Mini-proyecto N°1

Nombres:

Vicente Ignacio Ríos Adasme, 2020449783

Pablo Alonso Sanhueza Yévenes, 2021439005

Lunes, 08 de mayo de 2023, Concepción

Introducción

En este proyecto, se implementará un árbol binario en C++ utilizando arreglos y listas. Este estará diseñado para expandirse dinámicamente a medida que se inserten nuevos nodos conforme a la necesidad del usuario. Además de poder auto reestructurar el árbol binario asociado a la estructura que almacena los datos. El árbol nace de la mejora de los puntos débiles de las listas, esto es el acceso directo a sus elementos, pues este constará de nodos que resumen la información de los datos almacenados, esto se resume en cantidad de datos ingresados y capacidad de los arreglos de los nodos del subárbol descendiente, finalmente las hojas de este árbol apuntan a un nodo con datos ingresados, los cuales son enlazados con punteros, cada nodo con datos ingresados tendrá un tamaño fijo. Cuando un nodo esté lleno, se creará un nuevo nodo que se enlazará con el nodo lleno, y se reestructurará el árbol para acomodar el nuevo nodo y así facilitar su búsqueda.

Con esto, el árbol podrá crecer de manera dinámica a medida que se inserten nuevos nodos, y se mantendrá balanceado gracias a la estrategia de reestructuración implementada en el cual se comenta más adelante durante en el presente informe.

Se pretende comparar esta implementación con lo anteriormente vistos en clases como lo son los arreglos y listas enlazadas.

Los arreglos: estructura de datos que almacena una colección de elementos del mismo tipo en una secuencia contigua de memoria se acceden mediante un índice numérico que indica su posición en el arreglo.

Las principales ventajas de los arreglos son su eficiencia en el acceso a los elementos y su capacidad para realizar operaciones matemáticas eficientes en los datos almacenados

Las desventajas son su longitud fija e ineficiencia al momento de insertar elemento en una posición ya ocupada tardando un tiempo teórico o(n).

lista enlazada: es una estructura de datos en la que cada elemento (nodo) contiene un valor y un puntero que apunta al siguiente elemento de la lista.

Las principales ventajas de las listas enlazadas son su capacidad para manejar colecciones de datos sin requerir una longitud fija, y su capacidad para realizar operaciones de inserción y eliminación de elementos de manera eficiente

Sin embargo, sus desventajas son el acceso a los elementos ya que una lista enlazada no es tan eficiente como en un arreglo, porque requiere recorrer los nodos de la lista hasta llegar al elemento deseado, esto se hace en buscando direcciones de los espacios de memoria lo que lo hace más lento.

En teoría deberíamos conseguir arreglos enlazados como listas siendo una estructura de datos que combina las ventajas de ambos enfoques. Cada nodo contendrá un arreglo que almacenará varios elementos y cada elemento del arreglo estará enlazado a través de punteros con otros elementos de la lista. Esto permitirá una manipulación eficiente de colecciones de datos dinámicas, al mismo tiempo que proporciona un acceso eficiente a los elementos individuales del arreglo.

La finalidad de nuestro trabajo es utilizar lo mejor de ambos, así generaremos un análisis teórico y experimental comparando su eficiencia y comparándolos para ver cuál es más eficiente en cada caso.

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.

Todo código creado está presente en el siguiente repositorio, o bien, en el documento adjunto a esta entrega:

https://github.com/PabloSanhueza1/ED-Miniproyecto-1

Universidad de Concepción

Implementación de ListArr

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

Script: ListArrADT.h

ListArrADT.h

```h
#ifndef LISTARRADT_H
#define LISTARRADT_H

#include <bits/stdc++.h>
using namespace std;

class ListArrADT
{
  public:
    virtual int size() = 0;                // Retorna la cantidad de elementos almacenados en el ListArr
    virtual void insert_left(int v) = 0;   // Inserta un nuevo valor v a la izquierda del ListArr
    virtual void insert_right(int v) = 0;  // Inserta un nuevo valor v a la derecha del ListArr
    virtual void insert(int v, int i) = 0; // Inserta un nuevo valor v en el índice i del ListArr
    virtual void print() = 0;              // Imprime por pantalla todos los valores almacenados en el ListArr
    virtual bool find(int v) = 0;          // Busca en el ListArr si el valor v se encuentra almacenado
};

#endif
```

