30/10/2022

EDD

Manual Técnico

Universidad de San Carlos

Estructura de datos

Carlos Eduardo Soto Marroquín

201902502

Segundo semestre octubre 2022

Introducción

Dicho proyecto procesa información ingresada desde un archivo Json, o manualmente por consola. La cual al ser cargada al sistema se almacena en diferentes tipos de estructura de datos, se posee un menú donde se puede escoger cargar los archivos, registrar un usuario, iniciar sesión o generar reportes.

Especificación técnica:

Los requerimientos mínimos para el posterior programa son:

- Lenguajes implementados: C++, Python
- IDE usado: Ninguno
- Editor de código: visual studio code
- Sistema operativo: Ubuntu (Linux)
- Interfaz gráfica: Tkinter
- Librerías utilizadas: Variadic Table, replace, sha256, graphviz, tkinter.

Objetivos:

- Demostrar los conocimientos adquiridos sobre estructuras de datos como: matrices, tablas hash, árboles, blockchain, etc, poniéndolas en práctica durante el desarrollo del juego batalla naval.
- Utilizar el lenguaje C++ para implementar estructuras de datos no lineales, lineales y arbóreas para completar el programa.
- Utilizar la herramienta Graphviz para graficar estructuras de datos lineales.
- Definir e implementar algoritmos de búsqueda, recorrido y eliminación.
- Utilizar el lenguaje de programación Python para el desarrollo de interfaces gráficas.

Alcances del proyecto:

La intención del proyecto es que el desarrollador de esta aplicación pueda definir e implementar las diferentes estructuras de datos no lineales como lo son matrices y árboles. Así mismo, la elaboración de algoritmos, de búsqueda, eliminación y ordenamiento para el

manejo de la información. Las salidas del programa es el reporte que se genera al graficar las diferentes estructuras de datos que se han desarrollado con la herramienta **Graphviz**.

Lógica del programa:

Se tendrán un menú principal con el cual el usuario podrá interactuar con él. A continuación, se presentan las clases utilizadas para el proyecto:

- Funcionamiento del programa:
  - Main
  - Menu
  - User
  - GeneracionImg
- Estructuras lineales:
  - LinkedListCategoria
  - DoublyLinkedListCircularUser
  - Cola tutorial
  - ArticleB
  - LinkedListBarco
  - PilaMov
  - ListaPilaMov
  - Blockchain
  - Hash Table
- Estructuras no lineales
  - Merkle tree
  - Matriz
  - Avl
  - B tree

Funcionamiento del programa:

Main:

Es el encargado de dar inicio al programa, llamando a la función menú, que es importada desde otro módulo. A continuación, se muestra la clase main:

```cpp
#include <iostream>
#include <cstdlib>

#include "menu.cpp"


using namespace std;

int main(int argc, char const *argv[]){
    //cout<<"Inicio del proyecto"<<endl;
    menu();
    return 0;
}
```

Elaboración propia, 2022

Menú:

Es el archivo encargado que contiene toda la lógica del programa, como los menús principales, los submenús, y los procedimientos que complementan los menús principales y submenús.

```cpp
#include "edd/DoublyLinkedListCircularUser.h"
#include "edd/LinkedListCategoria.h"
#include "edd/ColaTutorial.h"
#include "edd/PilaMov.h"
#include "edd/ListaPilaMov.h"

using namespace std;
using std::stoi;
using json = nlohmann::json;

//*Prototipos de funciones
void menu();
void cargaMasiva();
void registrarUsuario();
void login();
void subMenuUser(NodoUsuario*);
void editarInfoUser(NodoUsuario*);
int eliminarUser(string);
void mostrarTutorial();
void comprarArticulos(NodoUsuario*);
void realizarMovimientos(NodoUsuario*);
void generarReportes();

//*EDD y variables a usar
int opcion = 0, edad = 0, monedas = 0;
string nick = "", password = "";
DoublyLinkedListCircularUser* DoublyLinkedListU = new DoublyLinkedListCircularUser();
```

Elaboración propia, 2022

User:

Dicha clase es la plantilla de todos los usuarios que se crearán contando con varios atributos como, el Nick, su contraseña, la cantidad de monedas y una edad, entre otros atributos.

```cpp
class User{
private:
    string nick;
    string password;
    int money;
    int age;
public:
    ListaPilaMov* listaPilaMovimientos;
    User(string nick, string password, int money, int
age){
        this->nick = nick;
        this->password = password;
        this->money = money;
        this->age = age;
        this->listaPilaMovimientos = new ListaPilaMov();
    }

    User(){ //Constructor vacio
        this->nick = "";
        this->password = "";
        this->money = 0;
        this->age = 0;
        this->listaPilaMovimientos = new ListaPilaMov();
    }

    ~User(){ //Destructor

    }
```

Elaboración propia, 2022

generacionImg:

Posee varias funciones, genera un dot y lo pasa a una imagen, otra mueve los archivos de una carpeta a otra, y por último, una función que abre la imagen.

```
void generacionImg(string nombreEstructura, string cadena){
    try{
        ofstream file;
        file.open(nombreEstructura + ".dot", std::ios::out);

        if(file.fail()){
            exit(1);
        }
        file << cadena;
        file.close();
        string command = "dot -Tpng " + nombreEstructura + ".dot -o " + nombreEstructura + ".png";
        system(command.c_str());
    }catch(const std::exception& e){
        cout<<"No se pudo crear la imagen :("<<endl;
    }
}
```

Elaboración propia, 2022

Nodo Usuario:

Es el nodo que contendrá como atributo un objeto de tipo usuario, y que


```
void generacionImg(string nombreEstructura, string cadena){
    try{
        ofstream file;
        file.open(nombreEstructura + ".dot", std::ios::out);

        if(file.fail()){
            exit(1);
        }
        file << cadena;
        file.close();
        string command = "dot -Tpng " + nombreEstructura + ".dot -o " + nombreEstructura + ".png";
        system(command.c_str());
    }catch(const std::exception& e){
        cout<<"No se pudo crear la imagen :("<<endl;
    }
}
```

nos servirá como una pieza para la lista doblemente enlazada circular.

Elaboración propia, 2022

Doubly Linked List circular User:

Es la lista doblemente enlazada circular, que almacenará cada nodo dentro de ella, y la cual posee varios métodos, como de búsqueda, ordenamiento, y de graficar, etc.

```
class DoublyLinkedListCircularUser{
private:
    int tam;
    NodoUsuario* primero;
    NodoUsuario* ultimo;
    void __joinNodes();

public:
    bool isEmpty();
    int length();
    void insertAtStart(string, string, int, int);
    void insertAtEnd(string, string, int, int);
    void deleteAtStart();
    void deleteAtEnd();
    bool deleteNode(string);
    void displayListSE();
    void displayListES();
    NodoUsuario* searchUser(string, string);
    NodoUsuario* searchUser2(string);
    bool searchUserForNick(string);
    bool updateUser();
    void sort();
    void sortReverse();
    void drawList();
    DoublyLinkedListCircularUser();
};
```

Elaboración propia, 2022

Nodo Cola:

Dicho nodo contendrá las coordenadas “X” e “Y”, o el ancho y alto, de los movimientos que se realicen en el tutorial del juego.

```
class NodoCola{
private:
    int anchoX;
    int altoY;
    NodoCola* sig;
public:
    NodoCola(int, int);
    NodoCola();
    ~NodoCola();
    void setAnchoX(int);
    int getAnchoX();
    void setAltoY(int);
    int getAltoY();
    void setSiguiente(NodoCola* sig);
    NodoCola* getSiguiente();
};
```

Elaboración propia, 2022

Cola tutorial:

Es la estructura encargada de almacenar cada uno de los nodos cola. El cual a su vez contiene las coordenadas del juego, y el ancho y alto del tablero, este se usa para desplegar los movimientos del tutorial del juego.

```
class ColaTutorial{
private:
    NodoCola* frente;
    NodoCola* final;
    int tamanio;
public:
    bool isEmpty();
    int length();
    void enqueue(int, int);
    NodoCola* dequeue();
    string in_front();
    void displayQueue();
    void drawQueue();
    ColaTutorial();
    ~ColaTutorial();
};
```

Elaboración propia, 2022

ArticleB:

Dicha clase es la que contiene información de los barcos que se integrarán al juego.

```
class ColaTutorial{
private:
    NodoCola* frente;
    NodoCola* final;
    int tamanio;
public:
    bool isEmpty();
    int length();
    void enqueue(int, int);
    NodoCola* dequeue();
    string in_front();
    void displayQueue();
    void drawQueue();
    ColaTutorial();
    ~ColaTutorial();
};
```

Elaboración propia, 2022

Nodo Barco:

Almacena la información del objeto ArticleB, (Barco), y la categoría del barco. Además de mostrar datos y sus apuntadores.

```
class NodoBarco{
public:
    ArticleB* barco;
public:
    string categoria;
    NodoBarco* sig;
    NodoBarco* ant;
    NodoBarco();
    NodoBarco(int, int, string, string,
string);
    void mostrarDatos();
```

Elaboración propia, 2022

Linked List Barco:

Es el encargado de llevar el control de los nodos barco, de ingresarlos, buscarlos, ordenarlos y mostrarlos.

```
class LinkedListBarco{
private:
    int tam;
    NodoBarco* primero;
    NodoBarco* ultimo;
public:
    bool isEmpty();
    int length();
    NodoBarco* returnHead();
    void insertAtStart(int, int, string, string, string
);
    void insertAtEnd(int, int, string, string, string);
    NodoBarco* searchBoat(int);
    bool searchBoat2(int);
    void displayList();
    bool updateBoat();
    void sort();
    void sortReverse();
    void drawList();
    LinkedListBarco();
};
```

Elaboración propia, 2022

Linked List Barco:

Es el encargado de almacenar la lista de barcos como atributo del nodo y el nombre de la categoría a la cual

```
class LinkedListBarco{
private:
    int tam;
    NodoBarco* primero;
    NodoBarco* ultimo;
public:
    bool isEmpty();
    int length();
    NodoBarco* returnHead();
    void insertAtStart(int, int, string, string, string
);
    void insertAtEnd(int, int, string, string, string);
    NodoBarco* searchBoat(int);
    bool searchBoat2(int);
    void displayList();
    bool updateBoat();
    void sort();
    void sortReverse();
    void drawList();
    LinkedListBarco();
};
```

pertenecen.

