Interfaz de usuario gráfica

Descripción generada automáticamente con confianza mediaDibujo animado de un personaje de caricatura

Descripción generada automáticamente con confianza media

**Mini-proyecto N°1**

**Nombres:**

Vicente Ignacio Ríos Adasme, 2020449783

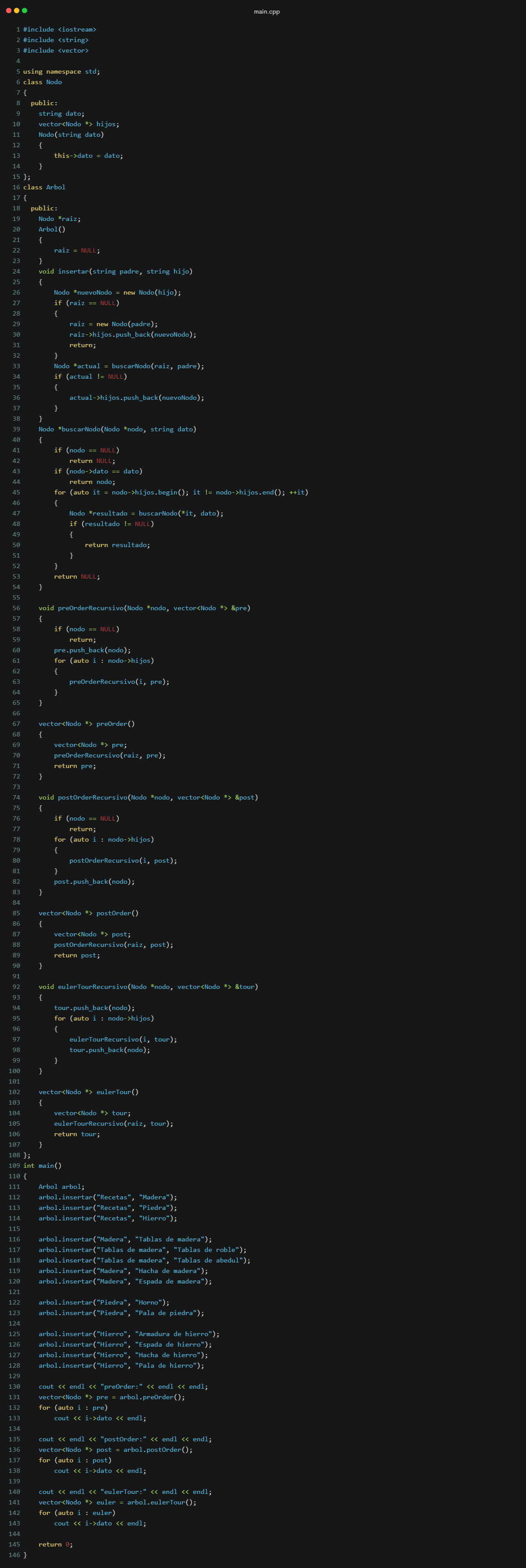
Pablo Alonso Sanhueza Yévenes, 2021439005

Lunes, 08 de mayo de 2023, Concepción

Introducción

Con el archivo main.cpp que se proporciona en las instrucciones del laboratorio, se pide implementar el recorrido *pre-order* para el árbol en cuestión para el método *preOrder().* Para esto, se crea un vector de punteros de objetos Nodo llamado “pre”, y luego se llama al método preOrderRecursivo(), este método vendría a ser el método usual de *pre-order*, donde recibe un nodo a visitar, se visita, y luego se visitan sus hijos de manera recursiva ingresando sus hijos como argumento de la misma función *preOrderRecursivo*. De esta forma, también se incluye el vector de punteros de objetos Nodo “pre”, de tal manera que para marcar un nodo como visitado, este se ingresa en el vector. Una vez se acaben todas las llamadas recursivas y se “regrese” a *preOrder(),* el método retorna el vector “pre”, luego estos valores se imprimen por pantalla. A continuación, se aprecia su implementación e impresión en pantalla.