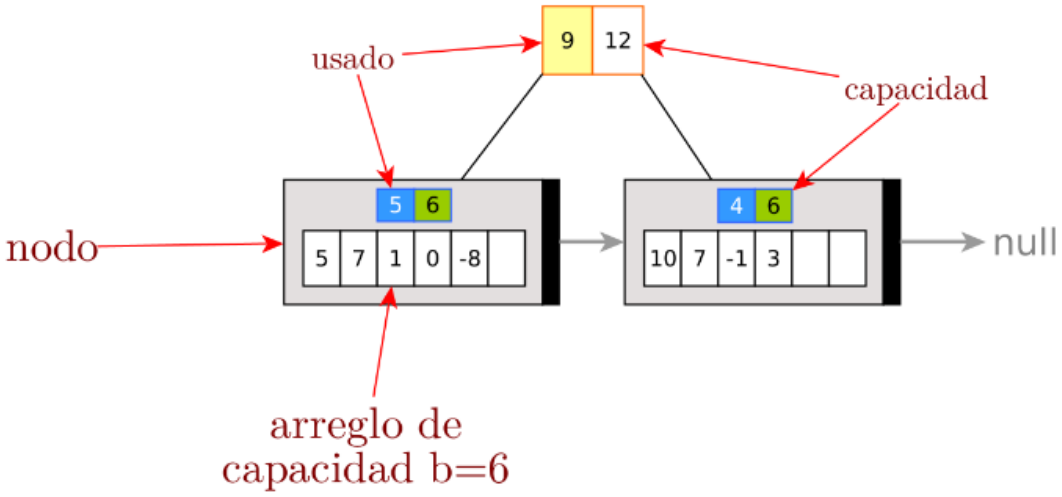
Se tiene que, para almacenar la información ingresada en la estructura, se hace uso de un híbrido entre arreglos y listas enlazadas. En este caso, se implementa la siguiente clase dentro de ListArr, llamada DataNode(), donde cada DataNode tiene un contenedor que es un arreglo de enteros cuya capacidad 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.

Script: ListArr.h

ListArr.h

```
#include "ListArrADT.h"

#include <bits/stdc++.h>
using namespace std;

class DataNode
{
  public:
    int *container;
    int count;
    int nCapacity;

    DataNode *next;

    DataNode(int nCapacity)
    {
        this->nCapacity = nCapacity;
        container = new int[nCapacity];
        count = 0;
        next = NULL;
    }

    bool isFull()
    {
        return count == nCapacity;
    }
};
```

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.



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.

Script: ListArr.h

```
29 class SummaryNode
30 {
31   public:
32     int quantity;
33     int sCapacity;
34 
35     SummaryNode *left;
36     SummaryNode *right;
37 
38     DataNode *data;
39 
40     SummaryNode()
41     {
42         quantity = 0;
43         sCapacity = 0;
44         left = NULL;
45         right = NULL;
46         data = NULL;
47     }
48 };
```

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.