Elaboración propia, 2022

Linked List Categoria:

Lleva el control general de los nodos categoría, logrando así insertar los barcos en su categoría correspondiente, comprar un artículo de la tienda, dibujar la lista de barcos.

```
class LinkedListCategoria{
private:
    int tam;
    NodoCategoria* primero;
    NodoCategoria* ultimo;
public:
    bool isEmpty();
    int length();
    void insert(int, string, int, string, string);
    NodoCategoria* get(string);
    bool search(string);
    NodoBarco* buyArticle(int);
    void printL();
    void printLTienda(char);
    void drawList();
    LinkedListCategoria();
    ~LinkedListCategoria();
};
```

Elaboración propia, 2022

NodoP:

Es el nodo que lleva en sus atributos las coordenadas que el jugador implementa y que se van apilando en la pila.

```
class NodoP{
public:
    int coordenadaX;
    int coordenadaY;
    NodoP* sig;
public:
    NodoP();
    NodoP(int, int);
    void setCoordenadaX(int);
    int getCoordenadaX();
    void setCoordenadaY(int);
    int getCoordenadaY();
    NodoP* getSiguiente(){
        return this->sig;
    }

    void setSiguiente(NodoP* sig){
        this->sig = sig;
    }
};
```

Elaboración propia, 2022

PilaMov:

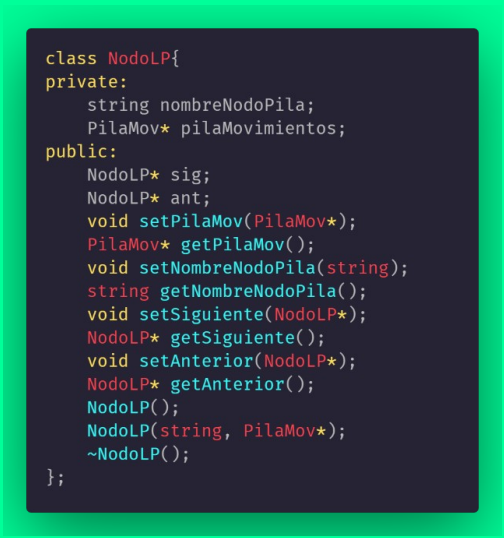
Lleva el control de los nodos pila, que llevan el registro de cada coordenada que ingrese el jugador o mejor mencionado como los movimientos que hace el jugador.

```
class PilaMov{
private:
    string nombrePilaMov;
    int tamanio;
    NodoP* start;
public:
    void setNombrePilaMov(string);
    string getNombrePilaMov();
    bool isEmpty();
    int length();
    void push(int, int);
    NodoP* pop();
    string top();
    string displayStack();
    string drawStack(string);
    PilaMov();
    ~PilaMov();
};
```

Elaboración propia, 2022

NodoLP:

Este nodo llevara el control de una pila de movimientos, y el registro del nombre de la pila.

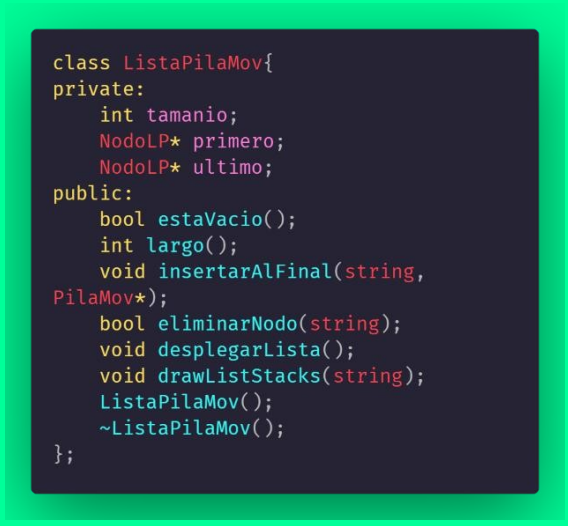


```
class NodoLP{
private:
    string nombreNodoPila;
    PilaMov* pilaMovimientos;
public:
    NodoLP* sig;
    NodoLP* ant;
    void setPilaMov(PilaMov*);
    PilaMov* getPilaMov();
    void setNombreNodoPila(string);
    string getNombreNodoPila();
    void setSiguiente(NodoLP*);
    NodoLP* getSiguiente();
    void setAnterior(NodoLP*);
    NodoLP* getAnterior();
    NodoLP();
    NodoLP(string, PilaMov*);
    ~NodoLP();
};
```

Elaboración propia, 2022

ListaPilaMov:

Será la encargada de llevar el listado de pilas que tendrá la lista de usuarios.



```
class ListaPilaMov{
private:
    int tamanio;
    NodoLP* primero;
    NodoLP* ultimo;
public:
    bool estaVacio();
    int largo();
    void insertarAlFinal(string,
PilaMov*);
    bool eliminarNodo(string);
    void desplegarLista();
    void drawListStacks(string);
    ListaPilaMov();
    ~ListaPilaMov();
};
```

Elaboración propia, 2022

Estructuras lineales:

Lista Circular doblemente enlazada:

Dicha lista se utilizó para almacenar los usuarios de la aplicación. Posee varios métodos que se utilizaron para que fuera funcional, los cuales son los siguientes:

IsEmpty:

```
bool DoublyLinkedListCircularUser::isEmpty(){
    return this->primero == NULL;
}
```

Se utiliza para verificar si la lista se encuentra vacía.

Length:

```
int DoublyLinkedListCircularUser::length(){
    return this->tam;
}
```

Se utiliza para llevar el conteo de nodos en la lista.

JoinNodes:

```
void DoublyLinkedListCircularUser::_joinNodes(){
    if(this->primero != NULL){
        this->primero->ant = this->ultimo;
        this->ultimo->sig = this->primero;
    }
}
```

Se utiliza para unir el primer nodo con el último. Completando
el enlace circular.

InsertAtEnd:

```
void DoublyLinkedListCircularUser::insertAtEnd(string
nick, string password, int money, int age){
    NodoUsuario* nuevo = new NodoUsuario(nick,
password, money, age);
    NodoUsuario* aux = new NodoUsuario();
```

```
void DoublyLinkedListCircularUser::displayListSE(){
    if(this->isEmpty()){
        cout<<"Lista vacia"<<endl;
        return;
    }
     VariadicTable<std::string, std::string , int, int>
ut({"nick", "password", "monedas", "edad"});
     NodoUsuario* aux = new NodoUsuario();
     aux = this->primero;
     while (aux != NULL){
         ut.addRow(aux->user->getNick(),aux->user-
>getPassword(), aux->user->getMoney(), aux->user->getAge());
         aux = aux->sig;

         if(aux == this->primero){
             break;
         }
     }
     ut.print(cout);
     aux = NULL;
     cout<<"\n";
}
```

Se utiliza para desplegar la lista de usuarios de inicio a fin.

DeleteNode:

```
bool DoublyLinkedListCircularUser::deleteNode(string nick)
    { NodoUsuario* actual = new NodoUsuario();
    actual = this->primero;
    bool check = false;
    if(this->primero != NULL && check != true){
        do{
            if(actual->user->getNick() == nick)
                { check = true;
                (actual->ant)->sig = actual->sig;
            }
            actual = actual->sig;
        } while (actual != this->primero && check == false);
```

Se utiliza para eliminar un nodo en cualquier posición.

SearchUser:

```
NodoUsuario* DoublyLinkedListCircularUser::searchUser(string nick, string password){
    if(this->isEmpty()) return NULL; //*Retorna nulo en caso de que la lista este no contenga usuarios
    NodoUsuario* actual = this->primero;
    while (actual != NULL){
        if( (actual->user->getNick().compare(nick) == 0) && (actual->user->getPassword().compare(password) == 0)) return actual; //*retorna el nodo del usuario en caso de encontrarlo
        actual = actual->sig;
        if(actual == this->primero) return NULL; //*Retorna nulo en caso de no encontrar al usuario
    }
```

Se utiliza para buscar un usuario, en caso de que exista retorna el nodo usuario, en caso contrario retorna NULL.