**Script**: main.cpp



Texto

Descripción generada automáticamente

Resumen

*ListArr* consiste en un híbrido entre arreglos y listas, además se hace uso de un árbol binario. *ListArr* es una estructura de datos que almacena números enteros, que debe cumplir con poder ingresar datos tanto al principio como por el final (izquierda y derecha), además de ingresar datos en el i-ésimo índice, buscar un dato especifico y retornar si existe o no en la estructura, también debe cumplir con poder imprimir todos los datos ingresados en la estructura.

La implementación realizada para el desarrollo de este proyecto consiste en la siguiente: Se tienen nodos con datos llamados *DataNodes* que posee un arreglo de enteros y un puntero a *DataNodes* que le sigue inmediatamente a la derecha. También se tienen nodos resumen llamados *SummaryNode* que almacenan la cantidad de datos ingresados y la capacidad de datos a partir de los nodos descendientes de cada *SummaryNode*. Estos *SummaryNodes* forman un árbol binario sobre los *DataNodes*, donde las hojas de este árbol apuntan a un único *DataNode*.

insert\_left():

Para ingresar elementos por la izquierda, se desplazan todos los elementos del primer *DataNode* a la derecha, y en la posición 0 se ingresa el dato en cuestión. Si el primer *DataNode* está lleno antes de ingresar el dato, se mueven todos los elementos del nodo siguiente a la derecha, y en la posición 0 de tal nodo se ingresa el último elemento del primero. Análogamente, en el primer nodo se mueve todo a la derecha, eliminando el último valor, pues se pasa a otro nodo, y en la posición 0 se ingresa el valor nuevo. Si el primer nodo está lleno, y el nodo qué le sigue también, o bien, no existe otro nodo, se crea un *DataNode* nuevo inmediatamente después de él que se conecta con el resto de la estructura. En este *DataNode* nuevo se ingresa el último elemento del primer nodo, y en el primero nodo todo se desplaza a la derecha, y en la posición 0 se ingresa el nuevo valor.

insert\_right():

Al momento de ingresar por la derecha, se llega hasta el último *DataNode* de la derecha, navegando por medio del árbol binario, una vez en dicho *DataNode*, si tiene espacio disponible, se ingresa en la primera posición libre del nodo, en caso contrario, se crea un *DataNode* nuevo inmediatamente a la derecha y ahí, en el índice 0 se ingresa el nuevo valor.

insert():

Si se inserta en una posición que no es ni la primera ni la última, se llega hasta el i-ésimo índice de la estructura total (iniciando desde el índice 0) navegando por medio del árbol, ya en el DataNode correspondiente al del i-ésimo índice, si tiene espacio, se mueven los elementos hacía la derecha a partir del índice i, y se ingresa el dato en la posición i. Si está lleno y el nodo que le sigue también está lleno, o bien, no existe, se crea un DataNode a su derecha, y se ingresa el último elemento del nodo anterior al nuevo, de mueven hacia la derecha los elementos del nodo a partir del índice i. Si el nodo está lleno, y el que le sigue tiene espacio disponible, se mueven hacia la derecha los elementos del nodo siguiente y en la posición 0 se ingresa el último elemento del nodo anterior al nuevo, y en el nodo original se mueven los elementos hacia la derecha a partir del índice i, y se ingresa el valor nuevo en la posición i.

Actualización árbol binario:

Cada *SummaryNode* tiene un *DataNode* asociado, solamente las hojas del árbol tienen un *DataNode* asociado no nulo. Cuando se debe agregar un *DataNode* a la estructura, independiente del método de inserción utilizado, cada vez que se tiene un nuevo *DataNode*, o bien, cada vez que se alteran los valores almacenados en los *DataNodes* se actualiza el árbol binario por completo. Para esto, primero se tiene el método *“freeBinaryTree()”* que elimina todo el árbol binario, para liberar memoria, luego se tiene *“cleanAllParents()”* para eliminar toda relación entre los *SummaryNodes* con los *DataNodes*, luego se llama a *“createBinaryTree()”* para crear todos los nodos necesarios en el árbol binario a partir de una cantidad de nodos deseados dada, este es un método recursivo, comenzando desde la raíz root, hasta las hojas, caso base de la función recursiva. Posteriormente se llama a “assignDataNodes()”. Método que asigna la i-ésima hoja del árbol, con el i-ésimo *dataNode*, en base a esto, por medio de *“updateTree()”,* cada *SummaryNode* del árbol actualiza su valor de datos ingresados y su capacidad total.