Script: ListArr.h

```
50 class ListArr : public ListArrADT
51 {
52   private:
53     SummaryNode *root;
54     DataNode *head;
55
56     int capacity;
57     int leafs;
58     bool dataAssigned;
59
60   public:
61     ListArr(int b);
62     ~ListArr();
63
64     // Seters y getters
65     void setDataAssigned(bool v); // Setter de variable auxiliar dataAssigned, util para assignDataNodes
66     bool getDataAssigned();       // Getter de variable auxiliar dataAssigned, util para assignDataNodes
67     void setLeafs(int v);         // Setter de variable auxiliar leafs, hojas del arbol binario
68     int getLeafs();               // Getter de variable auxiliar leafs, hojas del arbol binario
69
70     // Métodos para actualizar arbol binario
71     void freeBinaryTree(SummaryNode *node);           // Elimina el arbol binario (libera memoria)
72     DataNode *getNode(int index);                     // Retorna el i-ésimo DataNode
73     void assignDataNodes(SummaryNode *root, int leaf); // Asigna el i-ésimo DataNode a la i-ésima hoja del arbol binario
74     void cleanAllParents(SummaryNode *node); // Elimina toda relacion entre las hojas del arbol con los DataNodes
75     int updateQuantity(SummaryNode *node);   // Actualiza la cantidad de elementos ingresados en los nodos del arbol
76     int updateCapacity(SummaryNode *node);   // Actualiza la capacidad  en los nodos del arbol
77     void updateTree();                       // Actualiza la cantidad y capacidad de los nodos del arbol
78     SummaryNode *createBinaryTree(int leaf); // Crea un arbol binario
79
80     // Métodos de inserción
81     void insertNode(DataNode *&dataNode); // Crea un DataNode inmediatamente después del nodo recibido en el argumento
82     void insertBefore(DataNode *dataNode, int v, int i);        // Inserta a la izquierda de un dataNode
83     void insertAfter(DataNode *dataNode, int v, int i);         // Inserta a la derecha de un dataNode
84     void insertRecursivo(SummaryNode *actualNode, int v, int i); // Método auxiliar para insert() a implementar
85
86     // Métodos a implementar:
87     int size();               // Retorna la cantidad de elementos almacenados en el ListArr
88     void insert_left(int v);  // Inserta un nuevo valor v a la izquierda del ListArr
89     void insert_right(int v); // Inserta un nuevo valor v a la derecha del ListArr
90     void insert(int v, int i); // Inserta un nuevo valor v en el índice i del ListArr
91     void print();             // Imprime por pantalla todos los valores almacenados en el ListArr
92     bool find(int v);         // Busca en el ListArr si el valor v se encuentra almacenado
93 };
```

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.

Script: ListArr.cpp

ListArr.cpp

```cpp
#include "ListArr.h"

// Mueve los elementos hacia la derecha de un arreglo a partir del j-ésimo elemento
void moveRight(DataNode *&dataNode, int j)
{
    for (int i = dataNode->count - 1; i >= j; i--)
    {
        dataNode->container[i + 1] = dataNode->container[i];
    }
}

ListArr::ListArr(int b)
{
    capacity = b;
    setLeafs(1);
    setDataAssigned(false);
    DataNode *initialNode = new DataNode(capacity);
    head = initialNode;
    root = createBinaryTree(getLeafs());
    updateTree();
}
```

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*.

Script: ListArr.cpp

```
23 ListArr::~ListArr()
24 {
25     freeBinaryTree(root);
26     DataNode *temp = head;
27     while (temp != NULL)
28     {
29         DataNode *next = temp->next;
30         delete[] temp->container;
31         delete temp;
32         temp = next;
33     }
34 }
```

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.

Script: ListArr.cpp

```
58 void ListArr::freeBinaryTree(SummaryNode *node)
59 {
60     if (node == nullptr)
61     {
62         return;
63     }
64     freeBinaryTree(node->left);
65     freeBinaryTree(node->right);
66     delete node;
67 }
```

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.

Script: ListArr.cpp

```
175 SummaryNode *ListArr::createBinaryTree(int leaf)
176 {
177     if (leaf == 1)
178     {
179         SummaryNode *summaryNode = new SummaryNode();
180         summaryNode->data = head;
181         return summaryNode;
182     }
183 
184     SummaryNode *newNode = new SummaryNode();
185 
186     // crear sub-árboles izquierdo y derecho recursivamente
187     newNode->left = createBinaryTree(leaf / 2);
188     newNode->right = createBinaryTree(leaf - leaf / 2);
189 
190     updateQuantity(newNode);
191     updateCapacity(newNode);
192 
193     return newNode;
194 }
```

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.

Script: ListArr.cpp

```
116 int ListArr::updateQuantity(SummaryNode *node)
117 {
118     int leftQuantity = 0;
119     int rightQuantity = 0;
120     int dataQuantity = 0;
121 
122     if (node->left != NULL)
123     {
124         leftQuantity = updateQuantity(node->left);
125     }
126 
127     if (node->right != NULL)
128     {
129         rightQuantity = updateQuantity(node->right);
130     }
131 
132     if (node->data != NULL)
133     {
134         dataQuantity = node->data->count;
135     }
136 
137     node->quantity = leftQuantity + rightQuantity + dataQuantity;
138 
139     return node->quantity;
140 }
```

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.