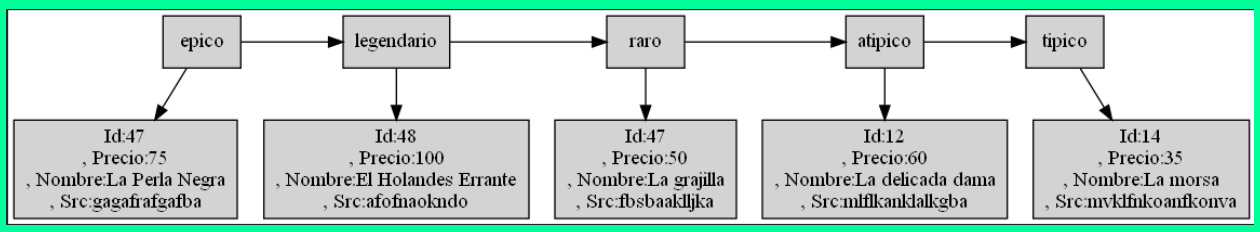
Sort:

```
void DoublyLinkedListCircularUser::sort(){
    NodoUsuario* aux = new NodoUsuario();
    NodoUsuario* actual = new NodoUsuario();
    NodoUsuario* temp = new NodoUsuario();

    if(!this->isEmpty()){
        actual = this->primero;
        while(actual->sig != this->primero){
            aux = actual->sig;
            while(aux != this->primero){
                if(aux->user->getAge() < actual->user->getAge()){
                    temp->user = actual->user;
                    actual->user = aux->user;
                    aux->user = temp->user;
```

```
            }
            actual = actual->sig;
        }
    }
    return; //*Llegados aqui se ordeno todo de menor a
    mayor
```

Lista de Listas:

Esta estructura de datos fue utilizada para los artículos, ya que se van dividiendo por categorías, y según su categoría es como se irán almacenando los datos de los barcos. Dicha estructura, está compuesta de dos listas y dos nodos, se tendrá una lista principal y otra secundaria, al igual que los nodos uno principal y otro secundario.



Pila:

Se utilizó la pila para almacenar los movimientos que realiza el jugador, al mismo tiempo que se pueden retirar de la pila en caso de ser necesario.

Push:

Es usado para ingresar los datos por la cabeza de la pila.

```
void PilaMov::push(int coordenadaX, int coordenadaY){
    NodoP* nuevo = new NodoP(coordenadaX,
    coordenadaY); if(this->isEmpty()){
        this->start = nuevo;
    }else{
        nuevo->setSiguiente(this->start);
        this->start = nuevo;
    }
    this->tamanio += 1;
}
```

Pop:

Es usado para retornar los datos por la cabeza de la pila.

```
NodoP* PilaMov::pop(){
    NodoP* tope= NULL;
    if(!this->isEmpty()){
        tope = this->start;
        this->start = this->start->sig;
```

```
        this->tamanio -=1;
    }
    return tope;
}
```

Top:

Sirve para verificar que valores se tienen en la cima de la pila.

```
string PilaMov::top(){
    if(!this->isEmpty()){
        return "X: " + to_string(this->start->coordenadaX) +
", Y: " + to_string(this->start->coordenadaY);
    }else{
        return "pila vacia";
    }
}
```

Cola:

Se utilizo la cola para almacenar los movimientos que se van cargando al momento de cargar el tutorial al juego.

Enqueue:

Sirve para poner los datos al final de la cola.

```
void ColaTutorial::enqueue(int anchoX, int altoY)
    {  NodoCola*  nuevo  =  new  NodoCola(anchoX,
    altoY); if(this->isEmpty()){
        this->frente = nuevo;
    }else{
        this->final->setSiguiente(nuevo);
    }
    this->final = nuevo;
    this->tamanio += 1;
}
```

Dequeue:

Sirve para retirar los datos al inicio de la cola.

```
NodoCola* ColaTutorial::dequeue(){
    NodoCola* nuevo = this->frente;
    this->frente = this->frente->getSiguiente();

    if(this->isEmpty()){
        this->final = NULL;
    }
    this->tamanio -= 1;
    return nuevo;
}
```

In front:

Sirve para saber que datos poseemos al frente de la cola.

```
string ColaTutorial::in_front(){
    return "anchoX:" + to_string(this->frente->getAnchoX()) +
", altoY:" + to_string(this->frente->getAltoY());
}
```

Lista de Pilas:

Nos sirve para llevar el conteo de jugadas que ha realizado el jugador.

Insertar al final:

Nos sirve para insertar el nombre de la pila y la pila al final de la lista.

```
void ListaPilaMov::insertarAlFinal(string nombreNodoPila,
PilaMov* pilaMovimientos){
    NodoLP* nuevo = new NodoLP(nombreNodoPila,
pilaMovimientos);
```

```
    NodoLP* aux = new NodoLP();

    if(this->estaVacio()){
        this->primero = this->ultimo = nuevo;
    }else{
        aux = this->ultimo;
        this->ultimo = aux->sig = nuevo;
        this->ultimo->setAnterior(aux);
    }
    this->tamanio +=1;
}
```

Desplegar Lista:

Nos sirve para recorrer la lista de pilas e ir viendo cada una de las pilas de movimiento que ha realizado el jugador.

```
void ListaPilaMov::desplegarLista(){
    string cadena = "", pilas = "";
    if(this->estaVacio()){
        cout<<"La lista esta vacia"<<endl;
        return;
    }
    NodoLP* aux = new NodoLP();
    aux = this->primero;
    while (aux !=  NULL){
        cout<<"Pila: "<<aux->getNombreNodoPila()<<endl;
        cout<<aux->getPilaMov()->displayStack()<<endl;
        // cout<<aux->getPilaMov()->drawStack(aux-
>getNombreNodoPila());
        aux = aux->getSiguiente();
    }}
```

Estructuras no lineales:

Árbol B:

El árbol B fue utilizado para almacenar la referencia de cada nodo de la lista de usuarios.

Contenedor:

Es la clase que engloba todo de nuestro Árbol B.

```
//Se hara una clase que podra ir dentro de las llaves para no complicarse la vida
You, last week | 1 author (You)
class Contenedor{
public:
    string idUnico;
    NodoUsuario* nodo_usuario;

    Contenedor(string idUnico, NodoUsuario* nodo_usuario){
        this->idUnico = idUnico;
        this->nodo_usuario = nodo_usuario;
    }

    Contenedor(){
        this->idUnico = "";
        this->nodo_usuario = nullptr;
    }

    ~Contenedor(){} //DESTRUCTOR

    string getIdUnico(){
        return this->idUnico;
    }

    NodoUsuario* getNodoUsuario(){
        return this->nodo_usuario;
    }

    void setIdUnico(string idUnico){
        this->idUnico = idUnico;
    }

    void setNodoUsuario(NodoUsuario* nodo_usuario){
        this->nodo_usuario = nodo_usuario;
    }
```

BTreeNode:

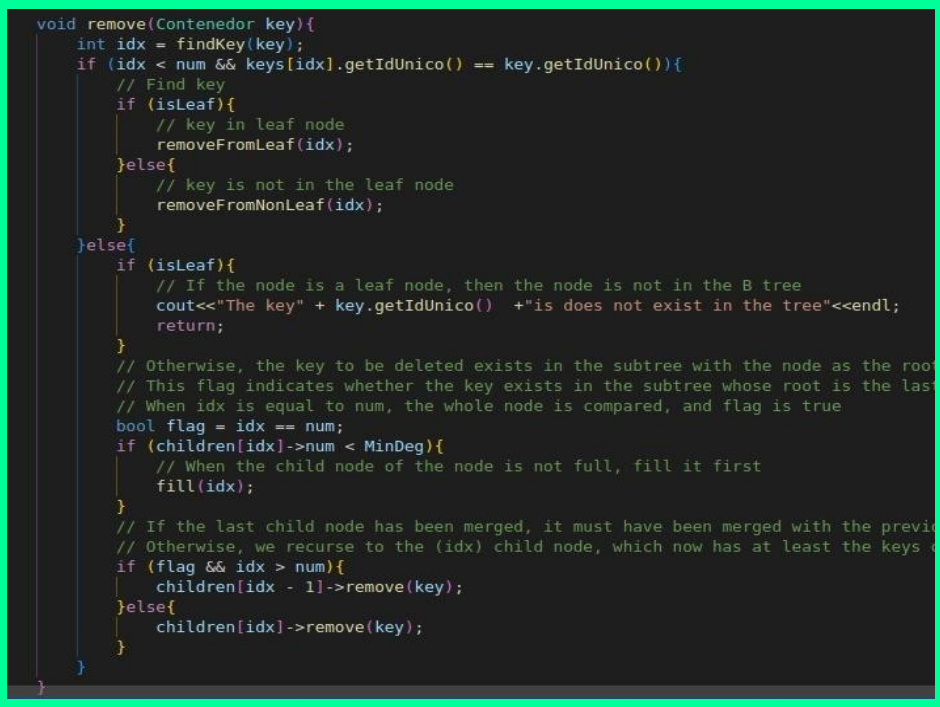
El nodo de nuestro árbol b es el encargado de poder almacenar cada uno de los usuarios encontrados en la

```
    You, last week • arbolB ...
You, 7 days ago | 1 author (You)
class BTreeNode{
private:
    int count;
    std::string buff;
    string id = "";
public:
    std::vector<Contenedor> keys;          //llave de nodos
    int MinDeg;                           //Grado minimo del nodo B-Tree
    std::vector<BTreeNode *> children; //Nodos hijo
    int num;                              //NUmero de llaves del nodo
    bool isLeaf;                          //Verdadero cuando el nodo es una hoja

    //Constructor
    BTreeNode(int deg, bool isLeaf){
        this->MinDeg = deg;
        this->isLeaf = isLeaf;
        this->keys = std::vector<Contenedor>(2 * this->MinDeg - 1);
        this->children = std::vector<BTreeNode *>(2 * this->MinDeg);
        this->num = 0;
        this->id = get_uuid();
    }

    // Find the first location index equal to or greater than key
    int findKey(Contenedor key){
        int idx = 0;
        // The conditions for exiting the loop are: 1.idx == num, i.e. scan all o
        // 2. IDX < num, i.e. key found or greater than key
        //((keys[idx].getIdUnico().compare(key.getIdUnico())) < 0) buena
        while (idx < num && ((keys[idx].getIdUnico().compare(key.getIdUnico())) <
            ++idx;
        }
        return idx;
    }
```

lista:

Remove:

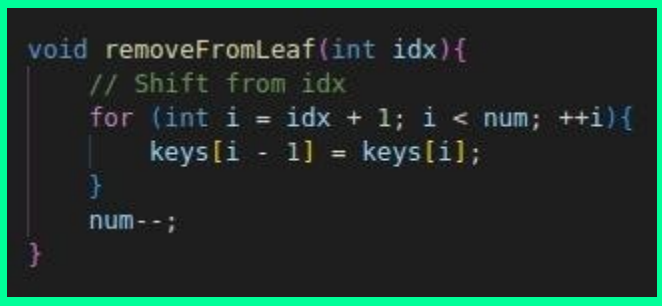
Es el encargado de eliminar un nodo especificado por el usuario de nuestro árbol b:



```
void remove(Contenedor key){
    int idx = findKey(key);
    if (idx < num && keys[idx].getIdUnico() == key.getIdUnico()){
        // Find key
        if (isLeaf){
            // key in leaf node
            removeFromLeaf(idx);
        }else{
            // key is not in the leaf node
            removeFromNonLeaf(idx);
        }
    }else{
        if (isLeaf){
            // If the node is a leaf node, then the node is not in the B tree
            cout<<"The key" + key.getIdUnico()  +"is does not exist in the tree"<<endl;
            return;
        }
        // Otherwise, the key to be deleted exists in the subtree with the node as the root
        // This flag indicates whether the key exists in the subtree whose root is the last
        // When idx is equal to num, the whole node is compared, and flag is true
        bool flag = idx == num;
        if (children[idx]->num < MinDeg){
            // When the child node of the node is not full, fill it first
            fill(idx);
        }
        // If the last child node has been merged, it must have been merged with the previ
        // Otherwise, we recurse to the (idx) child node, which now has at least the keys
        if (flag && idx > num){
            children[idx - 1]->remove(key);
        }else{
            children[idx]->remove(key);
        }
    }
}
```

removeFromLeaf:

A la hora de eliminar un usuario del árbol b tambien lo eliminamos de la hoja de dicho árbol:



```
void removeFromLeaf(int idx){
    // Shift from idx
    for (int i = idx + 1; i < num; ++i){
        keys[i - 1] = keys[i];
    }
    num--;
}
```

removeFromNonLeaf:

Eliminamos el nodo, pero no de la hoja del árbol b:

```
void removeFromNonLeaf(int idx){
    Contenedor key = keys[idx];
    // If the subtree before key (children[idx]) has at least t keys
    // Then find the precursor 'pred' of key in the subtree with children[idx] as the root
    // Replace key with 'pred', recursively delete pred in children[idx]
    if (children[idx]->num >= MinDeg){
        Contenedor pred = getPred(idx);
        keys[idx] = pred;
        children[idx]->remove(pred);
    }else if (children[idx + 1]->num >= MinDeg){
        Contenedor succ = getSucc(idx);
        keys[idx] = succ;
        children[idx + 1]->remove(succ);
    }else{
        // If the number of keys of children[idx] and children[idx+1] is less than MinDeg
        // Then key and children[idx+1] are combined into children[idx]
        // Now children[idx] contains the 2t-1 key
        // Release children[idx+1], recursively delete the key in children[idx]
        merge(idx);
        children[idx]->remove(key);
    }
}
```

getPred y getSucc:

getPred encargado de encontrar el nodo más a la derecha del subárbol de la izquierda y se mueve hacia el nodo más a la derecha hasta llegar al nodo hoja. getSucc es el encargado de que encuentre desde el subárbol de la derecha hasta el de la izquierda y continuar moviendo el nodo más a la izquierda de los hijos hasta llegar al nodo hoja:

```
Contenedor getPred(int idx)
{
    // The predecessor node is the node that always finds the rightmost node from the left subtree
    // Move to the rightmost node until you reach the leaf node
    BTreeNode *cur = children[idx];
    while (!cur->isLeaf){
        cur = cur->children[cur->num];
    }
    return cur->keys[cur->num - 1];
}

Contenedor getSucc(int idx){
    // Subsequent nodes are found from the right subtree all the way to the left
    // Continue to move the leftmost node from children[idx+1] until it reaches the leaf node
    BTreeNode *cur = children[idx + 1];
    while (!cur->isLeaf){
        cur = cur->children[0];
    }
    return cur->keys[0];
}
```

borrowFromPrev:

Llena una llave del nodo anterior:

```
// Borrow a key from children[idx-1] and insert it into children[idx]
void borrowFromPrev(int idx){
    BTreeNode *child = children[idx];
    BTreeNode *sibling = children[idx - 1];
    // The last key from children[idx-1] overflows to the parent node
    // The key[idx-1] underflow from the parent node is inserted as the first key in children[idx]
    // Therefore, sibling decreases by one and children increases by one
    for (int i = child->num - 1; i >= 0; --i){
        // children[idx] move forward
        child->keys[i + 1] = child->keys[i];
    }
    if (!child->isLeaf){
        // Move children[idx] forward when they are not leaf nodes
        for (int i = child->num; i >= 0; --i){
            child->children[i + 1] = child->children[i];
        }
    }
    // Set the first key of the child node to the keys of the current node [idx-1]
    child->keys[0] = keys[idx - 1];
    if (!child->isLeaf){
        // Take the last child of sibling as the first child of children[idx]
        child->children[0] = sibling->children[sibling->num];
    }
    // Move the last key of sibling up to the last key of the current node
    keys[idx - 1] = sibling->keys[sibling->num - 1];
    child->num += 1;
    sibling->num -= 1;
}
```

borrowFromNext:

Rellena una llave del siguiente nodo:

```
// Symmetric with borowfromprev
void borrowFromNext(int idx){
    BTreeNode *child = children[idx];
    BTreeNode *sibling = children[idx + 1];
    child->keys[child->num] = keys[idx];
    if (!child->isLeaf){
        child->children[child->num + 1] = sibling->children[0];
    }
    keys[idx] = sibling->keys[0];
    for (int i = 1; i < sibling->num; ++i){
        sibling->keys[i - 1] = sibling->keys[i];
    }
    if (!sibling->isLeaf){
        for (int i = 1; i <= sibling->num; ++i){
            sibling->children[i - 1] = sibling->children[i];
        }
    }
    child->num += 1;
    sibling->num -= 1;
}
```

insertNotFull:

Cuando se trata de un nodo hoja encuentra el lugar donde debe insertarse la nueva llave y luego encuentra la ubicación del nodo hijo que debe ser insertado:

```
void insertNotFull(Contenedor key){
    int i = num - 1;
    // Initialize i as the rightmost index
    if (isLeaf){
        // When it is a leaf node
        // Find the location where the new key should be inserted
        //keys[i].getIdUnico() > key.getIdUnico() mala
        while(i >= 0 && ((keys[i].getIdUnico().compare(key.getIdUnico())) > 0)){
            keys[i + 1] = keys[i];
            // keys backward shift
            i--;
        }
        keys[i + 1] = key;
        num = num + 1;
    }else{
        // Find the child node location that should be inserted
        while (i >= 0 && ((keys[i].getIdUnico().compare(key.getIdUnico())) > 0)){
            i--;
        }
        if (children[i + 1]->num == 2 * MinDeg - 1){
            // When the child node is full
            splitChild(i + 1, children[i + 1]);
            // After splitting, the key in the middle of the child node moves up, and the child node splits into two
            if(((keys[i + 1].getIdUnico().compare(key.getIdUnico())) < 0)){
                i++;
            }
        }
        children[i + 1]->insertNotFull(key);
    }
}
```

Elaboración propia, 2022

splitChild:

Crea un nodo con la llave obtenida. Luego pasa las propiedades de la variable y a la z e inserta un nuevo nodo hijo y por último mueve la llave de y al nuevo

```
void splitChild(int i, BTreeNode *&y){
    // First, create a node to hold the keys of MinDeg-1 of y
    BTreeNode *z = new BTreeNode(y->MinDeg, y->isLeaf);
    z->num = MinDeg - 1;
    // Pass the properties of y to z
    for (int j = 0; j < MinDeg - 1; j++){
        z->keys[j] = y->keys[j + MinDeg];
    }
    if (!y->isLeaf){
        for (int j = 0; j < MinDeg; j++){
            z->children[j] = y->children[j + MinDeg];
        }
    }
    y->num = MinDeg - 1;
    // Insert a new child into the child
    for (int j = num; j >= i + 1; j--){
        children[j + 1] = children[j];
    }
    children[i + 1] = z;
    // Move a key in y to this node
    for (int j = num - 1; j >= i; j--){
        keys[j + 1] = keys[j];
    }
    keys[i] = y->keys[MinDeg - 1];
    num = num + 1;
}
```

nodo hijo creado

Search:

Clase encargada de buscar un dato dentro del árbol B.

```
BTreeNode *search(string key){
    int i = 0;
    while (i < num && ((key.compare(keys[i].getIdUnico())) > 0)){
        i++;
    }
    if(keys[i].getIdUnico() == key){
        return this;
    }
    if (isLeaf){
        return nullptr;
    }
    return children[i]->search(key);
}
```

toDot:

Clase encargada para generar el dot de nuestro árbol b y poder graficarlo con graphviz:

```
public:
    std::string toDot(){
        Writer();
        Counter();
        //cout<<this->walk()<<endl;
        //"digraph G{\n node [shape = record,height=.1];\n" + this->walk() + "\n" + "}";
        //"digraph G{ \nrankdir=TB;\nnode[color=\"blue\",style=\"rounded,filled\",fillcolor=lightgray, shape=record];\n" + this->walk()
        return "digraph G{ \nrankdir=TB;\nnode[color=\"blue\",style=\"rounded,filled\",fillcolor=lightgray, shape=record];\n" + this->walk
        //return "hola";
    }
```

Árbol AVL:

El árbol AVL es el encargado de almacenar los barcos que cada usuario se compra en su cuenta:

```
class AVL{
private:
    NodoAvl* addInternal(NodoBarco*, NodoAvl*); //*Metodo interno de insercion
    int getAltura(NodoAvl*);
    int max(int, int);
    NodoAvl* srl(NodoAvl*);
    NodoAvl* srr(NodoAvl*);
    NodoAvl* drl(NodoAvl*);
    NodoAvl* drr(NodoAvl*);
    bool search(NodoAvl*, string);
    void PreOrdenInterno(NodoAvl*);
    void PostOrdenInterno(NodoAvl*);
    void InOrdenInterno(NodoAvl*);
    string graficadora(NodoAvl*);
public:
    bool repetido;
    NodoAvl* root;
    AVL();
    ~AVL();
    void add(NodoBarco*);
    void PreOrden();
    void PostOrden();
    void InOrden();
    bool search(string);
    void graficar(string);
};
```

nodoAvl:

Encargado de almacenar los barcos usados por cada compra realizada:

```
You, 4 days ago | 1 author (You)
class NodoAvl{
public:
    NodoBarco* nodoBarco;
    NodoAvl* left;
    NodoAvl* right;
    int alturaN;
public:
    NodoAvl();
    NodoAvl(NodoBarco*);
    ~NodoAvl();
};

NodoAvl::NodoAvl(NodoBarco* nodoBarco){
    this->nodoBarco = nodoBarco;
    left = right = nullptr;
}

NodoAvl::NodoAvl(){
    this->nodoBarco = nullptr;
    left = right = nullptr;
}

NodoAvl::~NodoAvl(){

}
```

Add:

Es el encargado de recibir el dato y enviarlo a addInternal que es el encargado de poder almacenar y ordenar los nodos dentro del árbol AVL:

```
void AVL::add(NodoBarco* nodoBarco){
    this->root = this->addInternal(nodoBarco, this->root);
}
```

getAltura:

Función encargada de obtener la altura de dicho árbol:

```
int AVL::getAltura(NodoAvl* temp){
    if (temp != NULL) {
        return temp->alturaN;
    }
    return -1;
}
```

Srr:

Encargada de realizar la rotación simple por la derecha dentro del árbol AVL:

```
NodoAvl* AVL::srr(NodoAvl* pivote){
    NodoAvl* aux = pivote->right;
    pivote->right = aux->left;
    aux->left = pivote;
    pivote->alturaN = this->max(this->getAltura(pivote->right), this->getAltura(pivote->left)) + 1;
    aux->alturaN = this->max(this->getAltura(aux->right), pivote->alturaN) + 1 ;
    return aux;
}
```

Srl:

Realiza la rotación simple por la izquierda dentro del árbol AVL:

```
NodoAvl* AVL::srl(NodoAvl* pivote){
    NodoAvl* aux = pivote->left;
    pivote->left = aux->right;
    aux->right = pivote;
    pivote->alturaN = this->max(this->getAltura(pivote->left), this->getAltura(pivote->right)) + 1;
    aux->alturaN = this->max(this->getAltura(aux->left), pivote->alturaN) + 1;
    return aux;
}
```

Rotaciones dobles por la izquierda y derecha:

```
NodoAvl* AVL::drl(NodoAvl* pivote){
    pivote->left = this->srr(pivote->left);
    return this->srl(pivote);
}

NodoAvl* AVL::drr(NodoAvl* pivote){
    pivote->right = this->srl(pivote->right);
    return this->srr(pivote);
}
```

addInternal:

Realiza la inserción de cada uno de los datos ingresados/obtenidos al árbol AVL:

```
NodoAvl* AVL::addInternal(NodoBarco* nodoBarco, NodoAvl* root){
    //cout<<value<<endl;
    if(root == NULL){
        //cout<<"root = null"<<endl;
        root = new NodoAvl(nodoBarco);
    }else{
        //cout<<"root = not null"<<endl;
        //(nodoBarco->getId().compare(root->nodoBarco->getId())) < 0;
        if((nodoBarco->getId().compare(root->nodoBarco->getId())) < 0){
            root->left = this->addInternal(nodoBarco, root->left);
            if(this->getAltura(root->right) - this->getAltura(root->left) == -2){
                //(nodoBarco->getId().compare(root->left->nodoBarco->getId())) < 0;
                if ((nodoBarco->getId().compare(root->left->nodoBarco->getId())) < 0){
                    root = this->srl(root);
                }else{
                    root = this->drl(root);
                }
            }
        }else if(nodoBarco->getId() == root->nodoBarco->getId()){
            this->repetido = true;
            cout<<"NO se puede ingresar valores repetidos al arbol avl"<<endl;
        }else{
            if((nodoBarco->getId().compare(root->nodoBarco->getId())) > 0){
                root->right = this->addInternal(nodoBarco, root->right);
                if(this->getAltura(root->right) - this->getAltura(root->left) == 2){
                    if((nodoBarco->getId().compare(root->right->nodoBarco->getId())) > 0){
                        root = this->srr(root);
                    }else{
                        root = this->drr(root);
                    }
                }
            }
        }
    }
    root->alturaN = this->max(this->getAltura(root->left), this->getAltura(root->right)) + 1;
    return root;
}

bool AVL::search(string indice){
```

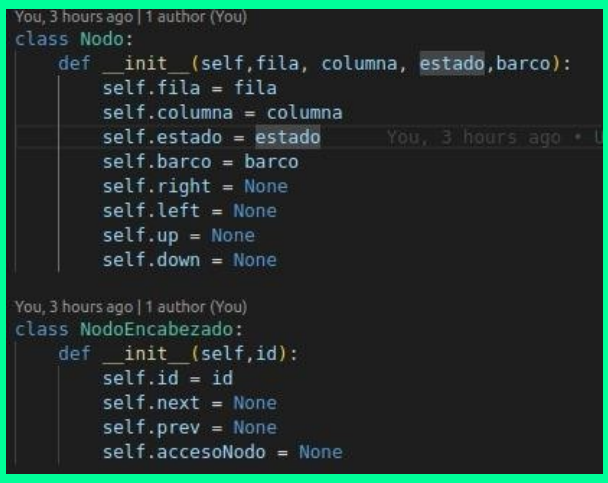
Matriz Dispersa:

Utilizada para la implementación e impresión del tablero de juego:

```
You, 3 hours ago | 1 author (You)
class matriz:
    def __init__(self, tamano):
        self.tamano = tamano
        self.filas = ListaEncabezado()
        self.columnas = ListaEncabezado()

    def insertar(self,fila,columna,estado,barco):
        nuevo = Nodo(fila,columna,estado,barco)

        eFila = self.filas.getEncabezado(fila)
        if eFila ==None:
            eFila = NodoEncabezado(fila)
            eFila.accesoNodo = nuevo
            self.filas.setEncabezado(eFila)

        else:
            if nuevo.columna < eFila.accesoNodo.columna:
                nuevo.right = eFila.accesoNodo
                eFila.accesoNodo.left = nuevo
                eFila.accesoNodo = nuevo

            else:
                actual = eFila.accesoNodo
                while actual.right != None:
                    if nuevo.columna < actual.right.colu
                        nuevo.right = actual.right
                        actual.right.left = nuevo
                        nuevo.left = actual
                        actual.right = nuevo
                        break
                    actual = actual.right
                if actual.right == None:
                    actual.right = nuevo
                    nuevo.left = actual

        #-----------------------------------------------
        eColumna = self.columnas.getEncabezado(columna)
```

