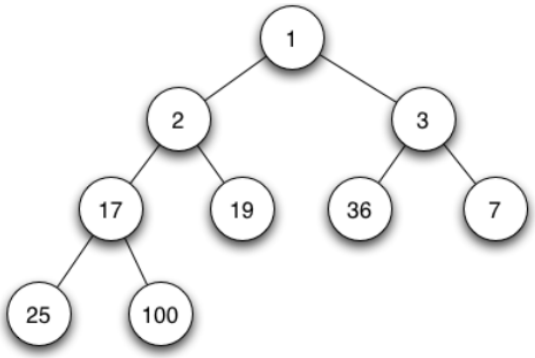
La manera de organizar los arboles binarios de busqueda suele ser que lo menor que la raiz se encuentre a la izquierda y lo mayor a la derecha. Pero es posible que el arbol se degenere si hay mayor cantidad de nodos menores o

mayores(tiende a irse para un costado), en este caso las busquedas van a tardar mas ya que no se descarta la mitad del arbol en cada busqueda, por eso si queremos recuperar esta eficiencia debemos balancear el arbol(un arbol balanceado es aquel que tiene una cantidad igual o parecida de nodos en ambos lados).

HEAP

El HEAP o MONTICULO es un tipo de arbol especial puesto que debe cumplir de 2 CONDICIONES:

1. El arbol debe estar COMPLETO o CASI COMPLETO, es decir, todos los niveles deben estar completos o el ultimo nivel debe tener todas sus hojas de izquierda a derecha sin huecos



2. El arbol tiene que estar definido por MINIMO o MAXIMO:
    - Si el arbol es MINIMO, entonces el valor clave siempre es menor que todos sus descendientes.
    - Si el arbol es MAXIMO, entonces el valor clave siempre es mayor que todos sus descendientes.

INSERCION: COMPLEJIDAD O(log n):

Cuando quiero agregar un valor nuevo a este arbol, para no romper con estas condiciones lo que se hace es lo siguiente, se agrega el nuevo nodo abajo a la izquierda(el ultimo de izquierda a derecha) y luego se va comparando e intercambiando con todos los demas nodos hasta encontrar donde debe ir su valor.

EXTRACCION: COMPLEJIDAD O(log n):

El nodo que quiero eliminar es reemplazado con el nodo de abajo a la izquierda(el ultimo de izquierda a derecha) este que reemplazamos se elimina y luego se vuelve a ordenar el arbol comparando e intercambiando.

Ahora, cuando queremos implementar el monticulo en la programacion, la manera mas eficiente es con un VECTOR(sin listas ni nodos o mas referencias), pero esto es solo cuando el ARBOL esta COMPLETO, en ese caso existe la siguiente relacion matematica:

Si el PADRE esta en la posicion “i”:

- El HIJO IZQUIERDO va a estar en 2*i
- El HIJO DERECHO va a estar en (2*i)+1

Si un elemento esta en la posicion “i”:

- El PADRE va a estar en la posicion (int) [i/2]

(tener en cuenta que el indice 0(cero) no se utiliza)