Implementación de ListArr

Se procede a realizar una implementación de ListArr para los siguientes métodos definidos en ListArrADT.

**Script**: ListArr.h

Texto

Descripción generada automáticamente

Se tiene que, para almacenar la información ingresada en la estructura, se hace uso de un híbrido entre arreglos y listas enlazadas. En este caso, se implementa la siguiente clase dentro de ListArr, llamada DataNode(), donde cada DataNode tiene un contenedor que es un arreglo de enteros cuya capacidad es inicializada en su constructor, *(sea este un valor b mencionado más adelante)* y almacenada en su atributo nCapacity, a su vez, se tiene el atributo “count” que representa la cantidad de datos ingresados en su contenedor, por lo tanto es inicializado en cero. Además, se hace uso de una lista enlazada simple, por lo tanto, DataNode tiene el atributo de un puntero a DataNode llamado “next”, donde este vendría a ser el nodo que le sigue inmediatamente a la derecha, de esta forma, el constructor de DataNode inicializa next apuntando al nulo. Por otro lado, se implementa el método isFull() que retorna si el arreglo del DataNode está lleno o no, true o false, respectivamente, donde será lleno solamente si count y nCapacity contienen el mismo valor.

Dado que, de los objetivos de la representación ListArr es mejorar el acceso directo a sus elementos, cosa que no es del todo eficiente una lista enlazada, se hace uso de nodos resumen por cada par consecutivos de nodos de ListArr, donde cada nodo resumen almacena la capacidad y cantidad de datos totales de ambos nodos.

Diagrama

Descripción generada automáticamente

Se implementa la clase *SummaryNode* que posee el atributo *“quantity”* y *“sCapacity”,* la capacidad y cantidad antes mencionada, respectivamente. Por otro lado, dado que, al tener más de dos *DataNodes*, se necesita más de un *SummaryNode*, donde, por cada par de *DataNodes* consecutivos se tiene un *SummaryNode*, se tiene un árbol binario donde cada *SummaryNode* corresponde a un nodo del árbol. De esta forma, cada *SummaryNode*, además contiene los atributos correspondientes a punteros a *SummaryNode* para sus hijos izquierdo ni derecho, llamados *“left”* y *“right”*, respectivamente, esto dado que se puede tener un árbol que es un grafo no necesariamente trivial.

Además, se tiene que cada *SummaryNode* está relacionado con solamente un único *DataNode*, un atributo llamado data. De esta forma, se tiene que, en la imagen anterior, cada representación de la cantidad y capacidad de datos de un *DataNode*, es en realidad un *SummaryNode* donde su data apunta a cada *DataNode*, siendo estos, hijos del *SummaryNode* que contiene la cantidad y capacidad de los *SummaryNodes* hijos. Finalmente, solamente las hojas del árbol de *SummaryNode* tienen un data que no apunta al nulo. Por lo tanto, en su constructor, la cantidad y capacidad se inicializa en cero, mientras que sus hijos y su data apuntan al nulo.

Para ***ListArr*** heredado de *ListArrADT*, se presenta la definición de la clase, posteriormente, se describe los métodos utilizados.

Donde en el constructor se pide un valor *b* ingresado en el atributo *capacity*, mencionado anteriormente, siendo este la capacidad de cada arreglo dentro de los *DataNode*. Al crear un *ListArr*, automáticamente se crea un *SummaryNode* con un *“data”* (de capacidad *b*, o *capacity*) que no contiene elementos, por lo que el único *SummaryNode* sería la raíz *“root”.* De esta forma se llama al método *“setLeafs()”,* método setter del atributo *“leafs”,* atributo que representa la cantidad de hojas, del árbol binario. Además, la variable DataAssigned es false por medio de la llamada a su setter *“setDataAssigned()”,* dicha variable es útil para realizar la asignación de los *DataNodes* a los *SummaryNode*. La raíz root creada automáticamente es asignada por medio del método *“createBinaryTree()”,* método que crea el árbol binario a partir una cantidad de hojas dada, donde el único data que existe y está relacionado con el *SummaryNode* *root* también es *“head”,* es decir, el primer nodo de la lista enlazada de *DataNodes*. Luego se actualiza el árbol.

