Programación 3

Nombre: Santiago Vietto

Docente: Matías Murgui

Institución: UCC

Año: 2020

Unidad de Repaso

Clase abstracta: clase en la cual sus métodos no están implementados, es decir, no tienen codigo, para que las clases que heredan de dicha clase implementen esos métodos. Se considera una clase abstracta cuando tienen por lo menos uno de sus métodos virtual puro, y este se define como tal cuando se utiliza la palabra reservada *virtual* antes de la definición del método y la función se iguala a cero.

Función con const: cuando declaramos una función que retorna valor como constante, no declaramos que en si la función misma es constante sino que declaramos que todo lo que se ejecute en una función no va a modificar el método que lo está implementando, es decir, la función no va a poder modificar ninguno de los atributos que tenga la clase, únicamente va a poder devolverlos/mostrarlos.

this: hace referencia al objeto de la clase que está instanciándose, y no un objeto en particular.

_ Los include para herencia dependen del compilador, a veces se toma .h o .cpp.

Recursividad

_ Cuando un trata de desarrollar un algoritmo recursivo es tener en claro cuál va a ser el caso de salida o base, en donde la recursividad frene y se obtenga el resultado. el caso base es la condición con la cual la recursión deja de ser recursiva.

- Unisgned int: declaramos enteros positivos para evitar errores. Podemos declarar tanto los valores devueltos como los pasados por parámetro.

_ El propósito de la recursividad es invocar a la misma función de donde se está invocando una y otra vez. Básicamente es para implementar ciclos explícitos como for, while, do while. Entonces de esta manera nos queda un codigo super elegante para utilizar, donde si lo hacemos con bucles quizás queda rebuscado y difícil de mantener. La recursividad proporciona dos o tres líneas de codigo que facilitan la lectura y comprensión, que de otra manera debería ser un codigo más grande.

_ El nombre de funciones y variables también es importante que sean lo más explícito que se puedan. Se recomienda ademas poner variables en inglés.

Ejercicio 1: en los parámetros de la función pasamos como parámetro el valor de la base y el exponente de la misma. El caso base será cuando el exponente valga cero y el resultado es uno sin importar cuanto valga la base. En el caso recursivo tenemos que multiplicar la base de la potencia por sí misma todas las veces que corresponda, entonces invocamos la función potencia de nuevo reduciendo el exponente una unidad cada vez que se llama. Por ejemplo si la base es 2 y el exponente es uno el procedimiento seria:

$2^3 = 2 * (2, 3-1) = 2 * 2 * (2, 2-1) = 2 * 2 * 2 * (2, 1-1) = 2 * 2 * 2 * 1 = 8$

Ejercicio 2: en los parámetros de la función pasamos un puntero que es lo mismo que un arreglo sin tamaño fijo y tenemos el tamaño del arreglo. La variable size son las posiciones del arreglo. Nuestro caso base seria si el size es cero, es decir, tenemos una única posición, la suma de todos los valores de ese arreglo va a ser igual al valor que este en la posición cero del arreglo. En el caso recursivo tenemos el valor en la posición del valor del size más el siguiente valor del arreglo que será el que estará en una posición detrás por eso restamos el size menos uno. Por ejemplo si viene un arreglo con dos valores el procedimiento seria:

arr={2, 6}

6 + (arr, 1-1} = 6 + 2 = 8

_ En el main, si se le pasa el tamaño total del arreglo a la función, esta devuelve un valor sin sentido, haciendo referencia a un valor no existente, ya que cuando se intenta indexar un arreglo en una posición mayor a su máximo (size), se está intentando ingresar a un valor de memoria no inicializado. Si se le pasa como parámetro (size - 1), la función devuelve el valor deseado, porque queremos mandarle la última posición.

Ejercicio 3: el algoritmo de Ackermann es un algoritmo que básicamente devuelve el número de combinaciones de M objetos tomados de N objetos. Esta versión del ejercicio es únicamente válida para valores enteros positivos nada más. Para resolverlo, los casos base y recursivos serían los siguientes:

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0; \\ A(m-1,1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m-1, A(m,n-1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Ejercicio 4: buscamos en internet los casos para la solución del algoritmo de Euclides, y que son los siguientes:

$$\mathrm{mcd}(x, y) = \begin{cases} x & \text{si } y = 0 \\ \mathrm{mcd}(y, \mathrm{resto}(x,y)) & \text{si } y > 0 \end{cases}$$

Ejercicio 5:

1)_ Tenemos que calcular el cociente de forma recursiva, para el caso base siempre que el divisor (b) sea mayor al dividendo (a) devuelve uno, de lo contrario para el caso recursivo donde restamos a-b para sacar una unidad de multiplicación al valor “a” donde llegamos

al caso de que “a” sea menor a “b” donde devuelve cero, y el uno hace referencia a las veces que b entra en a. Entonces una solución sería por ejemplo:

12/5 (a=12 y b=5)

1 + (12-5, 5) = 1 + 1 + (7-5, 5) = 1 + 1 + 0 = 2

2)_ Para calcular el resto aplicamos la misma idea que el anterior tanto en el caso base como la parte recursiva. Por ejemplo si hacemos 2 dividido 8 el cociente es 2 porque el cociente es cero y nos queda 2 porque nunca llegamos a cumplir, entonces ese 2 será el valor de “a”, y ese es nuestro caso base, si “a” es menor a “b”. para el caso recursivo aplicamos lo mismo de antes. Por ejemplo:

9/2 (a=9 y b=2)

(9-2, 2) = (7-2, 2) = (5-2, 2) = (3-2, 2) = 1

3)_ Analizando tenemos el cociente y resto de la siguiente división:

1024/10 = 102

1024%10 = 4

_ Si nosotros seguimos llamando recursivamente a la función escribir_espaciado, repetimos lo mismo con el cociente y el resto que obtuvimos y así sucesivamente. Nuestro resto va quedando almacenado de forma temporal para luego mostrarlo de forma espaciado

102/10 = 10

102%10 = 2

10/10 = 1

10%10 = 0

1/10 = 0

1%10 = 1

Ejercicio 6: calculamos el factorial de un número, y para esto analizamos el caso base y la parte recursiva como tenemos a continuación:

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n * (n-1)! & \text{si } n > 0 \end{cases}$$

Listas

_ Una lista al igual que un arreglo está compuesto por campos, pero cada campo en si es otra estructura de datos que va a tener dos partes, una es en donde se almacena el dato en sí (entero, string, carácter, punto flotante, etc) y la otra parte va a ser un puntero al siguiente nodo de la lista. El ultimo nodo apunta a nullptr, porque de esta forma sabemos cuándo llegamos al final de la lista, es decir, si un nodo apunta a null podemos saber entonces que es el último de la lista.