Script: ListArr.cpp

```
142 int ListArr::updateCapacity(SummaryNode *node)
143 {
144     int leftCapacity = 0;
145     int rightCapacity = 0;
146     int dataCapacity = 0;
147 
148     if (node->left != NULL)
149     {
150         leftCapacity = updateCapacity(node->left);
151     }
152 
153     if (node->right != NULL)
154     {
155         rightCapacity = updateCapacity(node->right);
156     }
157 
158     if (node->data != NULL)
159     {
160         dataCapacity = node->data->nCapacity;
161     }
162 
163     node->sCapacity = leftCapacity + rightCapacity + dataCapacity;
164 
165     return node->sCapacity;
166 }
```

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.

```
305 int ListArr::size()
306 {
307     return root->quantity;
308 }
```

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.

Script: ListArr.cpp

```cpp
void ListArr::insertBefore(DataNode *dataNode, int v, int i)
{
    if (dataNode->count == dataNode->nCapacity)
    {
        if (dataNode->next == NULL || dataNode->next->isFull())
        {
            setLeafs(getLeafs() + 1);
            insertNode(dataNode);

            dataNode->next->container[0] = dataNode->container[dataNode->count - 1];
            dataNode->next->count = dataNode->next->count + 1;

            moveRight(dataNode, i);
            dataNode->container[i] = v;

            freeBinaryTree(root);
            root = createBinaryTree(getLeafs());
            cleanAllParents(root);
            for (int j = 1; j <= getLeafs(); j++)
            {
                setDataAssigned(false);
                assignDataNodes(root, j);
            }
            updateTree();
        }
        else
        {
            moveRight(dataNode->next, 0);
            dataNode->next->container[0] = dataNode->container[dataNode->count - 1];
            dataNode->next->count = dataNode->next->count + 1;

            moveRight(dataNode, i);
            dataNode->container[i] = v;

            freeBinaryTree(root);
            root = createBinaryTree(getLeafs());
            cleanAllParents(root);
            for (int j = 1; j <= getLeafs(); j++)
            {
                setDataAssigned(false);
                assignDataNodes(root, j);
            }
            updateTree();
        }
    }
    else
    {
        moveRight(dataNode, i);
        dataNode->container[i] = v;
        dataNode->count = dataNode->count + 1;
        updateTree();
    }
}
```

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.*

Script: ListArr.cpp

```
329 void ListArr::insert(int v, int i)
330 {
331     try
332     {
333         if (i > root->sCapacity)
334             throw "Invalid index!";
335         if (root->data != NULL)
336             insertBefore(root->data, v, i);
337         else if (i < root->left->quantity)
338             insertRecursivo(root->left, v, i);
339         else
340             insertRecursivo(root->right, v, i - root->left->quantity);
341     }
342     catch (const char *message)
343     {
344         cerr << message << endl;
345         exit(EXIT_FAILURE);
346     }
347 }
```

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().*

Script: ListArr.cpp

```
291 void ListArr::insertRecursivo(SummaryNode *actualNode, int v, int i)
292 {
293     if (actualNode->data == NULL)
294     {
295         if (i < actualNode->left->quantity)
296             insertRecursivo(actualNode->left, v, i);
297         else
298             insertRecursivo(actualNode->right, v, i - actualNode->left->quantity);
299     }
300     else
301         insertBefore(actualNode->data, v, i);
302 }
```

Si se desea insertar en el índice 0 se debe llamar a "*insert_left()*" que ingresa el valor dado en el argumento al inicio de *ListArr*, respetando la estructura mencionada en el resumen y la estructura de *insertBefore()*, pues *insert_left()* se implementa de la siguiente manera.

Script: ListArr.cpp

```
310 void ListArr::insert_left(int v)
311 {
312     insertBefore(head, v, 0);
313 }
```

Para insertar en la última posición de *ListArr* se llama a *"insert_right()"*, donde se llega al último *DataNode*, y se pasa como argumento del método *"insertAfter()"*. Este último método realiza la misma tarea que *insertBefore()* pero con algunas diferencias.

