

## **Reflexión: Actividad integradora #2**



Miguel Arriaga Velasco – A01028570

Campus Santa Fe

Programación de estructuras de datos y algoritmos fundamentales

12 de abril del 2021

## 1. Estructuras de Datos Lineales

En el mundo de la programación existe una gran diversidad de problemas, ya que la naturaleza de estos es diferente, hay diversas estructuras de datos que se pueden utilizar dependiendo del problema que se tenga, ver figura 1.

Algunas de las estructuras de datos lineales más populares son las siguientes:

- Arreglos
- Vectores
- Listas encadenadas simples
- Listas encadenadas dobles
- Filas (Queue)
- Pilas (Stack)

Para seleccionar la estructura de datos que se quiere utilizar, primero se debe de analizar el problema. En este caso, se utilizó una lista encadenada doble y una pila. Ya que fue lo que se solicitó.

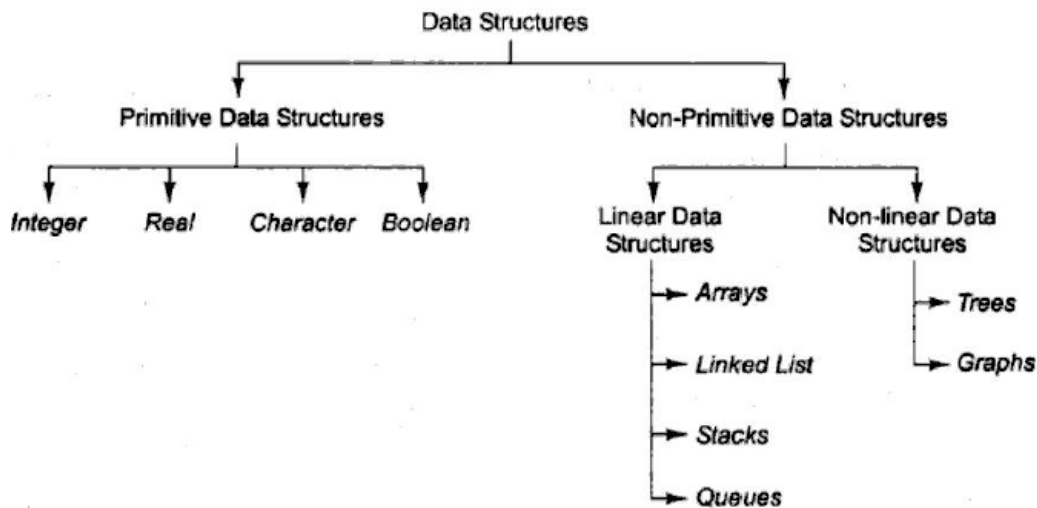


Figura 1. Diversas estructuras de datos.

### 1.1. Listas doblemente encadenadas

Ya se sabe que una lista encadenada está compuesta de una serie de apuntadores de tipo “Nodo”, que contienen cierta información y a su vez apuntan al siguiente Nodo.

Sin embargo, una lista doblemente encadenada, contiene un apuntador extra en cada Nodo. Este apuntador apunta al Nodo anterior. De esta manera la lista se puede recorrer al derecho y al revés, ver figura 2 (GeeksForGeeks, 2021).

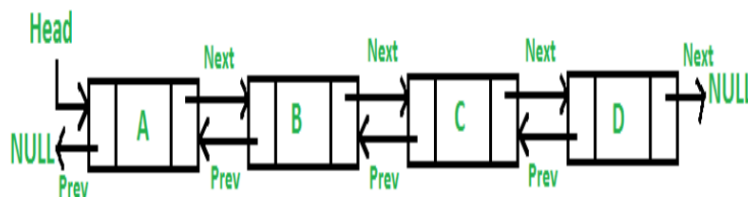


Figura 2. Representación gráfica de una lista doblemente ligada.

Para esta actividad integradora, se necesitaba ordenar y buscar dentro de la estructura de datos seleccionada. Una lista doblemente ligada es mucho mejor opción porque nos permite recorrer la lista en cualquiera de los dos sentidos y no siempre desde el principio, reduciendo el número de operaciones necesarias. Esto se puede ver en el método “getAt”.