<u>Nodo y NodoEncabezado:</u>

Encargados de almacenar cada uno de los barcos y ser utilizados para su inserción dentro del tablero:



```
You, 3 hours ago | 1 author (You)
class Nodo:
    def __init__(self,fila, columna, estado,barco):
        self.fila = fila
        self.columna = columna
        self.estado = estado      You, 3 hours ago • U
        self.barco = barco
        self.right = None
        self.left = None
        self.up = None
        self.down = None

You, 3 hours ago | 1 author (You)
class NodoEncabezado:
    def __init__(self,id):
        self.id = id
        self.next = None
        self.prev = None
        self.accesoNodo = None
```

ListaEncabezado

Clase encargada de poder obtener los números de fila y columna para la inserción de datos en su coordenada correcta:

```
You, 3 hours ago | 1 author (You)
class ListaEncabezado:
    def __init__(self,primero=None):
        self.primero = primero

    def setEncabezado(self,nuevo:NodoEncabezado):
        if self.primero == None:
            self.primero = nuevo

        elif nuevo.id < self.primero.id:
            nuevo.next = self.primero
            self.primero.prev = nuevo
            self.primero = nuevo

        else:
            actual:NodoEncabezado = self.primero
            while actual.next != None:
                if nuevo.id< actual.next.id:
                    nuevo.next = actual.next
                    actual.next.prev = nuevo
                    nuevo.prev = actual
                    actual.next = nuevo
                    break
                actual = actual.next

            if actual.next == None:
                actual.next = nuevo
                nuevo.anterior = actual

    def getEncabezado(self,id):
        actual:NodoEncabezado = self.primero
        while actual!=None:
            if actual.id == id:
                return actual
            actual = actual.next

        return None
```

MarcarDisparo:

Muestra las coordenadas hacia donde fue enviado el disparo:

```
def MarcarDisparo(self,x,y):
    if self.buscar(x,y)==False:
        self.insertar(y,x,"H","X")
    else:
        eColumna:NodoEncabezado = self.columnas.primero
        while eColumna != None:
            actual:Nodo = eColumna.accesoNodo
            while(actual != None):
                if actual.columna==x and actual.fila==y:
                  actual.estado = "H"
                  print("Disparo registrado en: ("+str(x)+","+str(y)+")",actual.estado)
                actual = actual.down
            eColumna = eColumna.next
        print('---------FIN---------')
```

Agregarbarcos:

Encargada de insertar cada uno de los barcos en su coordenada correcta dentro del tablero:

```
def Agregarbarcos(self): #dir representa la dirección del barco 0 = Vertical 1 = horizotal
    global contador,P,D,S,B
    if  contador==0:
        cantidad = int(((self.tamano-1)/10)+1))
        print(cantidad)
        contador = contador+1
        P = P*cantidad
        S = S*cantidad
        D = D*cantidad
        B = B*cantidad
        self.Agregarbarcos()
    else :
        while P!=0:
            dir = random.randint(0,1)
            x = random.randint(1,self.tamano-4)
            y = random.randint(1,self.tamano-4)
            if self.Haybarco(x,y,dir,"P")==False:
                if dir==0:
                    self.insertar(y,x,"L","P")
                    self.insertar(y+1,x,"L","P")
                    self.insertar(y+2,x,"L","P")
                    self.insertar(y+3,x,"L","P")
                    print("Agregado un portaviones")
                else:
                    self.insertar(y,x,"L","P")
                    self.insertar(y,x+1,"L","P")
                    self.insertar(y,x+2,"L","P")
                    self.insertar(y,x+3,"L","P")
                    print("Agregado un portaviones")
                P = P-1
            else: self.Agregarbarcos()
        while S!=0:
            dir = random.randint(0,1)
            x = random.randint(1,self.tamano-3)
```

Haybarco:

Identifica el tipo de barco que se encuentra en el tablero:

```
def Haybarco(self,x,y,dir,tipo):
    i = 0

    if (tipo == "P"):
      if(dir ==0):
        while i <= 4:
            if self.buscar(x,y)==False:
              y=y-1
              i=i+1
            else:
                return True
      else:
        while i <= 4:

            if self.buscar(x,y)==False:
              x=x+1
              i=i+1
            else:
                return True
    elif (tipo == "S"):
      if(dir ==0):
        while i <= 3:
            if self.buscar(x,y)==False:
              y=y-1
              i=i+1
            else:
                return True
      else:
        while i <= 3:

            if self.buscar(x,y)==False:
              x=x+1
              i=i+1
            else:
```

printMatrix0:

Es la encargada de imprimir la matriz como tablero con sus numero de fila y columna al igual que cada uno de los barcos en sus coordenadas correctas:

```
    def printMatrix0(self, nombre):
        contenido = ""     You, 14 hours ago • Compra Barcos _
        cadena = ""
        contenido += """digraph html {
node [fontname="Helvetica,Arial,sans-serif", fontcolor="white"]
abc [shape = none, margin = 0, label=<
<TABLE BORDER = "1" CELLBORDER = "1" CELLSPACING="0" CELLPADDING="10" style='rounded'>\n
                """
        Cabeceras = ""
        for i in range(self.tamano):
            #Cabeceras+="C"+str(i+1)+"[label=\""+str(i+1)+"\",group = "+str(i+2)+", fillcolor=\"FireBrick\", fontcolor=
            Cabeceras += f"<TD BGCOLOR=\"FireBrick\">{str(i+1)}</TD>\n\t"

        contenido += "<TR>\n\t<TD > </TD>" + "\n" + Cabeceras + "\n</TR>\n"

        #print(contenido)
        matrizs = []
        for i in range(1, self.tamano + 1):
            matrizs.append([])
            for j in range(1, self.tamano +1):
                matrizs[i-1].append(str(i) + "X" + str(j))
                #print(str(i) + "x" + str(j), end= " ")
            #print()
        #pprint(matrizs)


        #Para columnas
        eColumna = self.columnas.primero
        while eColumna != None:
            actual = eColumna.accesoNodo
            while(actual != None):
                #if(actual.columna > self.tamano and actual.fila > self.tamano):
                matrizs[actual.columna-1][actual.fila-1] = [actual.columna ,actual.fila, actual.barco, actual.estado]
                #print(str(actual.columna)+","+str(actual.fila)+"("+actual.barco+")")
```

Graficar:

Encargada de generar en la cadena del dot de la gráfica del tablero:

```
def graficar(self):

    raiz = "raiz->F1 \nraiz->C1 "
    rCol ="{rank=same;raiz;"
    rF = ""
    Cabeceras = ""
    Uniones = ""
    for i in range(self.tamano):
        Cabeceras+="F"+str(i+1)+"[label=\""+str(i+1)+"\",group = 1]\n"
        Cabeceras+="C"+str(i+1)+"[label=\""+str(i+1)+"\",group = "+str(i+2)+"]\n"
        if (i+1==self.tamano):
            rCol +="C"+str(i+1)+"}\n"
        else:
            rCol +="C"+str(i+1)+";"
            Uniones +="F"+str(i+1)+"->F"+str(i+2)+"\n"
            Uniones +="F"+str(i+1)+"->F"+str(i+2)+" [dir=back]\n"
            Uniones +="C"+str(i+1)+"->C"+str(i+2)+"\n"
            Uniones +="C"+str(i+1)+"->C"+str(i+2)+" [dir=back]\n"
    eColumna = self.columnas.primero
    Dir1 = ""
    Dir2 = ""
    Dir3 = ""
    Dir4 = ""
    Nodos =""

    while eColumna != None:
        actual = eColumna.accesoNodo
        cont = 0
        while(actual != None):
            while cont!=1:
              Dir1+="F"+str(actual.columna)+"->"
              Dir2+="F"+str(actual.columna)+"->"
              rF+= "{rank=same;F"+str(actual.columna)+";"
              cont = cont+1
            if (actual.down == None):
                Nodos+=("N"+str(actual.columna)+"_"+str(actual.fila)+"[label=\""+str(actual.columna)+","+str(actual.fila)+"("+actua
                Dir1+=("N"+str(actual.columna)+"_"+str(actual.fila)+"")
                Dir2+=("N"+str(actual.columna)+"_"+str(actual.fila)+"[dir=back]")
```

ColorBarco:

Establece el color de cada uno de los barcos según su rareza:

```
def colorbarco(self,tipo,estado):
    if estado!="L":
        return "\"Red\""
    else:
        if tipo=="P":
            return "\"Maroon\""
        elif tipo=="S":
            return "\"Navy\""
        elif tipo=="D":
            return "\"Gray\""
        elif tipo=="B":
            return "\"#008080\""
```

Elaboración propia, 2022

Hash Table:

Utilizada para almacenar las skins de barcos en un carrito de compras:

```
class HashTable():
    def __init__(self, m, min, max):
        self.m = m
        self.min = min
        self.max = max
        self.tabla = [None] * self.m
        self.elementos = 0
        self.contCol = 1 #Contador de colisiones
```

División:

Es la función de dispersión a utilizar:

```
def division(self, k):
    nuevaPosicion = (k % (self.m))
    return nuevaPosicion
```

Resolución de colisiones:

Doble dispersión por medio de la función: (llv mod 3 +1) * i, donde i es el número de colisiones que han ocurrido al insertar un nuevo valor.

```
def dobleHash(self, k):
    nuevaPosicion = ((k % 3) + 1) * self.contCol
    return nuevaPosicion
```

Agregar:

Es la función encargada de agregar el par clave, valor.

```
def agregar(self, key, value):
    #Agrega un elemento a la tabla cerrada
    posicion = self.division(self.toAscii(key))
    #print(posicion)
    if(self.tabla[posicion] is None):
        self.tabla[posicion] = (key, value)
    else:
        #Ejecutar funcion de sondeo para reubicar elemento
        posicion = self.dobleHash(self.toAscii(key))
        #print(posicion)
        while(self.tabla[posicion] is not None):
            self.contCol += 1
            posicion = self.dobleHash(self.toAscii(key))
        self.tabla[posicion] = (key, value)
    self.elementos += 1
    self.rehashing()
```

Rehashing:

Es el responsable de expandir la tabla en caso de que llegue a un 80% de ocupación, y dejar la tabla en 20%

```
def rehashing(self):
    #Si el factor de carga es mayor o igual a max, fijo rehashing, sino solo imprime la lista
    if(round(self.elementos*100 / self.m) >= self.max):
        temp = self.tabla
        #self.printHashTable()
        mprev = self.m
        self.m = int(self.elementos * 100 / self.min) #Se cambia el tamanio del arreglo
        #re-init
        self.elementos = 0 #Se reinicia el contador de elementos
        self.tabla = [None] * self.m #Se le da un nuevo tamanio al arreglo
        for x in range(0, mprev):
            if(temp[x] != None):
                self.agregar(temp[x][0], temp[x][1])
    else:
        pass
        #self.printHashTable()
```

libre.