Script: ListArr.cpp

```
315 void ListArr::insert_right(int v)
316 {
317     SummaryNode *actualNode = root;
318 
319     // Llega al último DataNode de la derecha
320     while (actualNode->right != NULL)
321         actualNode = actualNode->right;
322 
323     DataNode *dataNode = actualNode->data;
324     int pos = dataNode->count;
325 
326     insertAfter(dataNode, v, pos);
327 }
```

Script: ListArr.cpp

```
262 void ListArr::insertAfter(DataNode *dataNode, int v, int i)
263 {
264     if (dataNode->count == dataNode->nCapacity)
265     {
266         setLeafs(getLeafs() + 1);
267         insertNode(dataNode);
268 
269         dataNode->next->container[0] = v;
270         dataNode->next->count = dataNode->next->count + 1;
271 
272         freeBinaryTree(root);
273         root = createBinaryTree(getLeafs());
274         cleanAllParents(root);
275         for (int j = 1; j <= getLeafs(); j++)
276         {
277             setDataAssigned(false);
278             assignDataNodes(root, j);
279         }
280         updateTree();
281     }
282     else
283     {
284         dataNode->container[i] = v;
285         dataNode->count = dataNode->count + 1;
286         updateTree();
287     }
288 }
```

La principal diferencia entre *insertBefore()* y *insertAfter()* es que el método *insertBefore()* trata de mantener el balance del árbol al mover elementos y creando nuevos nodos si es necesario, mientras que el método *insertAfter()* siempre crea un nuevo nodo para insertar el nuevo elemento.

Para imprimir los datos almacenados en la estructura *ListArr*, se recorre cada *DataNode*, imprimiendo el número correspondiente al *i-ésimo* nodo, y a su vez, a lo mucho b números contenidos en dicho *DataNode*. A continuación, su implementación.

Script: ListArr.cpp

```
349 void ListArr::print()
350 {
351     DataNode *dataNode = head;
352 
353     cout << endl;
354     int i = 0;
355 
356     while (dataNode != NULL)
357     {
358         cout << "Node " << i << ":" << endl;
359         for (int j = 0; j < dataNode->count; j++)
360             cout << dataNode->container[j] << " ";
361 
362         cout << endl << endl;
363         i++;
364 
365         dataNode = dataNode->next;
366     }
367 }
```

Si se necesita saber de la existencia de un elemento en específico dentro de la estructura se hace uso del método *"find()"* que devuelve un booleano, donde se verifica cada elemento dentro de la estructura, navegando desde *head*, elemento por elemento, hasta encontrar coincidencia con el dato solicitado, de lo contrario se retorne *false*.

Script: ListArr.cpp

```
369 bool ListArr::find(int v)
370 {
371     DataNode *actualNode = head;
372 
373     while (actualNode != NULL)
374     {
375         for (int i = 0; i < actualNode->count; i++)
376             if (actualNode->container[i] == v)
377                 return true;
378         actualNode = actualNode->next;
379     }
380     return false;
381 }
```

Análisis teórico

Se procede a realizar un análisis teórico donde se describirá la complejidad de los métodos implementados utilizando la notación asintótica *Big-Oh*. Importante a destacar es que no se describirá el análisis teórico de los métodos que no son definidos en ListArrADT, y tampoco, aquellos métodos cuyo resultado puede ser obtenido de manera trivial, es decir, poseen tan pocas operaciones, o bien, su resultado es intuitivo debido al nombre que reciben los métodos, por lo que no es práctico mencionarlos en este apartado.

Método: size()

```
305 int ListArr::size()
306 {
307     return root->quantity;
308 }
```

Análisis asintótico
1 Total: O(1)

Método: insertNode()

```
197 void ListArr::insertNode(DataNode *&dataNode)
198 {
199     DataNode *newDataNode = new DataNode(dataNode->nCapacity);
200
201     // se conecta el nuevo nodo con el resto de la estructura
202     if (dataNode->next != NULL)
203         newDataNode->next = dataNode->next;
204
205     dataNode->next = newDataNode;
206 }
```

1   1 1 1 Total: O(1)

Método: moveRight()

```
4 void moveRight(DataNode *&dataNode, int j)
5 {
6     for (int i = dataNode->count - 1; i >= j; i--)
7     {
8         dataNode->container[i + 1] = dataNode->container[i];
9     }
10 }
```

b b Total: O(b)