_ Uno de los atributos intrínsecos de la lista es que sabe dónde está el inicio, entonces se guarda el lugar de memoria donde está el inicio, de esta manera cuando una lista es instanciada el único valor que tiene es el inicio donde luego los otros datos se van encadenando o enlazando al próximo dato.

_ La estructura de datos lista está constituida a su vez por otra estructura de datos que es el nodo, el nodo es el componente principal de la lista, este posee el lugar para el dato y el puntero al siguiente nodo.

_ Una lista enlazada es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un “enlace”. La idea básica consiste en construir una lista cuyos elementos, llamados nodos, se componen de dos partes (campos): la primera parte contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado Dato, TipoElemento, Info, etc.), y la segunda parte es un enlace que apunta al siguiente nodo de la lista.

Estructura:

private:

Nodo<T> *inicio; (puntero al inicio de la lista, porque la única forma de saber dónde está la lista es que tengamos la noción de donde está el inicio de la misma).

public:

- Constructor(): inicializa en NULL ya que al inicio la lista está vacía, al menos que la copiemos pasándola por parámetro.

- Constructor(Lista): constructor por copia. Para copiar no hace falta copiar todos los datos de la lista li que vienen por parámetro, ya que como la lista por parámetro es de tipo constante, cualquiera sea la lista que pasemos ya sea por puntero o por referencia, el const resguarda de que no vayamos a modificar la lista que se pasa por parámetro. Entonces para copiar una lista pasada por parámetro lo único que tenemos que hacer es decirle a la lista nueva a donde arranca la lista que pasamos

por parámetro, es decir, que el inicio de la lista nueva sea igual al inicio de la lista pasada por parámetro y así se enlazan.

- ~Lista(): destructor, que va a llamar a la función vaciar().

- esVacia(): verifica si la lista está vacía, si esto es así devuelve true. Para saber si está vacía preguntamos si inicio apunta al puntero nulo, entonces no tenemos nada todavía.

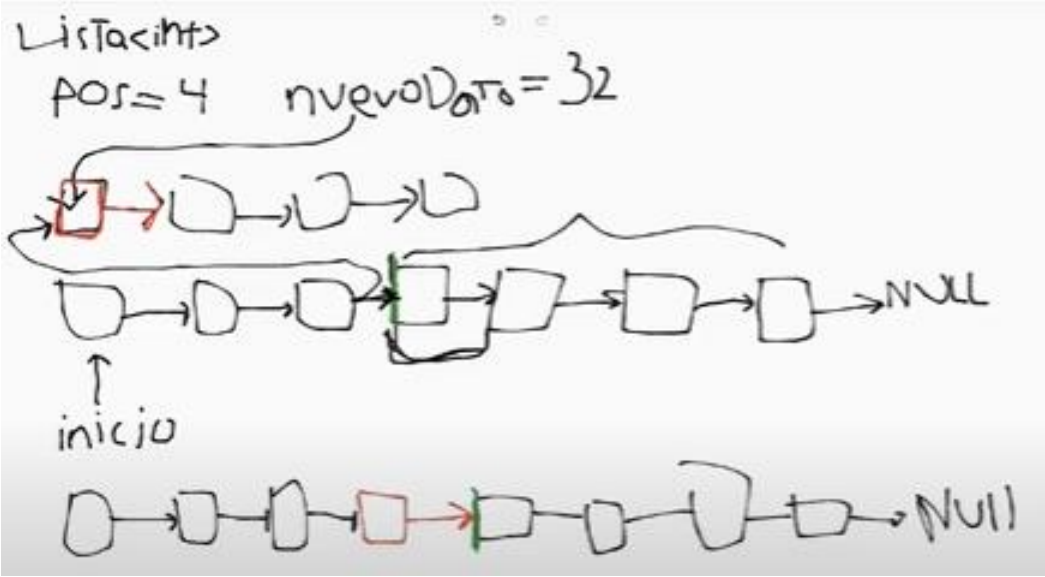
- getTamanio(): devuelve la cantidad de nodos que tiene la lista. Para saber cuántos nodos tiene nuestra lista, lo que hacemos es iterar sobre cada uno de los nodos que tiene la lista con un contador para llevar un registro de cuanto nodos tenemos. Necesitamos dos variables auxiliares una que sea contador y otra que sea un puntero que apunte primero al inicio de la lista y luego que vaya apuntando al siguiente para saber cuándo llegamos al final. Para movernos con el puntero a lo largo de toda la lista, hacemos un bucle en el que siempre y cuando la variable auxiliar no sea el puntero nullptr, trae la siguiente variable auxiliar que va a estar enganchada al inicio y lo registramos en la nueva variable con el método.

- getSiguiente() del nodo, para que de esta manera cada vez que hagamos una iteración vamos a incrementar el tamaño de nuestro contador. Cuando llegamos al final devolvemos el tamaño.

- insertar(posición, dato): insertar un dato en la posición que digamos o definamos. Tenemos por ejemplo una lista de 5 elementos y queremos insertar un nuevo dato en la posición 2. Nuestro nuevo dato va a ser un nodo nuevo, que va a tener el nuevo dato y el puntero al siguiente nodo que por el momento no apunta a nada (nullptr), entonces lo que hacemos es llegar a la posición del nodo anterior al que queremos llegar, tomamos el puntero de ese nodo y en vez de hacerlo apuntar al siguiente nodo de la lista, lo hacemos apuntar al nuevo nodo, pero antes de hacer esto tenemos que guardar el lugar de memoria del nodo en la posición 2 porque si lo perdemos perdimos la lista, entonces almacenamos ese lugar de memoria en una variable auxiliar donde el puntero del nuevo nodo va a apuntar a esa variable auxiliar, entonces de esta manera la lista queda con un nuevo dato en la posición 2 y el valor que estaba en la posición 2 se desplazó para adelante en la posición 3.
  - Si queremos ingresar un dato en la posición cero, no tiene sentido que llamemos al bucle para iterar sobre las posiciones de la lista, ya que si sabemos que si la posición es cero, al nuevo nodo que declaramos lo instanciamos como el nuevo nodo y le insertamos la componente dato que va a llevar. El siguiente nodo de este nuevo nodo va a ser el que apunte al inicio. De esta manera le decimos al nuevo nodo que se fije dónde está el inicio de la lista que ya existe

para que apunte ahí porque va a ser el nuevo inicio, entonces el siguiente del nuevo nodo es el inicio y el inicio de esta lista pasa a ser el nuevo.