Buscar:

Función encargada de buscar un elemento a partir de su clave.

```
def buscar(self, key):
    #Determine si un elemento existe en la tabla y determina su posicion
    dic = None
    for x in range(0, len(self.tabla)):
        if(self.tabla[x] != None):
            dato_a_encontrar = self.tabla[x]
            if(dato_a_encontrar[0] == key):
                dic = f"{dato_a_encontrar[0]}: {dato_a_encontrar[1]}"
                return dic
    return dic #No se encontro el dato a buscar dentro del hash table
```

Eliminar:
Encargado de eliminar un elemento de la tabla hash

```
def eliminar(self, key):
    for x in range(0, len(self.tabla)):
        if(self.tabla[x] != None):
            print(self.tabla[x][0], key)
            if(self.tabla[x][0] == key):
                #print("Eliminado")
                self.tabla[x] = None
                self.elementos -= 1
                return True
    return False
```

Clean Hash Table:

Función encargada de limpiar toda la tabla.

```
def cleanHashTable(self):
    for x in range(0, len(self.tabla)):
        self.tabla[x] = None
```

drawHashTable:

Función encargada de dibujar la estructura del hash table usando Graphviz.

```
    def drawHashTable(self, nombre):
        contenido = ""
        cadena = ""
        contenido += """digraph html {
node [fontname="Helvetica,Arial,sans-serif", fontcolor="white"]"""
        contenido += f"\nlabel=\"{nombre}\"\n"
        contenido += """abc [shape = none, margin = 0, label=<
<TABLE BORDER = "1" CELLBORDER = "1" CELLSPACING="0" CELLPADDING="10">\n"""
        cadena += "<TR>\n\t<TD BGCOLOR=\"FireBrick\">Indice</TD>\n\t<TD BGCOLOR=\"FireBrick\">Id</TD>\n\t<TD BGCOLOR=\"Fire
        for x in range(0, self.m):
            if(self.tabla[x] is not None):
                dato_a_encontrar = self.tabla[x]
                cadena += f"<TR>\n\t<TD BGCOLOR=\"#273746\">{x}</TD>\n\t<TD BGCOLOR=\"#273746\">{dato_a_encontrar[0]}</TD>\
        cadena += "</TABLE>>];\n}"
        contenido += cadena
        dotX = "./EDDimg/HashTable_{}_html.dot".format("Base")
        file = open(dotX, "w")
        file.write(contenido)
        file.close()
        result = "./EDDimg/HashTable_{}_html.png".format(nombre)
        system("dot -Tpng " + dotX + " -o " + result)
        commando = "xdg-open ./EDDimg/HashTable_{}_html.png".format(nombre)
        system(commando)
```

PrintHashTable:
Imprime los valores del hash table en consola.

```
def printHashTable(self):
    print("[", end="")
    for x in range(0, len(self.tabla)):
        print(" ", self.tabla[x], end="")
    print("]", f"{round(self.elementos*100/self.m)}%")
```

toAscii:

Función encargada de convertir sus valores a ascii.

```
def toAscii(self, cadena):
    result = 0
    for char in cadena:
        result += ord(char)
    return result
```

Node:

Es la clase nodo del árbol de merkle, responsable de llevar las transacciones.

```
class Node:
    def __init__(self, left, right, value: str, content)-> None:
        self.left: Node = left
        self.right: Node = right
        self.value = value
        self.content = content

    @staticmethod
    def hash(val: str)-> str:
        return hashlib.sha256(val.encode('utf-8')).hexdigest()

    @staticmethod
    def doubleHash(val: str)-> str:
        return Node.hash(Node.hash(val))
```

Merkle Tree:

Es la clase que lleva el control de los nodos de merkle.

```
class MerkleTree:
    def __init__(self, values: List[str])-> None:
        self.__values = self.calculateLength(values)
        self.__buildTree(self.__values)
```

__buildTree:

Función encargada de construir el árbol, por medio de una lista de datos, en caso de ser impar, se copia el último valor de la lista, ya que el árbol de merkle siempre está lleno.

```
def __buildTree(self, values: List[str])-> None:
    leaves: List[Node] = [Node(None, None, Node.doubleHash(e), e) for e in values]
    if len(leaves) % 2 == 1:

        leaves.append(leaves[-1:][0]) # duplicate last elem if odd number of elements
    self.root: Node = self.__buildTreeRec(leaves)
```

__buildTreeRec:

Es la función encargada de ir colocando los nodos del árbol de forma jerárquica.

```
def __buildTreeRec(self, nodes: List[Node])-> Node:
    half: int = len(nodes) // 2

    if len(nodes) == 2:
        return Node(nodes[0], nodes[1], Node.doubleHash(nodes[0].value + nodes[1].value), nodes[0].content + nodes[1].c

    left: Node = self.__buildTreeRec(nodes[:half])
    right: Node = self.__buildTreeRec(nodes[half:])
    value: str = Node.doubleHash(left.value + right.value)
    content: str = left.content + right.content
    return Node(left, right, value, content)
```

Graficar:

Función encargada de graficar el árbol de merkle por medio de Graphviz.

```
def graficar(self, nombre):
    print(self.__values)
    cadena = ""
    cadena += "digraph G { \n"
    cadena += "rankdir=TB; \n"
    cadena += "label=\""
    cadena += nombre + "\"\n"
    cadena += "fontname=\"Arial Black\"\n" + "fontsize=\"25pt\"\n"
    cadena += "node[color=\"blue\",style=\"rounded,filled\",fillcolor=lightgray, shape=record, fontname=\"Arial\"];\n"
    cadena +=  self.graficadora(self.root)
    cadena += "} \n"
    dotX = "MerkleTree_{}.dot".format("Base")
    file = open(dotX, "w")
    file.write(cadena)
    file.close()
    result = "MerkleTree_{}.png".format(nombre)
    system("dot -Tpng " + dotX + " -o " + result)
    commando = "xdg-open " + result
    system(commando)
    #print(cadena)
```

Graficadora:

Función complementaria recursiva para graficar el árbol de merkle.

```
def graficadora(self, node:Node):
    cadena = ""

    if((node.right is None) and (node.left is None)):
        cadena = "\"nodo" + node.value + "\"" + "[label =\"" + node.value + "\"]; \n"

        if (self.isNumeric(node.content)):
            cadena += "\"nodo" + node.content + "\"" + "[label =\"" + str(-1) + "\"]; \n"
            cadena += "\"nodo" + node.value + "\"" + "->" + "\"nodo" + node.content + "\"[dir=back];\n"
        else:
            cadena += "\"nodo" + node.content + "\"" + "[label =\"" + node.content + "\"]; \n"
            cadena += "\"nodo" + node.value + "\"" + "->" + "\"nodo" + node.content + "\"[dir=back];\n"

    else:
        cadena = "\"nodo" + node.value + "\"" + "[label =\"<C0>|" + node.value + "|<C1> \"]; \n"

    if(node.left is not None):
        cadena += self.graficadora(node.left) +  "\"nodo" + node.value + "\":C0->\"nodo" + node.left.value + "\"[dir=back]\
    if(node.right is not None):
        cadena += self.graficadora(node.right) + "\"nodo" + node.value +  "\":C1->\"nodo" + node.right.value + "\"[dir=back

    return cadena
```

0

calculateLength:

Calcula el largo de la lista, y la aumenta en caso de que no este la lista completa, y retorna una nueva lista con los valores llenos.

```python
def calculateLength(self, lts:list)-> list:
    k = 0
    bin = 2
    while(len(lts) >= 2 ** k):
        k += 1 #sigue creciendo
    bin = bin ** k
    nuevoLarge = bin-len(lts)
    for x in range(0, nuevoLarge):
        lts.append(str((-1) * (x+1)))
    print(lts)
    return lts
```

Clase Data:

Se creo esta clase, para conservar la data de la compra de skins y del usuario que la compró.

```python
class Data:
    _from: str      You, 5 days ago • BlockChain …
    skins: list

    def __init__(self, _from=None, skins=None) -> None:
        self._from = _from
        self.skins = skins
```