Método: freeBinaryTree()

```
58 void ListArr::freeBinaryTree(SummaryNode *node)
59 {
60     if (node == nullptr)
61     {
62         return;
63     }
64     freeBinaryTree(node->left);
65     freeBinaryTree(node->right);
66     delete node;
67 }
```

1
1
leafs
leafs
1
Total: O(leafs)

Método: createBinaryTree()

```
175 SummaryNode *ListArr::createBinaryTree(int leaf)
176 {
177     if (leaf == 1)
178     {
179         SummaryNode *summaryNode = new SummaryNode();
180         summaryNode->data = head;
181         return summaryNode;
182     }
183
184     SummaryNode *newNode = new SummaryNode();
185
186     // crear sub-árboles izquierdo y derecho recursivamente
187     newNode->left = createBinaryTree(leaf / 2);
188     newNode->right = createBinaryTree(leaf - leaf / 2);
189
190     updateQuantity(newNode);
191     updateCapacity(newNode);
192
193     return newNode;
194 }
```

1
1
1
1
1
leafs
leafs
leafs
1
Total: O(leafs)

Método: insertBefore()

```
208 void ListArr::insertBefore(DataNode *dataNode, int v, int i)
209 {
210     if (dataNode->count == dataNode->nCapacity)
211     {
212         if (dataNode->next == NULL || dataNode->next->isFull())
213         {
214             setLeafs(getLeafs() + 1);
215             insertNode(dataNode);
216 
217             dataNode->next->container[0] = dataNode->container[dataNode->count - 1];
218             dataNode->next->count = dataNode->next->count + 1;
219 
220             moveRight(dataNode, i);
221             dataNode->container[i] = v;
222 
223             freeBinaryTree(root);
224             root = createBinaryTree(getLeafs());
225             cleanAllParents(root);
226             for (int j = 1; j <= getLeafs(); j++)
227             {
228                 setDataAssigned(false);
229                 assignDataNodes(root, j);
230             }
231             updateTree();
232         }
233         else
234         {
235             moveRight(dataNode->next, 0);
236             dataNode->next->container[0] = dataNode->container[dataNode->count - 1];
237             dataNode->next->count = dataNode->next->count + 1;
238 
239             moveRight(dataNode, i);
240             dataNode->container[i] = v;
241 
242             freeBinaryTree(root);
243             root = createBinaryTree(getLeafs());
244             cleanAllParents(root);
245             for (int j = 1; j <= getLeafs(); j++)
246             {
247                 setDataAssigned(false);
248                 assignDataNodes(root, j);
249             }
250             updateTree();
251         }
252     }
253     else
254     {
255         moveRight(dataNode, i);
256         dataNode->container[i] = v;
257         dataNode->count = dataNode->count + 1;
258         updateTree();
259     }
260 }
```

1
1
1
1
1
1
1
1
leafs
leafs
leafs
leafs
leafs
leafs
leafs
leafs
1
leafs
1
1
leafs
1
leafs
leafs
leafs
leafs
leafs
leafs
leafs
1
leafs
1
1
leafs
Total: O(leafs)

Método: insert_left()

```
310 void ListArr::insert_left(int v)
311 {
312     insertBefore(head, v, 0);
313 }
```

leafs
Total: O(leafs)

Método: insert_right()

```
315 void ListArr::insert_right(int v)
316 {
317     SummaryNode *actualNode = root;
318
319     // Llega al último DataNode de la derecha
320     while (actualNode->right != NULL)
321         actualNode = actualNode->right;
322
323     DataNode *dataNode = actualNode->data;
324     int pos = dataNode->count;
325
326     insertAfter(dataNode, v, pos);
327 }
```

1

leafs
1

1
1

leafs
Total: O(leafs)

Método: insert()

```
329 void ListArr::insert(int v, int i)
330 {
331     try
332     {
333         if (i > root->sCapacity)
334             throw "Invalid index!";
335         if (root->data != NULL)
336             insertBefore(root->data, v, i);
337         else if (i < root->left->quantity)
338             insertRecursivo(root->left, v, i);
339         else
340             insertRecursivo(root->right, v, i - root->left->quantity);
341     }
342     catch (const char *message)
343     {
344         cerr << message << endl;
345         exit(EXIT_FAILURE);
346     }
347 }
```