- En el caso de que la posición no sea cero, en un bucle tenemos la condición de que mientras no lleguemos al final de la lista queremos que frene en la posición a la que queremos llegar, y a esto lo logramos haciendo que la posición actual sea menor a la posición menos un lugar ya que queremos llegar justamente al lugar anterior, si esto es así desplazamos el nodo auxiliar a lo largo de la lista. Tenemos que verificar si la variable auxiliar llega a nullptr para tirar un error, ya que no existe la posición en la lista, pero si no ocurre esto, declaramos un nuevo nodo en donde queremos que el siguiente nodo al cual este apuntando sea el siguiente nodo al cual está apuntando el nodo donde estamos parados, entonces de esta manera insertamos un nuevo nodo sin perder ningún dato.

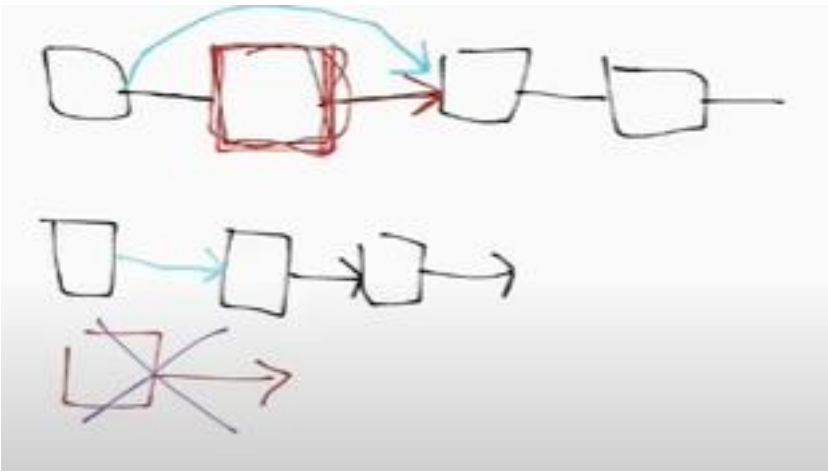


- insertarPrimero(dato): insertar un dato en el inicio de la lista. Es la misma lógica que el método anterior, pero insertamos solamente en la primera posición, donde llamamos a la función insertar y le pasamos por parámetro la posición cero y el dato que pasamos por parámetro.

- insertarUltimo(dato): insertar un dato al último de la lista. Acá necesitamos iterar sobre la lista que tenemos hasta al final para saber dónde estamos ya que tenemos que estar parados en el último nodo. Acá tenemos que ir hasta el último lugar. Primero creamos el nodo en la estructura de tipo de dato nodo.
  - Si la lista está vacía, que puede ser una probabilidad, al nuevo nodo lo que hacemos es ponerlo en el inicio de la lista y el inicio de la lista pasa a ser el nuevo.

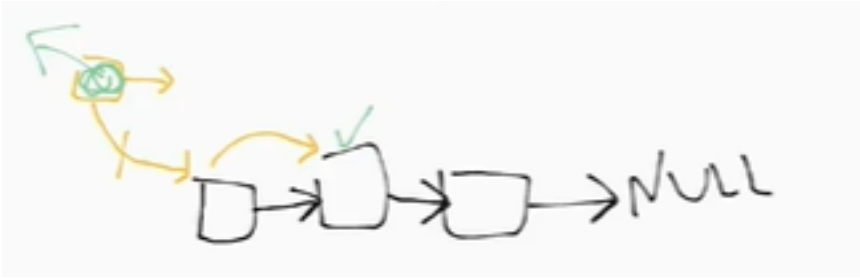
- En el caso en el que la lista no este vacía si necesitamos ir hasta el final. Donde siempre y cuando el siguiente valor al cual estemos apuntando sea distinto de nullptr, actualizamos la variable auxiliar. Entonces de esta manera iteramos para llegar al final de la lista.
- Una vez que llegamos al final de la lista queremos que el siguiente dato del nuevo sea el siguiente del cual estamos parados que en teoría debería ser el puntero nullptr, y donde estamos parados le decimos que apunte al nuevo nodo.

- remover(posición): remover un dato de la lista. Suponemos una lista de 4 elementos y queremos eliminar el nodo de la posición 1, para hacer esto lo que tenemos que llegar a la posición anterior que está apuntando al nodo al que queremos eliminar y hacemos que ese nodo en vez de que apunte al nodo que queremos eliminar, apunte al nodo siguiente del que queremos eliminar, en donde ahora la lista nos queda con 3 elementos. Si bien ya sacamos el nodo de la lista, aún sigue almacenado en memoria y tenemos que eliminarlo y lo hacemos con el operador delete.



  - Si la posición en la que queremos borrar es cero, o sea el inicio, el inicio va a pasar a ser el siguiente nodo o el segundo al cual estamos apuntando, entonces el siguiente nodo va a pasar a ser el siguiente del inicio actual. Para desalojar la memoria usamos el operador delete.
  - En el caso de que la posición no sea cero y no hayamos llegado al final de la lista, el nodo a borrar va a ser el siguiente al cual estemos apuntando y ahora queremos que el nodo en el que estamos parados apunte al siguiente del que esta borrado, es decir, hacemos el enlace al que esta por delante del que queremos borrar. Luego desalojamos de memoria la variable a borrar y no aux porque es un dato existente en la lista.

- getDato(posición): devuelve el valor del nodo al cual apuntamos. Vamos a iterara sobre la lista hasta llegar a la posición que queremos leer. Entonces tenemos un nodo auxiliar para iterar sobre la lista y tenemos un iterador para tener una referencia de donde estamos parados en la lista. No nos hace llegar hasta la posición anterior del nodo que buscamos sino que literalmente necesitamos llegar a él, por ende nos desplazamos y llegamos al punto que queremos, y si la posición del puntero no es nullptr devolvemos simplemente el valor de la variable auxiliar.