### 1.2. Pilas (Stacks)

Para esta entrega, también se solicitó que se utilizara una Pila o una Fila, para la implementación del método de ordenamiento. De esta forma, se reemplazó la recursividad del ordenamiento, por esta estructura de datos. Ya que los algoritmos de ordenamiento QuickSort y MergeSort son de naturaleza recursiva, se decidió utilizar una Pila, que es lo que mejor imita a la recursividad.

La Pila, es una estructura de datos lineal que sigue el orden de entrada y salida LIFO (Last Input First Output). Esto significa que los elementos que han sido más recientemente añadidos a la Pila también serán los primeros en salir de ella, ver figura 3 (GeeksForGeeks, 2019).

Algunos usos de la pila son:

- Cualquier aplicación que requiera controlar un orden inverso al orden de la entrada original.
- Ejemplos:
  - Control de llamadas a funciones
  - Conversión y evaluación de expresiones

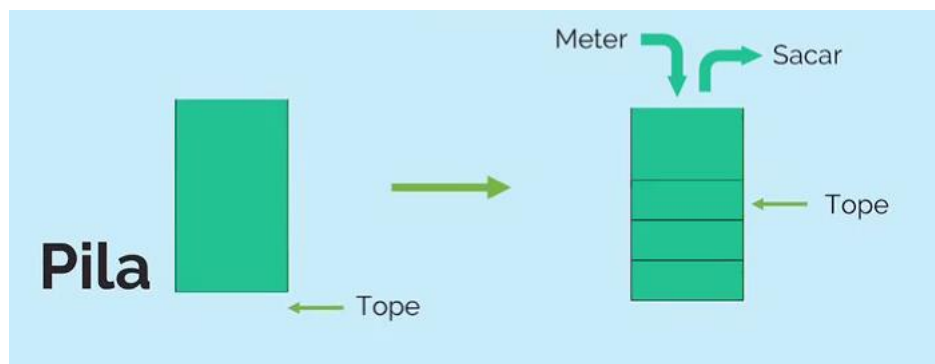


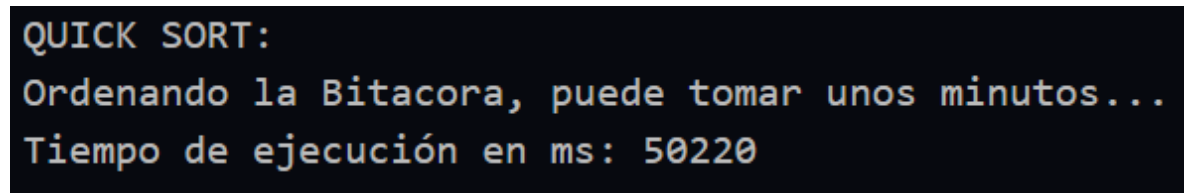
Figura 3. Representación gráfica de una Pila.

## 2. Ordenamiento

En la entrega pasada, se comprobó experimentalmente que quickSort fue más rápido que mergeSort al ordenar los registros en un vector. Se empezó a trabajar en esta entrega con la misma hipótesis, por lo que se realizó una implementación de QuickSort iterativamente, para ordenar la Bitácora. Sin embargo, por las limitaciones en tiempo y dificultad para realizar una implementación iterativa en listas encadenadas, el resultado no fue el óptimo. El tiempo de ejecución del ordenamiento tuvo un promedio de 50200 ms en mi computadora personal, casi 6 veces más que el ordenamiento del vector en la entrega pasada, ver figura 4. Sin embargo, esto tiene una explicación.

Para la implementación iterativa de QuickSort, se requiere acceder a los elementos por su índice. Esto no es algo que se pueda hacer en tiempo constante para una lista ligada, por lo que cada vez que se quería buscar un Nodo, se necesitó recorrer la lista de manera lineal. A pesar de que hay formas más ingeniosas de resolver este problema, la falta de tiempo no lo permitió. Sin embargo, se probó una solución recursiva de QuickSort, que resultó ser mucho más eficiente que la iterativa, con tiempos cercanos al ordenamiento de la entrega pasada.

