Año del Fortalecimiento de la Soberanía Nacional

Trabajo final del curso Algoritmos y Estructuras de Datos (CC182)

Integrantes:

Quispe Palacin, Diego Eloy - u202012453 (Ciencias de la Computación)

Urquiaga Rodríguez, Rafael Adrián - u202010506 (Ingeniería de Software)

Palacios Torres, Juan Pablo - u201915206 (Ingeniería de Software)

Profesor:

Heider Ysaias Sanchez Enriquez

Sección: CC31

Lima - Perú

24 de febrero del 2022

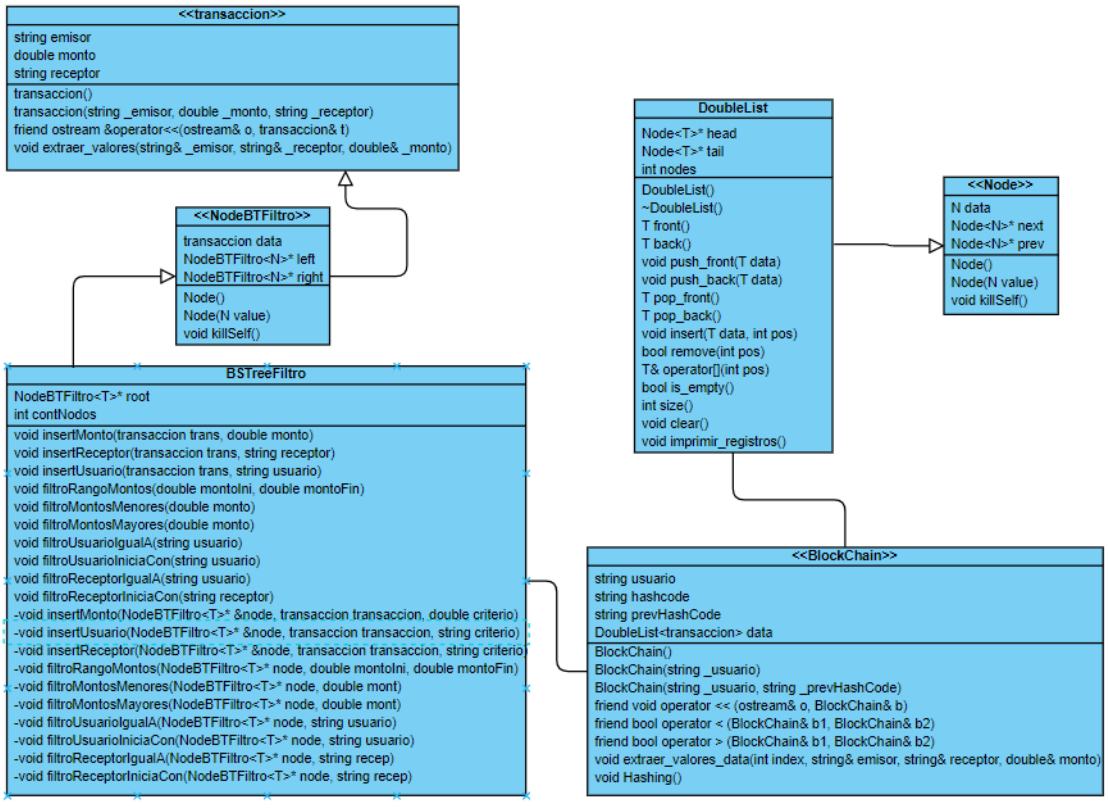
Introducción

En el presente informe se detallan las acciones tomadas por el grupo para la resolución del problema planteado y la administración de ellas por cada integrante. Asimismo, el requerimiento de aplicaciones como GitHub y Visual Studio LiveShare para el control de versiones del código y el entorno de trabajo en equipo. Finalmente, también se mostrará el proceso para la construcción de la BlockChain como estructura ideal para obtener una complejidad O (log n).