- reemplazar(posición, dato): cambiar una dato que ya exista en la lista. Va a ser igual que la función getDato solo que cambia al final. A la variable auxiliar le seteamos con la función del nodo que es setDato y pasamos el dato que queremos, y de esta manera se sobrescribe el dato que estaba antes y lo reemplazamos con el nuevo.

- vaciar(): eliminar datos o nodos de la lista. Este va a remover todos los elementos de la lista, y es el método que va a llamar el destructor. Tenemos nuestro nodo inicial y un nodo que es cada elemento de la lista que vamos a borrar. Entonces siempre y cuando el nodo auxiliar sea distinto de nullptr, hacemos que el nodo a borrar sea el auxiliar que recorre la lista, hacemos que la variable auxiliar sea la iteración al próximo nodo y borramos el nodo a borrar, y así iterativamente hasta llegar al final. Cuando llegamos al final, para vaciar la lista y que quede estable tenemos que hacer que inicio apunte a nullptr para que siga siendo una lista.

- print(): sirve para imprimir los datos de cada nodo. Tenemos un nodo auxiliar que itera a través de la lista y a medida que avanza lee el dato de dicho nodo y lo imprime por pantalla.

_ Para objetos punteros se utiliza ->, ej: int *aux aux=aux->Metodo()

_ Para objetos no punteros se utiliza . , ej: int aux aux=aux.Metodo()

Ejercicio 1: para invertir la lista básicamente usamos la lógica de que el inicio apunte al último y el ultimo al inicio, luego que el segundo apunte al penúltimo y que el penúltimo apunte al segundo y así sucesivamente. La función retorna void y no int, porque no

creamos una nueva lista invertida sino que es la misma la que se invierte y se muestra, por lo que también pasamos la dirección de memoria de la lista.

_ Vamos a tener dos variables auxiliares que van a ser los datos que estamos tomando, cuando estemos cambiando de posición vamos a querer tener registro del primer dato y el dato del último nodo, donde estos datos van a ser de tipo tamplate. Queremos saber el tamaño de la lista que queremos invertir, por ende traemos el tamaño de la lista.

Pilas

_ La pilas es una estructura de datos que nos permite almacenar nuestras variables/datos organizados de una manera en particular. El punto de tener los datos organizados de una cierta manera es, sabiendo que están en una estructura dada, damos por sentado que al momento de buscarlos con una metodología en base a esa estructura, vamos a encontrar el valor que queremos en el lugar donde vayamos a buscarlo directamente, es decir, cuando queramos leer o modificar algún dato, nosotros vamos a saber dónde está o tenemos noción de como encontrarlo. Esta es la razón por la que ordenamos las cosas.

_ A diferencia de las listas, no tenemos acceso a cada uno de los valores de la pila pero desde afuera. Cuando insertamos datos siempre es al inicio, y la pila siempre va a leer el ultimo valor ingresado. Las pilas aplican a la característica LIFO (last in first out), o sea el último dato que ingreso, es el primero en salir.

_ Una pila es una colección ordenada de elementos en la que pueden insertarse y

suprimirse elementos por un extremo llamado TOPE.

_ La pila no es más que un tipo de lista, donde tenemos nodos, el atributo inicio que es tope, no podemos remover del medio, y el ultimo nodo apunta a nullptr. La clase nodo que utilizamos antes es la misma para ahora.

Estructura:

private:

Nodo<T> *topNode; (al igual que en listas, vamos a tener un puntero a un nodo apuntando hacia el tope de la pila)

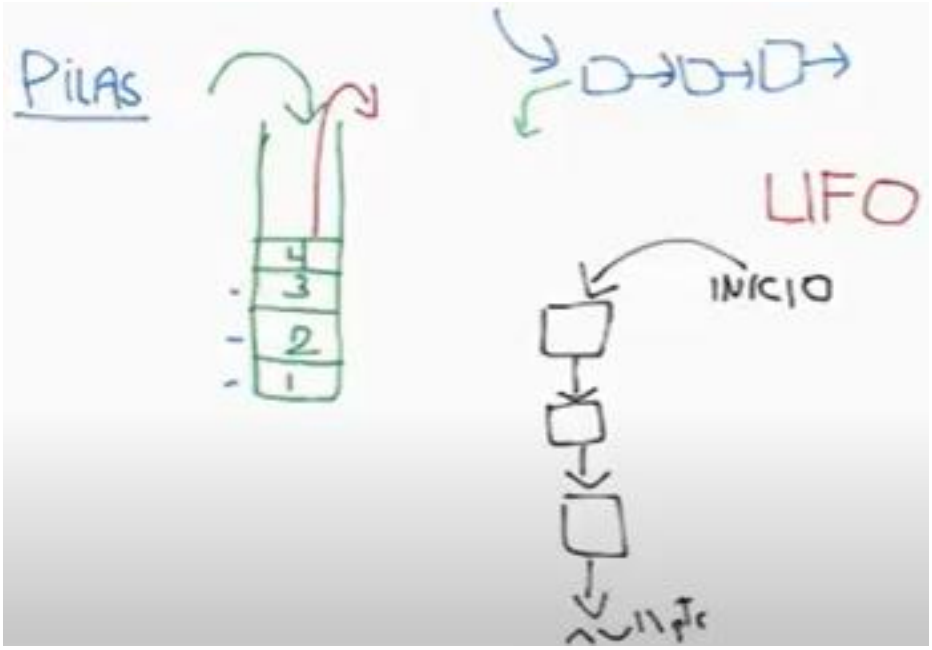
public:

- Constructor(): por defecto cuando inicializamos nuestra estructura, la pila va a estar vacía por lo cual el topNode va ser igual a nullptr.

- ~Pila(): llama a al método pop mientras el tope de la pila no apunte a nullptr. Este va a eliminar de memoria el objeto que instanciemos, entonces llamamos al método pop hasta que la pila este vacía y desalojamos la memoria.

- esVacia(): devuelve un bool si la lista está vacía o no. Nuestra pila está vacía cuando el único valor que tenemos en la pila está apuntando hacia nullptr, es decir, si nuestro tope apunta al puntero nulo entonces la pila está vacía. Le preguntamos al tope de la pila si apunta al puntero nulo o no, donde esto devuelve true o false.