Para el destructor, primero se libera la memoria utilizada por el árbol binario que existe en el momento mediante el método *“freeBinaryTree()”,* comenzando desde *root*. Luego, se recorre la lista enlazada simple, eliminando cada contenedor de enteros, y también, dicho *DataNode*.

Para liberar memoria, *freeBinaryTree()* recibe un *SummaryNode* que es raíz del árbol binario general, o bien, un subárbol de este. De manera recursiva, se eliminan todos los nodos hijos antes de que luego se elimine el nodo ingresado al método. Esto se logra, primero verificando si el *SummaryNode* recibido es nulo, si es así la función retorna, de lo contrario, llama recursivamente a *freeBinaryTree()* tanto en su hijo izquierdo como el derecho, luego se elimina el nodo actual, asegurando así que se liberen todos los nodos descendientes antes de liberar al nodo padre

El árbol binario es creado por medio de *“createBinaryTree()”* que recibe un número entero leaf, que representa la cantidad de hojas totales que se desea que tenga el árbol, y retorna un puntero a la raíz del árbol binario creado.

Primero se verifica si *leaf* es 1, de ser así, se crea un *SummaryNode* que almacena el primer DataNode head en su data y lo devuelve como raíz del árbol binario. Este es el caso base de la recursión, y se obtiene cuando se ha llegado a una hoja del árbol binario. De lo contrario, si *leaf* es mayor a 1, se crea un *SummaryNode “newNode”,* donde sus hijos izquierdo y derecho son establecidos llamando recursivamente a *createBinaryTree*(), con *“leaf / 2”* y *“leaf – leaf / 2”,* respectivamente. Esto crea subárboles izquierdo y derecho de igual tamaño, siempre y cuando *leaf* sea un número par, de lo contrario, el subárbol izquierdo tendrá un nodo más que el subárbol derecho.

Luego, se llama a las funcionas *updateQuantity()* y *updateCapacity(),* para actualizar los campos *“quantity”* y *“capacity”* de *newNode* a partir de los valores de los nodos hijos izquierdo y derecho, por último se devuelve *newNode* como la raíz del árbol binario creado.

El actualizar los campos *quantity* y *sCapacity* de un nodo resumen *SummaryNode* dado se hace uso de *updateQuantity()* y *updateCapacity(),* recibiendo dicho nodo resumen por argumento. El método *updateQuantity()* inicializa tres variables enteras *“leftQuantity”,* *“rightQuantity”* y *“dataQuantity”,* que representa la cantidad de elementos en el subárbol izquierdo y derecho y el elemento representado en el *DataNode* actual si es que existe, respectivamente.

Luego, se verifica si dicho nodo tiene un hijo izquierdo, si es así, se llama recursivamente a *updateQuantity()* para ese hijo y almacena el resultado en *leftQuantity,* de manera similar ocurre si aquel nodo tiene un hijo derecho, guardando la cantidad en *rightQuantity*, y también si es que el nodo tiene un *DataNode* asociado, almacenando el valor en *“dataQuantity”.* Finalmente, la cantidad total de elementos ingresados en el subárbol representado a partir de un nodo actual se calcula sumando *leftQuantity*, *rightQuantity* y *dataQuantity*, y se almacena el total en el atributo *quantity* del *SummaryNode* en cuestión.

De manera análoga se calcula la capacidad, guardando el total de la capacidad del subárbol representado por el *SummaryNode* actual. El proceso es idéntico, pero los elementos se almacenan en *nCapacity* de cada nodo, donde el resultado final es almacenado en *sCapacity*.

Si se desea actualizar todo el árbol, por completo, se debe llamar a *updateQuantity* y *updateCapacity* ingresando como argumento a la raíz del árbol. Para simplificar esta tarea se tiene el método *“updateTree()”* que llama a los métodos de actualización de cantidad y capacidad ingresando la raíz *root* del árbol binario de resumen, asegurando así que toda la información de resumen esté actualizada.

Para obtener la cantidad total de elementos almacenados en ListArr, se llama a *“size()”,* que retorna la cantidad de datos ingresados en la raíz del árbol binario.