Adicionalmente, GeeksForGeeks menciona que Quick Sort es preferido para arreglos, mientras que MergeSort se recomienda para listas ligadas. Es muy posible que, para esta entrega, el algoritmo MergeSort hubiera sido más rápido.

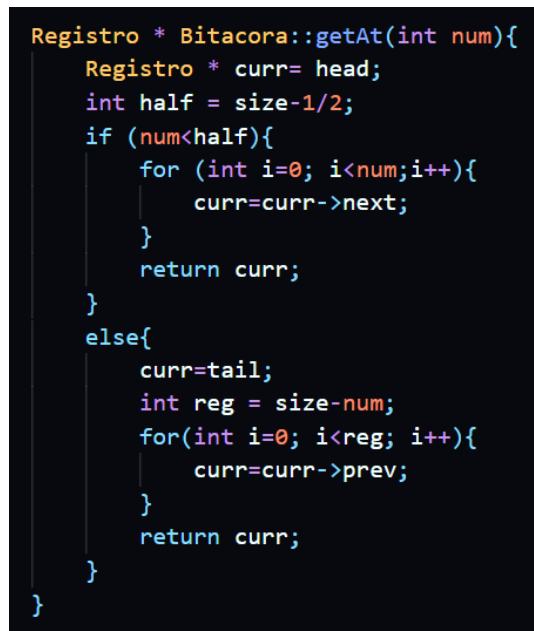


QUICK SORT:  
Ordenando la Bitacora, puede tomar unos minutos...  
Tiempo de ejecución en ms: 50220

Figura 4. Tiempo de ejecución de QuickSort

### 2.1. Análisis de complejidad de Quick Sort

Para analizar la complejidad temporal de Quick Sort, primero debemos de obtener la complejidad temporal del algoritmo “getAt”, que es el que obtiene el índice. El código para este algoritmo es el siguiente (figura 6).



```
Registro * Bitacora::getAt(int num){
    Registro * curr= head;
    int half = size-1/2;
    if (num<half){
        for (int i=0; i<num;i++){
            curr=curr->next;
        }
        return curr;
    }
    else{
        curr=tail;
        int reg = size-num;
        for(int i=0; i<reg; i++){
            curr=curr->prev;
        }
        return curr;
    }
}
```

Figura 6. Algoritmo “getAt”

Empíricamente, se puede deducir que este algoritmo corre en tiempo  $O(n/2)$ , esto se debe a que hay un for loop que recorre una mitad del arreglo. Esto solo es posible por la implementación de la lista doblemente ligada. El peor caso de getAt, se simplifica como  $O(n)$ .

Lo siguiente que se debe de hacer, es analizar el algoritmo “partition”, ver figura 7.

```

int Bitacora::partition(int l, int h){
    Registro* pivot = getAt(h);
    int i = l-1;
    int j;
    for (j=l; j<=h-1; j++){
        if (getAt(j)->getFecha()<=pivot->getFecha()){
            i++;
            intercambiar(i,j);
        }
    }
    intercambiar(i+1,h);
    return i+1;
}

```

Figura 7. Algoritmo “partition”

Como se puede observar en el código, la operación básica más costosa es un `getAt`, que corre en tiempo lineal. Sin embargo, esa operación se debe de realizar un número de veces determinado por un loop `for`. Este loop corre desde `l`(low) hasta `high-1`, en dónde `l` es el primer número del arreglo con el que se está trabajando (ya sea el arreglo original o una parte de este) y `h` es el mayor número de dicho arreglo. También se puede observar que el parámetro de control se incrementa en un valor constante de 1. Por esto, y siguiendo las estrategias para determinar la complejidad de un algoritmo iterativo de manera empírica, podemos determinar que la complejidad del algoritmo “partition” es de  $O(n^2)$ .