- push(dato): es lo mismo que el método insertar en listas, este método va insertar un valor desde el tope de la pila. Insertamos un dato por parámetro. Necesitamos un nuevo nodo y lo inicializamos, porque un nuevo dato es un nuevo nodo. Los nodos tienen dos atributos, el dato y el puntero al siguiente nodo, entonces primero seteamos el dato que es el que queremos meter en la pila. Si nosotros estamos insertando un valor en la pila, en el que siempre insertamos valores al inicio, significa que el nuevo nodo que estemos insertando va a ser siempre el viejo inicio de la pila, entonces nuestro nuevo nodo va a estar siempre apuntando a el tope actual (topNode). Ahora lo que queremos hacer es que el tope de nuestra pila apunte al nuevo nodo que es ahora el nuevo tope de la pila porque es el último que agregamos.



- pop(): devuelve el tope de la pila y lo saca de la estructura (lo elimina). Por un lado va a devolver nuestro dato que es de tipo template, luego necesitamos un nodo auxiliar que nos permita a nosotros tomar ese último nodo que tenemos y después

tenerlo para eliminarlo, entonces este nodo auxiliar apuntara al tope o inicio de la pila.
    - Si queremos sacar un valor de una pila que está vacía, en este caso nos tiene que saltar un error porque no hay ningún valor.
    - Entonces como queremos mostrar el dato ademas de borrarlo, tenemos que la variable data va a ser igual al valor que tenga el tope de la pila. Luego lo que hacemos es pasar el siguiente nodo del tope que queremos sacar como nuevo tope de la pila, es decir, asignamos el tope de la pila al siguiente nodo. Desalojamos memoria del nodo que estaba al inicio que guardamos previamente en la variable auxiliar. Devolvemos el dato.

- peek(): básicamente es lo mismo que el método pop pero en este caso no eliminamos el dato, solo lo mostramos.

- getTamanio(): devuelve la cantidad de nodos que tiene la pila. Para saber cuántos nodos tiene nuestra pila, lo que hacemos es iterar sobre cada uno de los nodos que tiene la pila con un contador para llevar un registro de cuanto nodos tenemos. Necesitamos dos variables auxiliares una que sea contador y otra que sea un puntero que apunte primero al tope de la pila y luego que vaya apuntando al siguiente para saber cuándo llegamos al final. Para movernos con el puntero a lo largo de toda la pila, hacemos un bucle en el que siempre y cuando la variable auxiliar no sea el puntero nullptr, trae la siguiente variable auxiliar que va a estar enganchada al tope y lo registramos en la nueva variable con el método.

- insertAfter2(): aplicamos similar al de listas.

Colas

_ Esta estructura es muy parecida a la de pilas, pero esta responde a la propiedad FIFO (first in first out). Es decir Una COLA es una colección ordenada de elementos de la que se pueden borrar elementos en un extremo (FRENTE de la cola) o insertarlos en el otro (FINAL de la cola). La estructura mínima que vamos a usar es el nodo así que seguimos con la misma de antes. Entonces tenemos el nodo con su valor y su puntero apuntando al siguiente nodo. En la estructura colas tenemos dos propiedades intrínsecas, es decir debemos saber el inicio y el final de la cola, para poder retirar los datos de ese lugar. Entonces vamos a tener el frente que apunta al inicio de la cola y tenemos el fondo apuntando al último nodo que a su vez el ultimo apunta a nullptr.

Estructura:

private:

Nodo<T> *topNode, *bottomNode; (necesitamos dos punteros a nodo tanto para el frente como para el fondo de la cola).

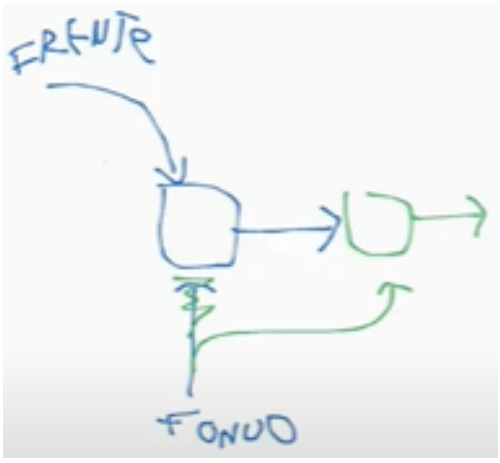
public:

- Constructor(): su propósito es inicializar la estructura. Cuando inicializamos la estructura sin ningún nodo tanto topNode como bottomNode se inicializan en nullptr.

- Destructor():

- esVacia(): método booleano que nos va a decir si la estructura está vacía o no

- encolar(dato): este método se va a encargar de agregar datos a la estructura. Es como el método push de pilas. Cuando se inserta un nuevo nodo vamos a ponerlo en el fondo, para que siempre tengamos disponible el frente, es decir, el primero que insertamos. Entonces el ultimo nodo ahora apunta al nuevo nodo que agregamos que a su vez apunta a null, y tiene asignado el fondo. Siempre el que insertemos al último va a ser el último que podamos retirar de la cola.
  - Lo primero que tenemos que hacer es crear un nodo con su nuevo dato y su puntero con el cual va a estar alineado a los siguientes, como sabemos que ese nuevo nodo va a ir al fondo por las propiedades de la estructura, siempre ese nuevo nodo va a estar apuntando hacia el puntero nulo.
  - Al momento de encolar un dato tenemos que considerar el caso de que la estructura este vacía en el momento, donde si la estructura está vacía significa que el fondo es el puntero nulo, no es que está apuntando al puntero nulo sino que el fundo es el puntero a nullptr. De ser así la estructura va a pasar a ser un solo nodo, entonces tanto el topNode como el bottomNode van a apuntar al único nuevo nodo. Ahora si la cola no está vacía queremos que el frente siga apuntando al primero que agregamos, entonces decimos que el ultimo nodo o el fondo anterior que teníamos queremos que apunte al nuevo nodo que se agrega. Luego de hacer esto queremos que el ultimo nodo sea igual al nuevo