Descripción del caso de estudio planteado por el grupo

El ejercicio trata, en resumen, de generar la BlockChain. Esto es crear Block Chains identificables por su hashcode y el hash code del anterior. Los cuales a su vez tendrán almacenada información de transacciones realizadas desde un usuario a otro con un monto de dinero de por medio. Asimismo, la naturaleza de la Block Chain requiere que está misma sea segura; por lo tanto, cuando se modifique la información de un Blockchain anterior que ya ha sido completado, el hashcode de este será actualizado y, para que el enlace con los Block Chains posteriores se mantenga, también se efectuarán actualizaciones en su hash code.

Diagrama de clases de entidades principales

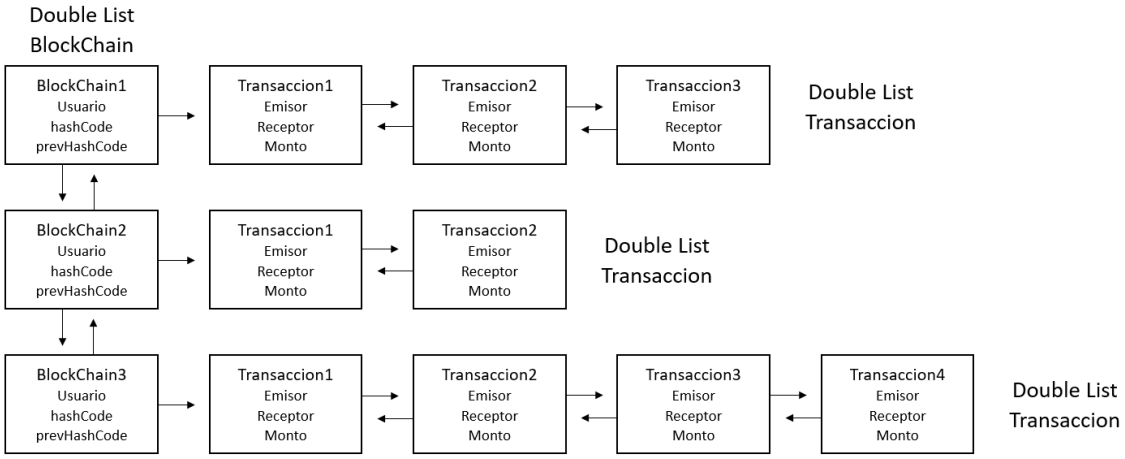


Definición de TDA y estructuras de datos a usar o diseño de archivos a utilizar

Lista Doble (Double List)

Esta lista se caracteriza por tener dos punteros uno que apunta al nodo siguiente y otro que apunta al nodo previo. Asimismo, cada nodo de esta lista contiene información que puede guardar y recuperar con ayuda de los punteros mencionados anteriormente.

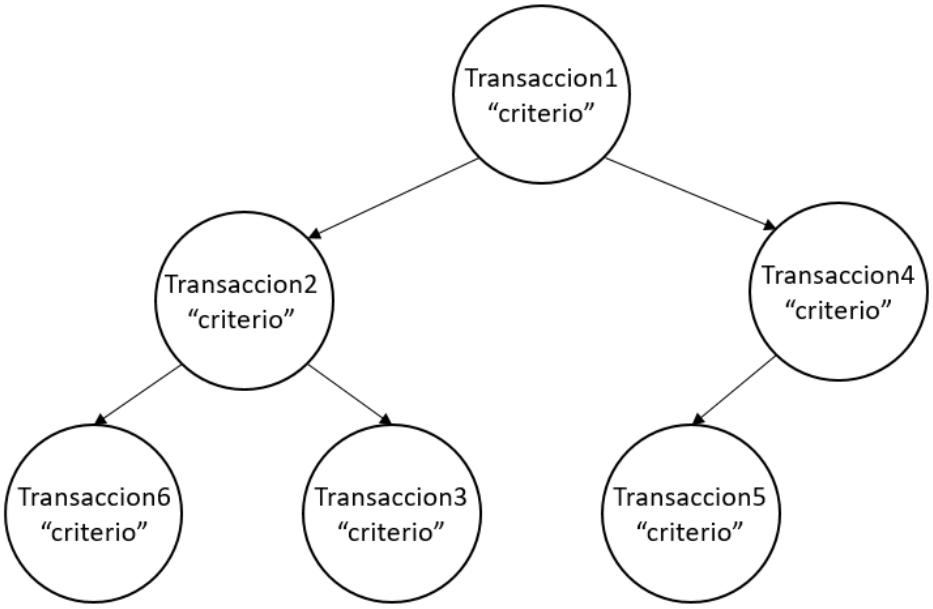
En esta estructura de datos hemos realizado el guardado de datos de cada struct correspondientemente. Es decir, e utilizó para crear una lista doble que guarda los BlockChains y dentro de cada BlockChain tiene de información el usuario, hashCode, prevHashCode y otra lista doble de las transacciones realizadas con el usuario del BlockChain.