1

1
leafs
1
leafs
1
leafs
Total: O(leafs)

Método: print()

```
349 void ListArr::print()
350 {
351     DataNode *dataNode = head;
352 
353     cout << endl;
354     int i = 0;
355 
356     while (dataNode != NULL)
357     {
358         cout << "Node " << i << ":" << endl;
359         for (int j = 0; j < dataNode->count; j++)
360             cout << dataNode->container[j] << " ";
361 
362         cout << endl << endl;
363         i++;
364 
365         dataNode = dataNode->next;
366     }
367 }
```

1
1
1
leafs
leafs
b*leafs
b*leafs
leafs
leafs
leafs
Total: O(leafs)

Método: find()

```
369 bool ListArr::find(int v)
370 {
371     DataNode *actualNode = head;
372 
373     while (actualNode != NULL)
374     {
375         for (int i = 0; i < actualNode->count; i++)
376             if (actualNode->container[i] == v)
377                 return true;
378         actualNode = actualNode->next;
379     }
380     return false;
381 }
```

1
leafs
b* leafs
b* leafs
b* leafs
b* leafs
1
Total: O(b*leafs)

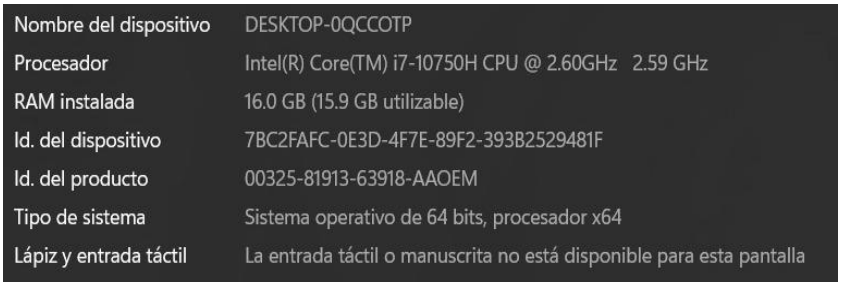
Descripción relevante para análisis experimental

Para el análisis experimental se harán 15 experimentos, y se realizará uso de dos máquinas, la primera máquina, por temas relacionados al tiempo de ejecución que se experimentó (descrito más adelante), donde se concluyó que resultaba poco práctico realizar ahí todas las pruebas, solamente realizó los primeros 4 experimentos, mientras que la segunda realizó los 11 siguientes. A partir de ahora, se referirá a tales máquinas como *"Máquina 1"* y *"Máquina 2"*, respectivamente. Ambas poseen la misma versión de sistema operativo, esto es:

Además, estas son las especificaciones técnicas de:

Máquina 1:

Máquina 2:



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. Además, se realiza un total de 15 experimentos.

Todo el output resultante es adjuntado en el siguiente documento:

https://github.com/PabloSanhueza1/ED-Miniproyecto-1/blob/main/Output%20Análisis%20Temporal.pdf

A continuación, se muestra una tabla los resultados del primer experimento realizado en Máquina 1:

Experimento	n	b	T.insert_left	P.insert_left	T.insert_right	P.insert_right	T. find	P. find
1	10000	128	18223 ms	0.002300 ms	23 ms	0.000000 ms	45 ms	0.004500 ms
1	20000	128	218 ms	0.010900 ms	0 ms	0.000000 ms	170 ms	0.008500 ms
1	40000	128	1258 ms	0.031450 ms	0 ms	0.000000 ms	683 ms	0.017075 ms
1	80000	128	6014 ms	0.075175 ms	0 ms	0.000000 ms	2893 ms	0.036163 ms
1	10000	256	17 ms	0.001700 ms	0 ms	0.000000 ms	42 ms	0.004200 ms
1	20000	256	232 ms	0.011600 ms	0 ms	0.000000 ms	171 ms	0.008550 ms
1	40000	256	1254 ms	0.031350 ms	1 ms	0.000025 ms	671 ms	0.016775 ms
1	80000	256	6054 ms	0.075675 ms	1 ms	0.000013 ms	2717 ms	0.033962 ms
1	10000	1024	18 ms	0.001800 ms	0 ms	0.000000 ms	41 ms	0.004100 ms
1	20000	1024	225 ms	0.011250 ms	0 ms	0.000000 ms	166 ms	0.008300 ms
1	40000	1024	1268 ms	0.031700 ms	0 ms	0.000000 ms	744 ms	0.018600 ms
1	80000	1024	5913 ms	0.073913 ms	1 ms	0.000013 ms	2785 ms	0.034813 ms
1	10000	2048	16 ms	0.001600 ms	1 ms	0.000100 ms	32 ms	0.003200 ms
1	20000	2048	186 ms	0.009300 ms	2 ms	0.000100 ms	129 ms	0.006450 ms
1	40000	2048	971 ms	0.024275 ms	20 ms	0.000500 ms	516 ms	0.012900 ms
1	80000	2048	5927 ms	0.074088 ms	104 ms	0.001300 ms	2146 ms	0.026825 ms