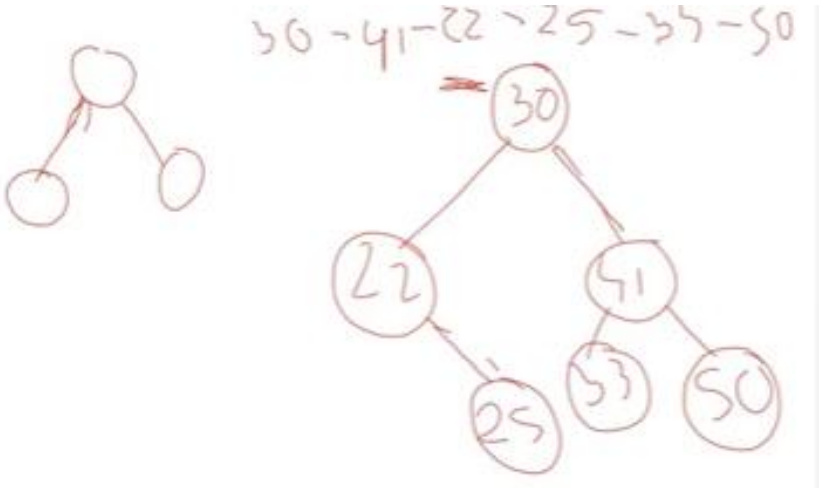
- Desencolar: permite sacar elementos de la cola. Parecido al método pop en pilas. Cuando queramos sacar algo de la estructura vamos a ver únicamente el frente que es el primero que insertamos. Por un lado va a devolver nuestro dato que es de tipo template, luego necesitamos un nodo auxiliar que nos permita a nosotros tomar el primer nodo que tenemos y después tenerlo para eliminarlo.
    - Si queremos sacar un valor de una pila que está vacía, en este caso nos tiene que saltar un error porque no hay ningún valor.
    - Ahora, necesitamos el dato del frente que es el que apunta al primer dato que insertamos. Para eso hacemos que nuestra variable auxiliar a quitar sea igual al tope de la cola, y hacemos la referencia al nodo que vamos a borrar justamente. Ahora le tenemos que asignar el frente al segundo valor de la cola, es decir, el que se insertó detrás del que vamos a eliminar.
    - En el caso de que la estructura haya tenido un solo dato, donde si el frente apunta al puntero nulo, tenemos que asegurarnos de que el fondo también apunte a nullptr.
    - Por último devolvemos la variable y desalojamos memoria.

- esVacia(): si la estructura está vacía es porque no posee nodos y con solo saber si el tope o el fondo es nulo, entonces sabremos si está vacía o no.

- ~Cola(): llama a al método desencolar mientras la cola no este vacía. Este va a eliminar de memoria el objeto que instanciemos, entonces llamamos al método desencolar hasta que la cola este vacía y desalojamos la memoria.

Árbol binario

_ Un árbol binario es una estructura de datos que me permite tener dos nodos por cada nodo. Por ejemplo, un árbol binario de enteros nos permite ir almacenando datos que dé a simple vista no está ordenado, pero cuando recorramos el árbol los podemos obtener a cada uno de manera ordenado. Como regla de diseño tenemos que si se inserta un numero entero en un árbol, si no hay nada se crea el primer nodo del árbol, entonces por ejemplo agregamos el primer número que es 30 y pasa a ser la raíz del árbol, luego viene el 45 y preguntamos si es más grande que el 30 y como esto es así lo colocamos a la derecha y se define como el nodo derecho del nodo raíz que es de donde estamos parados, por otro lado si viene un 22 que es más chico que 30, lo pondremos a la izquierda de la raíz. Ahora suponemos que viene el número 25, comparando con la raíz es más chico y va al lado izquierdo, como ya hay un valor lo comparamos ahora con 22 y como es más grande lo ponemos a la derecha. Y así sucesivamente vamos armando el árbol.



_ Al igual que las otras estructuras, siempre tenemos un puntero o referencia de donde arranca la estructura, en el caso de los árboles siempre arrancan por la raíz que es el primer nodo. A partir de este podemos encontrar todos los demás nodos.

Clase NodoArbol: la clase es de tipo template para que podamos instanciar con el tipo de dato que nosotros queramos. Cada nodo está compuesto por un dato de tipo T y luego vamos a tener dos punteros, uno que apunta al nodo izquierdo y otro al nodo derecho, que van a ser también de tipo nodoArbol.

- Constructor(): vamos a tener dos constructores, uno sin parámetros para inicializar los nodos izquierdo y derecho.

- Constructor(T d): También tenemos un constructor que acepta por parámetro un dato, e inicializamos data.

- getData(): por otro lado tenemos que tener los setters y getters clásicos. Nosotros con respecto a la clase no queremos que se modifique nada en el getter, por ende si ponemos la palabra const, este es un método que le dice a la función que no puede modificar absolutamente nada de los atributos de la clase.

- setData(T d): el método set dato no puede tener la palabra const ya que si estamos modificando al setear en el atributo data un nuevo dato pasado por parámetro.

- Tenemos los getters y setters del nodo izquierdo y derecho, donde devuelven los valores, aplicamos const, y seteamos los valores.

_ Como nosotros vamos a estar utilizando métodos recursivos, necesitamos tener acceso a los datos intrínsecos de la clase. Desde un main por ejemplo no podemos acceder a la raíz del árbol porque es un atributo privado, pero para hacer una metodología recursiva necesitamos pasar por parámetros los distintos nodos y utilizar punteros a nodoArbol para

recorrer el árbol. Entonces en casi todos los métodos vamos a tener el mismo método pero públicos que son los que desarrollamos y que generalmente o no aceptan nada o aceptan un dato de tipo T para insertar, y por otro lado vamos a tener un duplicado del método pero privado en este caso, en el cual es sobrecargado con distintas parametrizaciones para que podamos acceder desde el main.

Estructura:

private:

Nodo<T> *root; (necesitamos dos punteros a nodo tanto para el frente como para el fondo de la cola).

public:

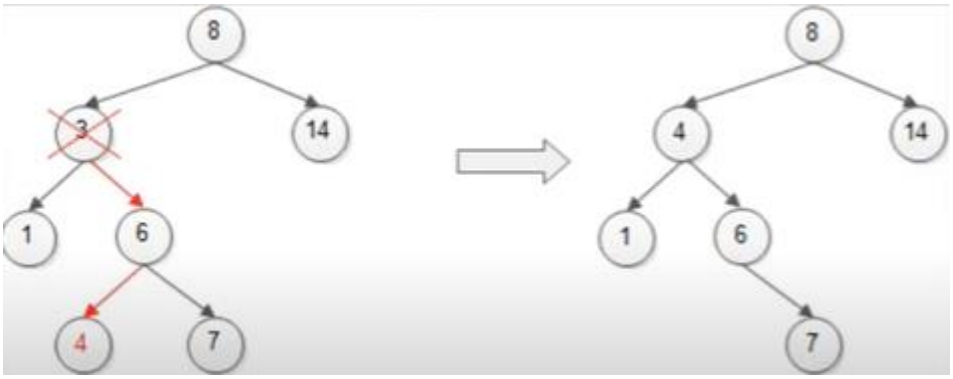
- Constructor(): este lo único que va a hacer es inicializar la raíz en nulo para tener nuestro árbol inicializado.