Clase Block:

La clase bloque, es la encargada de conservar los datos importantes, como el índice, el tiempo de creación, los datos de la compra, un hash del propio bloque, el hash del bloque previo, la raíz del árbol de merkle y el nonce.

```
class Block:
    difficulty_target = "0000"
    index: int
    timestamp: str
    data: Data
    nonce: int
    previoushash: str
    rootmerkle: str
    hash: str

    def __init__(self, index, data, difficulty_target, previoushash=''):
        self.index = index
        self.difficulty_target = difficulty_target
        self.timestamp = datetime.now().strftime("%d-%m-%Y::%H:%M:%S")
        self.data: Data = data
        self.nonce = 0
        self.previoushash = previoushash
        self.merkleroot = self.calcMerkleRoot()
        self.hash = self.calcBlockhash()
        self.mtEDD = None #Se toma la referencia en memoria del arbol merkle

        self.next = None
        self.back = None
```

calcMerkleRoot:

Es la función encargada de crear el árbol de merkle, y de retornar la raíz del árbol de merkle.

```
def calcMerkleRoot(self)-> str:
    elems = []
    for x in self.data.skins['data']:
        #print(x)
        elems.append(x["SKIN"]+str(x["VALUE"]))
    self.mtEDD = MerkleTree(elems)
    #mt.graficar(mt.getRootHash())
    return self.mtEDD.getRootHash()
```

calcBlockhash:

Hace el calculo del hash del bloque actual, por medio del sha256, y retorna un hash valido, cumpliendo con la prueba de trabajo especificada.

```
def calcBlockhash(self):
    # Verificar el orden de concatenacion
    nonce = 0
    while(True):
        h:str = str(self.index) + str(self.timestamp) + self.previoushash + self.merkleroot + str(nonce)
        nonce += 1
        if(str(hashlib.sha256(h.encode('utf-8')).hexdigest()).find(self.difficulty_target) == 0):
            self.nonce = nonce
            return hashlib.sha256(h.encode('utf-8')).hexdigest()
    #return hashlib.sha256(h.encode('utf-8')).hexdigest()
```

rawblockInfo y blockInfo:

Son funciones que una, coloca los datos en un diccionario, y la otra le da un formato de salida Json, para su posterior visualización, o almacenamiento en un archivo.

```
def rawblockInfo(self):
    return {'index': str(self.index), 'timestamp': self.timestamp, 'nonce': str(self.nonce), "data": {"FROM": self.data._fr

def blockInfo(self):
    return json.dumps(self.rawblockInfo(), indent=4)
```

Blockchain:

La clase blockchain es la encargada de unir todos los bloques, que se van creando mediante el proceso de compras de los jugadores.

```
class Blockchain:
    difficulty_target = "0000"

    def __init__(self) -> None:
        self.head = None
        self.tail = None
        self.count = 0
```

IsEmpty:

Función encargada de saber si el blockchain se encuentra vacío o no.

```
def isEmpty(self):
    return self.head == None
```

insertBlock:

Función encargada de insertar bloques de información, con todos los datos necesarios.

```
def insertBlock(self, _from, skins):
    data =  Data(_from, skins)
    #print(data.skins)
    if(self.isEmpty()):
        previoushash = '0000'
        #nonce = self.proof_of_work(previoushash)
        new_block = Block(self.count, data, self.difficulty_target, previoushash)
        self.head = self.tail = new_block
        #Se crea el bloque en json y se almacena en la carpeta de blockchain
        self.createJSONBlock(new_block)
    else:
        aux = self.tail
        previoushash = aux.hash
        #nonce = self.proof_of_work(aux.nonce)
        new_block = Block(self.count, data, self.difficulty_target, previoushash)
        self.tail = aux.next = new_block
        self.tail.back = aux
        #Se crea el bloque en json y se almacena en la carpeta de blockchain
        self.createJSONBlock(new_block)
    self.count += 1
```

drawBlockChain:

Función encargada de dibujar el blockchain.

```
def drawBlockchain(self):
    if(self.isEmpty()):
        return 'no se puede graficar'
    else:
        temporal:Block = self.head
        cont = 0
        cadena = ""
        cadena += "digraph G { \n"
        cadena += "rankdir=LR; \n"
        cadena += 'node[shape=box style=filled color="#ffffff" fontcolor="white" fillcolor="black"];\n'
        cadena += 'graph [fontsize=15 fontname="Verdana" compound=true];\n'
        cadena += 'bgcolor="SteelBlue";\n'


        while(temporal):

            cadena += "  subgraph cluster_" + str(cont) + "_block {\n"
            cadena += '  bgcolor="#283747";\n  fontcolor="white";\n'
            cadena += f'    label="Block header {cont}";\n'
            cadena += f'    "Time stamp {cont}:{temporal.timestamp}" "Nonce:{str(temporal.nonce)}";\n'
            cadena += f'    "Previoushash:{temporal.previoushash}" "Hash:{temporal.hash}";\n'
            cadena += f'    "Merkle root:{temporal.merkleroot}";\n'

            cadena += "  }\n"
            if(temporal != self.head):
                cadena += f'   "Hash:{temporal.previoushash}"->"Hash:{temporal.hash}" [ltail=cluster_{str(cont-1)}
            temporal = temporal.next
            cont += 1
        cadena += "}"
        print(cadena)
```

(variable) previoushash: str

CreateJSONBlock:

Función encargada de crear un bloque y se guarda como archivo.

```
def createJSONBlock(self, bloque:Block):
    timestamp:str = bloque.timestamp.replace(":","_")
    file = open(f"blockchain/{bloque.index}_{timestamp}.json", "w")
    file.write(bloque.blockInfo())
    file.close()
```

rep:

Función utilizada para representar los objetos de la clase.

```
def __repr__(self) -> str:
    rep = "[\n"
    if (self.isEmpty()):
        rep += "]"
    else:
        aux:Block = self.head
        while aux is not None:
            rep += '\t'
            rep += f'{aux.blockInfo()},' if aux.next != None else f'{aux.blockInfo()}'
            rep += '\n'
            aux = aux.next
        rep += "]"
    return rep
```

Grafo:

Sirve para crear el grafo del jugador ganador usando Graphvhiz.

```
    def Grafo(self):
        grafo = """
digraph G {
node [fontcolor="white", color=white , style=filled, fillcolor=Crimson]
bgcolor = "#508912";
label="Grafo";
"""
        i = 1
        eColumna:NodoEncabezado = self.columnas.primero
        while eColumna != None:
            actual:Nodo = eColumna.accesoNodo
            grafo+="\t"+str(i)+"\n"
            while(actual != None):
                #print(actual.columna,",",actual.fila,'        ',actual.estado)

                if actual.estado =="H":
                    grafo+="\t"+str(i)+"->"+str(actual.fila)+"\n"
                    #print(str(actual.fila) +","+str(actual.columna))
                    #print(str(actual.fila)) #+","+str(actual.columna))
                actual = actual.down

            i=i+1
            eColumna = eColumna.next
        if i<self.tamano+1:
            while i< self.tamano+1:
                grafo+="\t"+str(i)+"\n"

                i=i+1
        grafo += "\n}"
        #print(grafo)
        dotX = "./EDDimg/Grafo.dot"
        file = open(dotX, "w")
        file.write(grafo)
        file.close()
        result = "./EDDimg/Grafo.png"
        os.system("dot -Tpng " + dotX + " -o " + result)
```

ListaAdyacencia:

Función encargada de graficar la lista de adyacencia del jugador ganador.

```
def ListaAdyacencia(self):
    nodosc = ""
    nodos = ""
    i = 1
    eColumna:NodoEncabezado = self.columnas.primero
    while eColumna != None:
        actual:Nodo = eColumna.accesoNodo
        nodosc+= "\t\t\t n"+str(i)
        nodos+= "\t\t\t n"+str(i)+"[label =\""+str(i)+"\"]\n"
        while(actual != None):
            #print(actual.columna,",",actual.fila,'        ',actual.estado)
            if actual.estado =="H":
                nodosc+="->"+"n"+str(actual.fila)+"_"+str(actual.columna)
                nodos+="\t\t\t n"+str(actual.fila)+"_"+str(actual.columna)+"[label =\""+str(actual.fila)+"\"]\n"
            if actual.down == None:
                nodosc+="[color = \"#172A3A\"];\n"
            actual = actual.down


        i=i+1
        eColumna = eColumna.next
    if i<self.tamano+1:
        while i< self.tamano+1:
            nodosc+="n"+str(i)+"\n"
            nodos+="n"+str(i)+"[label =\""+str(i)+"\"]"+"\n"
            i=i+1
    encabezado="""
    digraph G {
        rankdir=LR
        compound = true;
        labelloc="t";
        bgcolor = "#508991";
        fontcolor = Black;
        color = "#004346"

    subgraph cluster_0 {
        node [style=filled,shape=note,fillcolor="#74b3ce",color = "#172A3A"];
        label = "Lista de adyacencia"
    """
    dot = encabezado+"\n"+nodos+"\n"+nodosc+"\n } \n }" #Copiar el metodo para generar la imagen
    dotX = "./EDDimg/ListaAdyacencia.dot"
    file = open(dotX, "w")
    file.write(dot)
    file.close()
    result = "./EDDimg/ListaAdyacencia.png"
```