Conclusiones

En lo personal, a pesar de que, en un principio habíamos pensado que la implementación ListArr podía incluso se más rápida en ejecución que la implementación por medio de las estructuras Array y List realizadas durante los laboratorios hasta este momento del semestre. Haciendo énfasis en el "*versus"* contra List, dado que ListArr presentaba similitudes como la de almacenar sus datos en nodos "conectados" entre sí, uno tras otro. Pero dado a que ListArr cuenta con un árbol binario sobre dichos nodos que almacenan datos ingresados, intuitivamente se pudo pensar que, para diferentes métodos, ListArr sería más eficiente que en List. Después de todo, es entendible tales resultados del análisis experimental debido a que la cantidad de datos a ingresar se considera como *"no pequeña"*, por lo que para, cuando era necesario, reestructurar el árbol por completo se vuelve una tarea sumamente costosa.

Por otro lado, en cuanto al desarrollo personal, el trabajo realizado para la implementación de ListArr trae como consiguiente una mejor manera de percibir y ver a los diferentes tipos y estructuras de datos, de tal forma, que hubo un punto en donde diseñar métodos se volvió una tarea no *tan difícil*. Donde también se intentó conseguir la manera más eficiente de realizar dichas implementaciones de esta estructura. Cabe destacar que en ningún momento declaramos que la implementación aquí presentada es sí o sí, la más eficiente de todas.

Finalmente, es importante recalcar que el uso de árboles en realidad no siempre trae consigo resultados como los obtenidos en estos experimentos, sino que, realmente son una excelente herramienta para organizar diferentes procesos en diferentes contextos de uso, por lo que no se puede desmerecer la utilidad de los árboles (en general), por lo tanto, a continuación, se presenta una breve percepción acerca de las ventajas y desventajas del uso de árboles, junto con su utilidad que puede llegar a ser bastante variada.

Un árbol que tiene por hojas arreglos enlazados es útil para organizar y acceder a grandes cantidades de datos que pueden ser agrupados en arreglos de diferentes tamaños. Algunas **ventajas** de esta estructura de datos son que permite un acceso rápido a los elementos del arreglo mediante búsquedas binarias eficientes, y que los arreglos pueden ser de diferentes tamaños, lo que facilita la **agrupación eficiente** de datos.

Además, este tipo de árbol puede ser auto balanceado, lo que mantiene el rendimiento de las búsquedas y acceso a los datos incluso cuando se insertan o eliminan elementos. Sin embargo, existen algunas **desventajas** a tener en cuenta al utilizar esta estructura de datos, tales como que la *implementación puede ser más compleja* que otras estructuras de datos, ya que se debe manejar tanto los nodos del árbol como los arreglos que contienen los datos.

Además, la inserción y eliminación (tarea que no se ha realizado en el proyecto actual) de elementos puede ser más compleja, ya que se debe asegurar que el árbol esté balanceado y que los arreglos no queden demasiado vacíos o llenos. Finalmente, el árbol puede consumir más memoria que otras estructuras de datos, especialmente si los arreglos son grandes y hay muchos nodos.

En resumen, un árbol con hojas que contienen arreglos enlazados es una buena opción para organizar y acceder a grandes cantidades de datos, especialmente si estos datos pueden ser agrupados en arreglos de diferentes tamaños. Sin embargo, es importante tener en cuenta las desventajas mencionadas y **evaluar cuidadosamente** si es la mejor opción para el problema que se está resolviendo.