- search(T data): este método privado de este devuelve un dato de tipo T y recibe un puntero de tipo nodoArbol, esta función que es privada la usamos después en la pública. En el método publico llamamos el privado pasándole por parámetro el atributo raíz (puntero) y el dato. En el método privado preguntamos si la raíz del árbol es nula o no, ya que si es nula significa que el árbol todavía no está instanciado o si lo está pero no hay nada en el por lo tanto tira error. Ahora tenemos lo siguiente:
  - Estamos tratando de buscar un dato dentro de la estructura de un árbol que ya tenemos creado. Entonces tenemos lo primero que tenemos que preguntar es si en el nodo en el que estamos parados, que inicialmente es la raíz, está el dato que buscamos, donde si lo encontramos entonces devolvemos el dato que queremos obtener. Entonces, suponemos que buscamos el dato 30, primero llamamos la función pública y pasamos 30 como dato y la raíz a nuestra sobrecarga privada del método y en este último verifica si r es nulo y esto es falso porque esta creado el árbol, y luego preguntamos si r tiene el valor que estamos buscando, y como instanciamos la raíz, lo encontramos y lo devolvemos. Este es el caso de encontrarlo en la primera instancia.
  - De otra forma preguntamos si el nodo en el que estamos es mayor o menor al valor del dato que buscamos. Si el valor del nodo en el que estamos parados es mayor que el que buscamos entonces significa que puede estar a la izquierda de este nodo, por lo cual llamamos recursivamente de nuevo a esta función pero mandándole por parámetro ahora el nodo izquierdo del cual estamos parados. De otra forma vamos hacia la derecha.

- put(T data): este sería como el método insertar. Necesitamos un método de sobrecarga que sea privado con los mismos parámetros que el método público y ademas devuelve un puntero de tipo nodoArbol, por otro lado acepta el dato que vamos a insertar. Entonces acepta un puntero a un nodo y también devuelve un puntero a un nodo.
  - En el método público, cuando insertemos un nuevo valor vamos a estar modificando la posición de la raíz, por lo cual volvemos a declarar la raíz en base a lo que nosotros creemos, por eso creamos un método privado que devuelve un puntero nodoArbol porque esto es la raíz. Entonces pasamos data y el puntero raíz.
  - En el método privado primero debemos preguntar si el árbol no fue creado, donde esto significa que estamos tratando de insertar un dato en el nodo raíz, entonces creamos un nuevo nodo y le pasamos al constructor por sobrecarga el dato que queremos insertar, lo creemos y devolvemos siendo este nuestra nueva raíz.
  - Si nosotros no queremos que nuestro árbol tenga valores repetidos, es decir, son todos valores únicos, hacemos que si en el nodo en el que estamos parados es igual al que ingresamos entonces salta un error.
  - Ahora tenemos que si el nodo en el que estamos parados es mayor al que estamos tratando de insertar, en ese caso tenemos que insertar el valor del lado izquierdo, y de forma recursiva mandamos la sobrecarga de la función para que vuelva a preguntar. Hacemos lo mismo si el nodo en el que estamos parados es menor entonces va a la izquierda y aplicamos la recursividad.
  - Por último devolvemos la raíz modificada en base al dato que insertamos.

- remove(T data): este es un poco más complejo ya que vamos a hacer una sobrecarga privada y un método más.
  - En el método público, tanto cuando insertamos como cuando removemos un nodo, se modifica lo que posee la raíz o la raíz misma, entonces por eso sobrescribimos la raíz que va a ser igual a la sobrecarga privada en base al dato que queremos sacar y nuestro primer nodo que es la raíz.
  - En la sobrecarga privada va a devolver un puntero de tipo nodoArbol y va a aceptar por parámetro un dato y un nodo.
  - En el caso de las estructuras anteriores para remover un nodo necesitábamos crear una variable auxiliar para que podamos guardar de manera temporal nuestra dirección de memoria. Por lo tanto en el método privado lo primero que vamos a tener es una variable auxiliar donde guardamos el puntero de la referencia que no queremos perder.
  - En el caso si el árbol no existe o está vacío sin nodos, entonces tiramos un error.

- Ahora analizamos el caso donde qué pasaría si estamos en el nodo que contiene el dato que queremos remover, a partir de esto tenemos varias opciones. La primera es el caso donde el nodo donde estemos parados no tenga hojas izquierda ni derecha, acá eliminamos el dato y devolvemos valor nullptr. El segundo es el caso donde el nodo donde estemos parados solo tiene hijo izquierdo, vamos a borrar el nodo en el que estamos pero vamos a almacenar temporalmente lo que está en el nodo izquierdo en nuestra variable auxiliar y luego la devolvemos quedando definido como la nueva raíz. El tercero es el caso donde el nodo donde estemos parados solo tiene hijo derecho es exactamente lo mismo con el anterior solo que el nodo derecho ahora es la raíz. Y el cuarto es el caso donde el nodo donde estemos parados tiene tanto hijos izquierdo como derecho, analizando el grafico vemos que queremos borrar el nodo 3, para esto preguntamos si hay algo en el nodo derecho del que queremos borrar, en caso de que haya tomamos el nodo que está a la izquierda de este último porque sabemos que es menor que el nodo derecho pero mayor ue el que queremos borrar y los cambiamos.