Ahora debemos de analizar el algoritmo Quick Sort (figura 7), aquí es donde la cosa se pone algo complicada.

```

void Bitacora::_quickSort(int l, int h){
    int stack[h-1+1];
    int top = -1;

    stack[++top] = l;
    stack[++top] = h;

    while (top >= 0){
        //std::cout<<top<<std::endl;
        h=stack[top--];
        l=stack[top--];

        int pos = partition(l,h);
        //std::cout<<pos<<std::endl;

        if (pos-1 > l){
            stack[++top] = l;
            stack[++top] = pos-1;
        }
        if (pos+1 < h){
            stack[++top] = pos+1;
            stack[++top] = h;
        }
    }
}

```

Figura 8. Algoritmo “Quick Sort”

Ya que el arreglo constantemente se divide en 2 sub-arreglos, se puede deducir que ese loop se ejecuta  $\log_2(n)$  veces. Sin embargo, por cada ejecución del loop se hace una llamada a `partition()`. Es por esto por lo que la complejidad final del algoritmo es muy mala:

$$O(n) = (n^2 \log_2 n)$$

### 3. Análisis de otros algoritmos de la estructura de datos

#### 3.1. Búsqueda binaria

Primero analicemos el algoritmo de búsqueda binaria (figura 9).

```

int Bitacora::binSearch(Fecha clave){
    //solo sirve para arreglos ordenados
    int left=0;
    int right=size-1;
    //int comparaciones=0;
    while(left<=right){
        int mid=int((left+right)/2);
        //comparaciones++;
        if (getAt(mid)->getFecha()==clave){
            right=mid-1;
        }
        else if(getAt(mid)->getFecha()<clave){
            left=mid+1;
        }
        else{
            return (mid);
        }
    }
    return -1;
}

```

Figura 9. Búsqueda binaria

Se observa que la operación más costosa que puede concluir de la decisión es un `getAt`, lo cual corre en tiempo lineal. Se puede observar que el loop corre desde el inicio al final del array. Sin embargo, la variable de control se divide entre dos por cada iteración del loop, ya que este es un valor constante, se determina que el número de veces que corre el algoritmo es de logaritmo base dos (constante entre la cual se divide la variable control).

$$O(n \log_2 n)$$

### 3.2. rangoFechas()

Ahora debemos de considerar que lo que se quiere es que el usuario introduzca dos fechas distintas y se impriman los registros entre ese rango de fechas. Por lo tanto, debemos de considerar que la búsqueda binaria se ejecuta dos veces y se requiere de un tiempo lineal para imprimir los registros dentro del rango. Esto se lleva a cabo en el método `rangoFechas` (figura 10).

```

void Bitacora::rangoFechas(Fecha inicio, Fecha fin){
    int start=binSearch(inicio);
    int finish=binSearch(fin);
    if (start == -1 || finish == -1){
        std::cout<<"No se encontraron esas fechas"<<std::endl;
    }
    else{
        std::cout<<"Resultado de la búsqueda en archivo resultado_busqueda.txt"<<std::endl;
        std::ofstream outfile("resultado_busqueda.txt");
        Registro* temp = getAt(start);
        for (start; start <= finish; start++){
            {
                outfile<<temp->getDato()<<std::endl;
                temp=temp->next;
            }
        }
        outfile.close();
    }
}

```

Figura 10. Método `rangoFechas`

La complejidad de este método es:

$$O(2n \log_2 n + O(k - m))$$

En dónde el primer método indica las dos llamadas de la búsqueda binaria y el segundo método indica la impresión de los registros entre los resultados de la búsqueda: la fecha mayor ( $k$ ) y la fecha menor

(m). En el peor caso, k es la última fecha del arreglo y m es la primera, por lo tanto, la complejidad sería de  $O(n)$ .

Ya que  $O(n)$  crece más lento que  $O(2n \log 2n)$ , la complejidad final de este método es de:

$$O(2n \log n)$$

### 3.3. addLast()

Este método introduce los registros al final de la lista.

```
void Bitacora::addLast(Registro* reg){
    //newnode->next=NULL;
    if (head==NULL){
        head=reg;
        tail=reg;
    }
    else{
        tail->next=reg;
        reg->prev=tail;
        tail=reg;
    }
    size+=1;
}
```

Figura 11. Método addLast

Su complejidad es de  $O(1)$ , es constante ya que en esta implementación existe “tail”, que es el último nodo de la lista. De esta manera, se le puede agregar algo de cualquiera de los dos extremos en tiempo constante.