Para insertar por la izquierda, se implementa mediante la idea mencionada en el *resumen* se debe llamar a *“insertBefore()”* proporcionando *head*, el valor *v* a ingresar, y el entero 0, que representa el ingresar el valor *v* en la cabeza en su índice 0.

*insertBefore()* respeta la estructura descrita en el resumen, pero a partir del índice i, primero mueve elementos hacia la derecha y luego inserta un elemento.

Si se desea insertar un elemento en una posición diferente al inicio o al final, se llama a *“insert()”* que recibe el valor *v* y el índice *i* en donde ingresar *v*. La búsqueda del elemento en la posición i se describe a continuación:

Si *i* es mayor a la capacidad de datos que posee la *root*, entonces el dato no se encuentra y se lanza excepción por índice invalido.

Si root tiene un data no nulo, entonces *root* es una hoja, por lo que se inserta el dato en el índice i del data asociado a *root*.

Si i es menor a la cantidad de datos ingresados del hijo izquierdo de *root* entonces se llama a un método auxiliar llamado *“insertRecursivo()”,* este recibe el valor v y el índice i, además del hijo izquierdo de *root*. En caso contrario, se procede a llamar a *insertRecursivo()* con el hijo derecho de la *root* con el índice i menos la cantidad de datos ingresados en el hijo izquierdo de *root*.

Para *insertRecursivo(),* se verifica si el nodo actual no es hoja, y, al igual de lo ocurrido anteriormente, si i es menor a la cantidad de datos ingresados del hijo izquierdo del nodo actual se llama a *insertRecursivo()* con el hijo izquierdo del nodo actual, en caso contrario se llama a *insertRecursivo()* con el hijo derecho del nodo actual y un índice *i* disminuido en la cantidad de datos ingresados en el hijo izquierdo del nodo actual. Note que se debe disminuir la cantidad de *i* debido a que se considera que el índice a buscar se encuentra después de los primeros *n* índices, donde n es la cantidad de datos ingresados en el hijo izquierdo entre las llamadas recursivas del nodo actual. Si el nodo actual es una hoja significa que ya se puede acceder a su data, y ahí se ingresa el elemento mediante *insertBefore().*

Análisis teórico

Para el recorrido de Euler Tour se tiene el vector de punteros de nodos “tour” donde se marca la visita de los nodos. Se tiene, además, el método *eulerTourRecursivo()* para visitar de manera correspondiente estos nodos: Al inicio de dicho método se marca como visitado el nodo, luego, por cada hijo de aquel nodo se llama al recorrido de Euler. Cada vez que termine la llamada recursiva, nuevamente se tiene que marcar la visita del mismo nodo, de esta forma se tiene que al visitar todos los nodos descendientes de un nodo raíz de los subárboles del árbol original se vuelve a marcar la visita de dicha raíz. En palabras simples, se tiene que, dentro del vector, todos los elementos que están “encerrados” por un término diferente, entonces todos esos elementos encerrados son descendientes de aquel término diferente (entiéndanse “*término*” como variable/objeto almacenado dentro del vector).

**Script**: main.cpp

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Descripción relevante para análisis teórico

Finalmente, al compilar y ejecutar main.cpp se tiene la siguiente salida.

Análisis experimental

Se procede a realizar un análisis experimental para los métodos de *ListArrADT* en las estructuras *Array, List y ListArr*. En este estudio experimental se considerarán los métodos *insert\_left(), insert\_right() y find().* Para esto, se probarán con ***b*** igual a 128, 512, 1024, 2048 y con un ***n*** igual a 10000, 20000, 40000, 80000. El formato de entrega de resultados es el siguiente: Primero se muestra la cantidad b y n correspondientes a la prueba, luego, el tiempo total de *n* iteraciones de *insert\_left()* en milisegundos, luego le sigue el tiempo promedio a las n iteraciones. Después de esto, se tiene el mismo formato para mostrar el resultado de *insert\_right()* e *insert(),* respectivamente, luego comienza otra prueba con *b* o *n* diferentes.

A continuación, output del experimento correspondiente a: **Array**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteOutput del experimento correspondiente a: **List**

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Output del experimento correspondiente a: **ListArr**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Conclusiones