Árbol

El árbol tiene como objetivo insertar conjuntos de datos según una característica que definirá su posición en este a través del hijo izquierdo o derecho.

En este caso, hemos realizado un archivo árbol, que trabaja con las transacciones, se crearán árboles según cantidad de criterios necesarios para desarrollar el filtro. Asimismo, se realizaron las funciones de filtrado de información en el archivo correspondiente al árbol de transacciones.



Complejidad en notación detallada y Big O de los métodos de las estructuras de datos.

A continuación, se mostrarán unas algunas imágenes de los métodos importantes de las estructuras de datos, más relevantes, utilizadas. La notación detallada se mostrará en cada línea de código y el Big O se ubicará al costado de la cabecera de cada función.

Métodos del struct Transacción

```
struct transaccion{
    string emisor; // ---> 1
    double monto; // ---> 1
    string receptor; // ---> 1
    transaccion(){}
    transaccion(string _emisor, double _monto, string _receptor) // O(1)
    {emisor = _emisor; monto = _monto; receptor = _receptor;} // ---> 3

    friend ostream &operator<<(ostream& o, transaccion& t) { // O(1)
        o << t.monto << " from " << t.emisor << " to " << t.receptor << "\n"; // ---> 12
        return o; // ---> 1
    }
    void extraer_valores(string& _emisor, string& _receptor, double& _monto){ // O(1)
        _emisor = emisor; // ---> 1
        _receptor = receptor; // ---> 1
        _monto = monto; // ---> 1
    }
};
```