## 1. Funcionamiento del programa

Este programa tiene como objetivo ordenar y buscar información de una bitácora. El programa está compuesto de 5 archivos. Tres archivos "header", que contienen la declaración e implementación de las clases, "Fecha", "Registro" y "Bitacora". El cuarto archivo es "main.cpp" y es el archivo en dónde se inicializan los objetos tipo "Bitacora" y se utilizan los algoritmos de ordenamiento y búsqueda para ordenar la bitacora y buscar un rango de fechas.

Finalmente, se incluye el archivo "bitacora.txt", que es la base de datos que se quiere ordenar. Esta no es necesaria para el programa y éste es capaz de ordenar otras bases de datos con la misma estructura, cambiando el parámetro del constructor del objeto tipo Bitacora.

NOTA: ES POSIBLE QUE LA PRIMERA VEZ QUE SE CORRA EL CÓDIGO ARROJE UN "SEGMENTATION FAULT", A PARTIR DE LA SEGUNDA CORRIDA DEBERÍA DE FUNCIONAR CORRECTAMENTE.

Para compilar el programa, se debe de acceder al archivo "main.cpp". Dependiendo del IDE en el que se esté trabajando, se puede compilar y correr el código con alguna extensión (boton) o directamente desde la consola con los siguientes comandos:

g++ main.cpp -o main : compila tú programa con nombre main

./main.out - corre el programa



El resultado de main.cpp es la creación de dos archivos: "bitacora\_ordenada.txt", que contiene a la bitácora ya ordenada por quickSort. Además, se imprimirá en pantalla, el tiempo en ms que tardó en ordenar a la bitácora. Posteriormente se imprimirá el resultado de una búsqueda binaria de ejemplo, para la fecha del 9 de Julio a las 16:19:58. Finalmente, se llama al método pideUsuario(), preguntándole al usuario una fecha inicial y una fecha final, primero se pregunta el mes, que debe de estar en formato de tres letras (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec). Posteriormente, se pregunta la hora, que debe de estar en el formato 00:00:00. En donde (Hora:Minuto:Segundo).

Es importante recalcar que para que el programa funcione correctamente, se deben introducir las fechas de manera exacta, de otra manera el programa imprimirá que las fechas introducidas no se encontraron. El resultado de la búsqueda se encuentra en el archivo "resultado\_busqueda.txt".

## **Conclusión**

Es un hecho que las estructuras de datos son de gran importancia en la programación. La velocidad de procesamiento de las computadoras y la memoria de estas todavía es un recurso limitado y asegurarnos de que estos recursos sean administrados de la mejor manera posible es nuestra labor como ingenieros en tecnologías computacionales.

Consecuentemente, se debe aspirar tanto por las estructuras, como los algoritmos más eficientes, tanto en uso de memoria como de tiempo de ejecución. Las estructuras de datos son como la base principal de cualquier programa, y esta define cómo se van a realizar las funciones posteriores de éste. Conocer y entender diversas estructuras, en qué situaciones una es mejor que otra y saber implementarlas es algo vital. En esta entrega se logró demostrar que un mismo problema puede ser resuelto a través de diferentes estructuras de datos.

Finalmente, de la misma forma que en la entrega anterior, la habilidad de un ingeniero de investigar y buscar información para resolver problemas fue un factor clave para resolver esta actividad.

## **Referencias**

GeeksForGeeks (2019) "Quick Sort vs. Merge Sort". Recuperado desde: <https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/>

GeeksForGeeks (2019) "Doubly Linked List". Recuperado desde: <https://www.geeksforgeeks.org/doubly-linked-list/>

GeeksForGeeks (2019) "Stack Data Structure". Recuperado desde: <https://www.geeksforgeeks.org/stack-data-structure/>

Presentaciones por el Dr. Eduardo Arturo Rodríguez Tello, durante la materia de Programación de estructuras de datos y algoritmos fundamentales