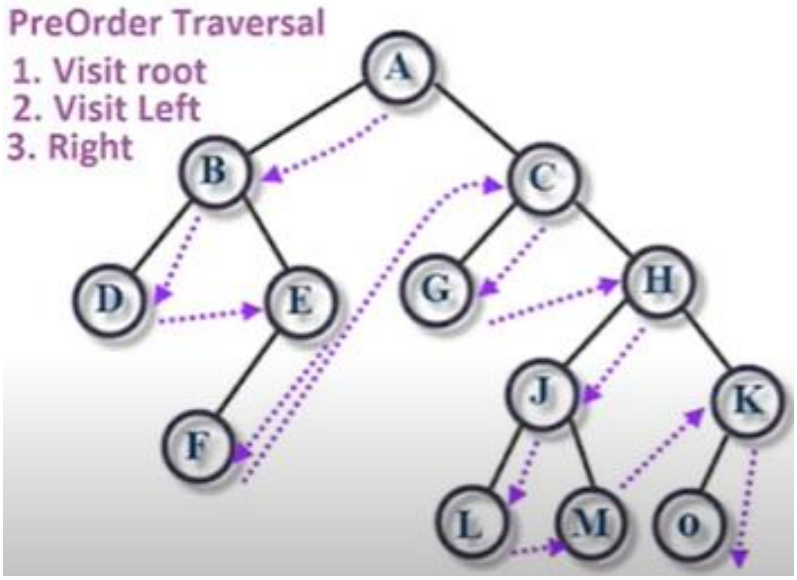


Entonces en el caso de que haya un nodo a la derecha, podemos eliminar el nodo raíz tranquilamente y asignar el hijo izquierdo del derecho del nodo raíz, para hacer esto debemos calcular el máximo y por eso creamos un nuevo método que encuentre el máximo y devuelva un puntero a ese nodo para que tengamos todo lo que hay por debajo de ese máximo.
Buscar máximo: recibe por parámetro el nodo por donde nosotros queremos empezar a buscar y ademas le pasamos un bool para que nos ayude a saber si encontramos o no el máximo. Primero inicializamos un nodo temporal que utilizamos para devolver y declaramos la variable de encontrada como falsa. Ahora tenemos el caso de que no haya nada del lado derecho, donde si es así encontramos el máximo y lo devolvemos. Nuestro retorno sabemos que va a estar si o si del lado derecho porque siempre está el más grande. Si encontramos el máximo seteamos el nuevo nodo derecho como nulo y devolvemos ese nodo máximo y lo ponemos en el nodo original del método remover para hacer el intercambio, entonces donde estamos parados

queremos que lo de la derecha sea nulo porque no hay nada más grande que el valor que encontramos y ponemos la bandera en false para frenar los ciclos recursivos. Por último devolvemos el máximo.
Ahora en el método remove, para hacer el intercambio, declaramos el booleano, y decimos que el máximo que vamos a guardar en la variable temporal va a ser el máximo desde el lado izquierdo, ahora para hacer el intercambio, a la derecha de la variable auxiliar vamos a poner lo que estaba a la derecha del nodo que eliminamos y a la izquierda de la variable auxiliar vamos a almacenar lo que estaba a la izquierda del nodo que acabamos de eliminar.
Caso contrario almacenamos temporalmente la parte izquierda, y luego obtenemos esa parte izquierda y le seteamos su parte derecha y a esa parte derecha le asignamos lo que antes teníamos en esa parte derecha.
Para terminar, eliminamos el nodo y devolvemos el auxiliar.

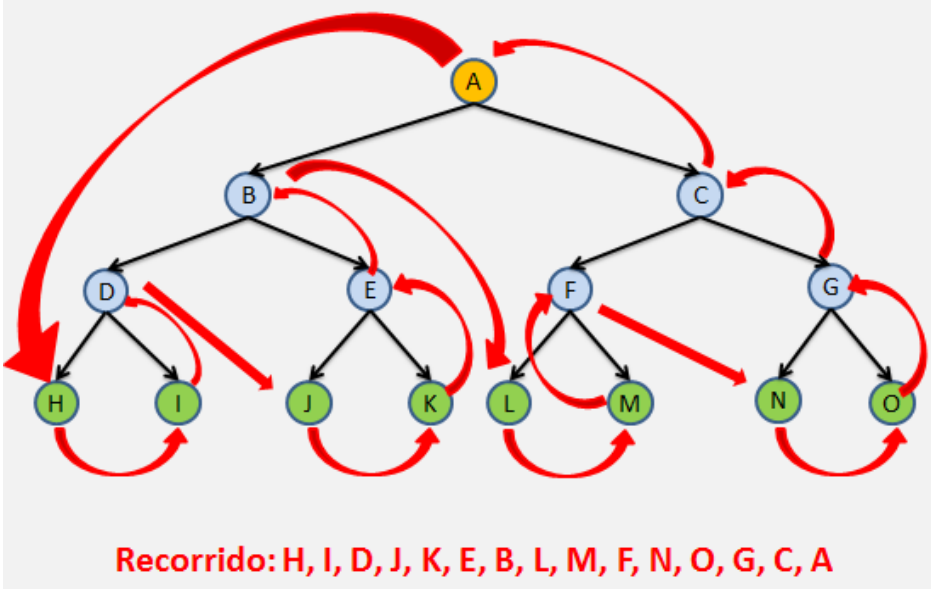
- En el caso de que no estemos parados en el nodo que queremos remover. Tenemos que si el valor del nodo en el que estamos es mayor al que queremos borrar entonces tenemos que ir del lado izquierdo para remover, donde modificamos el nodo izquierdo y de forma recursiva llamamos a remove mandándole el dato y toda la rama izquierda donde estamos parados.
Hacemos lo mismo si el valor del nodo donde estamos parados es menor que el que buscamos para eliminar.

- preorder(): el método publico va a llamar al método privado y manda por parámetro la raíz. Tenemos los métodos privados para poder pasar por parámetro la raíz y la raíz no es accesible por afuera de la clase por lo cual por eso tenemos este estilo de sobrecargas.
  - En la sobrecarga privada tenemos por un lado que si no hay nada en el árbol no significa que tenga que dar error sino que simplemente va a retornar nada.
  - En el caso de que tengamos elementos en el árbol, primero vamos a imprimir la raíz, luego llamamos de forma recursiva a la función pero llamando el nodo izquierdo y el nodo derecho. A modo de entender tenemos el siguiente grafico:



- inorder(): en la sobrecarga privada acepta por parámetro un nodo. Este es parecido al preorden nada más que la impresión va a ser primero se imprime lo que está a la izquierda, luego se imprime la raíz, y por último la parte derecha, permitiendo tener una impresión ordenada de menor a mayor.

- postorder(): en este caso primero imprimimos el nodo izquierdo, luego la derecha y por último la raíz.



- ~ArbolBinario(): destructor.

- esVacio(): devolvemos si el árbol está vacío o no. Devolvemos un booleano para el caso en el que la raíz sea nula o no.