Métodos del BlockChain

```
struct BlockChain {
    string usuario; // ---> 1
    string hashcode; // ---> 1
    string prevHashCode; // ---> 1
    DoubleList<transaccion> data; //---> 1

    BlockChain() {
    }

    BlockChain(string _usuario) { //---> O(1)
        usuario = _usuario; //---> 1
    }

    BlockChain(string _usuario, string _prevHashCode) {//---> O(1)
        usuario = _usuario; //---> 1
        prevHashCode = _prevHashCode;//---> 1
    }

    //friend bool operator < (string& criterio, BlockChain& b){···

    friend bool operator == (BlockChain& b, string& s){ //---> O(1)
        if (b.usuario == s) return true; // ---> 2
        else return false; // ---> 1
    }

    friend void operator << (ostream& o, BlockChain& b){ //---> O( k )
        b.data.imprimir_registros(); // ---> K ,  K: representa la cantidad de registros del BlockChain
    }

    friend bool operator < (BlockChain& b1, BlockChain& b2){ //---> O(1)
        if (b1.hashcode < b2.hashcode) return true; // ---> 2
        else return false; // ---> 1
    }

    friend bool operator > (BlockChain& b1, BlockChain& b2){ //---> O(1)
        if (b1.hashcode > b2.hashcode) return true; // ---> 2
        else return false; // ---> 1
    }

    void extraer_valores_data(int index, string& emisor, string& receptor, double& monto){ //---> O( k )
        data[index].extraer_valores(emisor,receptor,monto); //---> K
    }

    void Hashing() {
```

```cpp
// q : cantidad de letras del usuario , K: cantidad de registros del BlockChain
void Hashing() { // O( q ) + O( K )
    int aux = 0; // ---> 1
    int aux2 = 0; // ---> 1
    int aux3 = 0; // ---> 1
    string t; // ---> 1
    int j = 0; // ---> 1
    int num; // ---> 1
    t.resize(11); // ---> 1

    for (int i = 0; i < usuario.length(); ++i) { //---> 1 + q(2+4+2) = O( q )
        //suma todos los ascii del usuario
        aux = aux + (int)usuario[i]; // ---> 4
    }
    int tamanho = data.size(); // ---> 2
    for (int i = 0; i < tamanho; ++i) { // ---> 1 + k(1+7+2) = O( K )
        string s = data[i].receptor; // --> 3
        //suma los ascii de la primera letra de cada persona
        aux2 = aux2 + (int)s[0]; // ---> 4
    }
    for (int i = 0; i < tamanho; ++i) { // 1 + k(1+6+2) = O( k )
        double s = data[i].monto; // ---> 3
        //suma los montos
        aux3 = aux3 + (int)s; // ---> 3
    }
    for (int i = 0; i < 11; ++i) { // ---> 1 + 11(1 + 7 + 13 + 2) = 254
        if (i < 5) { // ---> 1 + 4 = 5
            //Condicional para insertar solo la parte del usuario
            num = aux + (int)prevHashCode[j]; // ---> 4
        }
        else if (5 <= i && i < 10) { // ---> 3 + 4 = 7
            //Condicional para insertar solo la parte del receptor
            num = aux2 + (int)prevHashCode[j]; // ---> 4
        }
        else {// ---> 4
            //Condicional para insertar solo la parte del monto
            num = aux3 + (int)prevHashCode[j]; // ---> 4
        }
        // Condicionales para realizar un hash estable que depende del prevhash y de los datos del bloque
        // Se inserta el char en la pos i del hash
        if (num % 2 == 0 && (int)prevHashCode[j] %2 == 0) {// 7 + 6 = 13
                num = 48 + num % 10; // ---> 3
                t[j] = char(num); // ---> 3
        }else if(num % 2 == 0 && (int)prevHashCode[j] % 2 != 0){// 7 + 6 = 13
                num = 65 + num % 25; // ---> 3
                t[j] = char(num); // ---> 3
        }else if (num % 2 != 0 && (int)prevHashCode[j] % 2 != 0) {// 7 + 6 = 13
                num = 97 + num % 25; // ---> 3
                t[j] = char(num); // ---> 3
        }else if (num % 2 != 0 && (int)prevHashCode[j] % 2 == 0) { // 7 + 6 = 13
                num = 35 + num % 4; // ---> 3
                t[j] = char(num); // ---> 3
        }
            ++j; // ---> 2
    }
    //Se asigna el valor al hash
    hashcode = t; // ---> 2
}
```

Métodos de los criterios de inserción en el árbol

```
//INSERTAR MONTOS
void insertMonto(NodeBTFiltro<T>* &node, transaccion transaccion, double criterio){ // ---> Log n
    if(node == nullptr) // ---> 1
        node = new NodeBTFiltro<T>(transaccion); // ---> 2
    else if(criterio < node->data.monto) // ---> 3
        insertMonto(node->left, transaccion, criterio); // ---> Log n
    else if(criterio >= node->data.monto) // ---> 3
        insertMonto(node->right, transaccion, criterio);// ---> Log n
}

//void inserCantTran(NodeBTFiltro<T>*& node, T BlockChain){···

//INSERTAR USUARIOS
void insertUsuario(NodeBTFiltro<T>* &node, transaccion transaccion, string criterio){ // ---> Log n
    if(node == nullptr) // ---> 1
        node = new NodeBTFiltro<T>(transaccion); // ---> 2
    else if(criterio < node->data.emisor) // ---> 3
        insertUsuario(node->left, transaccion, criterio); // ---> Log n
    else if(criterio >= node->data.emisor)// ---> 3
        insertUsuario(node->right, transaccion, criterio);// ---> Log n
}
//INSERTAR RECEPTOR
void insertReceptor(NodeBTFiltro<T>* &node, transaccion transaccion, string criterio){ // ---> Log n
    if(node == nullptr) // ---> 1
        node = new NodeBTFiltro<T>(transaccion); // ---> 2
    else if(criterio < node->data.receptor) // ---> 3
        insertReceptor(node->left, transaccion, criterio);// ---> Log n
    else if(criterio >= node->data.receptor) // ---> 3
        insertReceptor(node->right, transaccion, criterio);// ---> Log n
}
```

Métodos de los criterios de filtrado en el árbol

```
//FILTRO RANGO DE PRECIOS
void filtroRangoMontos(NodeBTFiltro<T>* node, double montoIni, double montoFin){ // ---> Log n
    if(node == nullptr) return; // ---> 2
    else if(montoIni <= node->data.monto && node->data.monto <= montoFin){ // ---> 7 + ...
        cout << node->data; // ---> 2
        filtroRangoMontos(node->right,montoIni,montoFin); // ---> Log n
        filtroRangoMontos(node->left,montoIni,montoFin); // ---> Log n
    }
    else if(montoFin < node->data.monto) // ---> 3
        filtroRangoMontos(node->left,montoIni,montoFin);// ---> Log n
    else if(montoIni > node->data.monto) // ---> 3
        filtroRangoMontos(node->right,montoIni,montoFin);// ---> Log n
    return;// ---> 1
}
//FILTRO MONTOS MENORES
void filtroMontosMenores(NodeBTFiltro<T>* node, double mont){ // ---> Log n
    if(node == nullptr) return; // --->2
    else if(mont >= node->data.monto){// ---> 3
        cout << node->data;// ---> 2
        filtroMontosMenores(node->right,mont);// ---> Log n
        filtroMontosMenores(node->left,mont);// ---> Log n
    }
    else if(mont < node->data.monto)// ---> 3
        filtroMontosMenores(node->left,mont);// ---> Log n
    return;// ---> 1
}
//FILTRO MONTOS MAYORES
void filtroMontosMayores(NodeBTFiltro<T>* node, double mont){ // ---> Log n
    if(node == nullptr) return;// ---> 2
    else if(mont > node->data.monto)// ---> 3
        filtroMontosMayores(node->right,mont);// ---> Log n
    else if(mont <= node->data.monto){// ---> 3
        cout << node->data;// ---> 2
        filtroMontosMayores(node->right,mont);// ---> Log n
        filtroMontosMayores(node->left,mont);// ---> Log n
    }
    return;// ---> 1
}
```

Conclusiones

1. Los blockchain fueron una gran practica para entender mejor las estructuras de datos.
2. Los árboles indexados nos ayudaron bastante a mejorar la complejidad de búsqueda de los datos que deseábamos.
3. El trabajo en equipo nos ayudó a realizar el trabajo en el tiempo acordado.

Referencias

Sena, M.(enero,2019). Estructuras de Datos. Primera parte — Arrays, Linked lists, Stacks, Queues.Medium,TechWo.Recuperado de https://medium.com/techwomenc/estructuras-de-datos-a29062de5483.

Sedgewick, R., et. al. (2011) Algorithms, Fourth Edition. Pearson.

Cormen, H., et. al. (2009) Introduction to Algorithms, MIT Press.

Allen, Mark (2014) Data Structures and Algorithms Analysis in C++, Fourth Edition. Pearson.