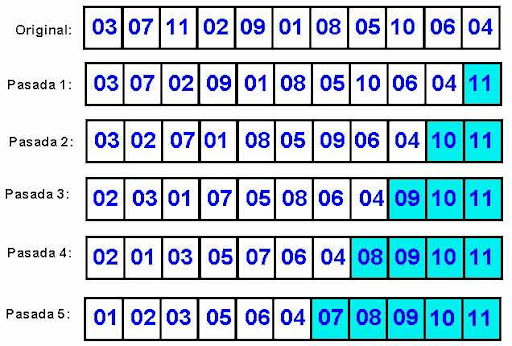
**Burbuja (Bubble sort).**

1. **Definición.**

El algoritmo de ordenamiento Burbuja (bubble sort en inglés) funciona intercambiando repetidamente elementos adyacentes de la lista si están en el orden incorrecto. Es decir, si el elemento de la izquierda es mayor a la derecha, se intercambian. Este proceso se repite hasta que se completa el barrido se realizara ningún intercambio, dando a entender que la lista está ordenada (Cormen et al., 2022).

Específicamente, bubble sort parte del inicio de la lista y compara cada par de elementos adyacentes. Si el primer elemento es mayor que el segundo, los intercambia. Luego pasa al segundo y tercer elemento, intercambiándolos si están desordenados, y así sucesivamente. Cuando llega al final, vuelve a empezar desde el principio. Esto se repite hasta que en una iteración no se realiza ningún intercambio, indicando que la lista está ordenada (Cormen et al., 2022).



**Imagen 1** Funcionamiento de algoritmo Burbuja de forma gráfica.

1. **Propiedades.**

* Complejidad del espacio: O(1).
* Rendimiento en el mejor caso: O(n).
* Rendimiento promedio de casos: O(n\*n).
* Rendimiento en el peor de los casos: O(n\*n).
* Estable: Sí.

***main.cpp***

#include <iostream>  
#include "ListaSimple.h"  
#include "Validacion.h"  
  
**int** **main**(){  
 Lista lista;  
 **do** {  
 system("cls");  
 std::cout << "**\n\t**ORDENAMIENTO BURBUJA CON LISTA SIMPLE**\n\n**" <<  
 "1. Insertar al inicio**\n**" <<  
 "2. Eliminar**\n**" <<  
 "3. Mostrar**\n**" <<  
 "4. Ordenamiento metodo burbuja**\n**" <<  
 "5. Salir**\n\n**";  
 **switch** (ingresar\_enteros("Ingresar la opcion: ")) {  
 **case** **1**:  
 lista.insertarAlInicio(ingresar\_enteros("**\n\n**Ingresar un numero: "));  
 std::cout << std::endl;  
 system("pause");  
 **break**;  
 **case** **2**:  
 lista.eliminar(ingresar\_enteros("**\n\n**Ingresar el numero a eliminar: "));  
 std::cout << std::endl;  
 system("pause");  
 **break**;  
 **case** **3**:  
 lista.imprimir();  
 std::cout << std::endl;  
 system("pause");  
 **break**;  
 **case** **4**:  
 lista.ordenarBurbuja();  
 system("pause");  
 **break**;  
 **case** **5**:  
 **return** **0**;  
 **default:**  
 **break**;  
 }  
 } **while** (true);  
}

***ListaSimple.h***

#pragma once  
#include <iostream>  
  
// Definición de la clase Nodo  
**class** **Nodo** {  
**public:**

**int** dato;  
 Nodo\* siguiente;  
  
 // Constructor  
 Nodo(**int** valor) : dato(valor), siguiente(nullptr) {}  
};  
  
// Clase Lista que contiene operaciones básicas  
**class** **Lista** {  
**private:**  
 Nodo\* cabeza;  
  
**public:**  
 // Constructor  
 Lista() {  
 cabeza = nullptr;  
 }  
  
 // Método para insertar un elemento al inicio de la lista  
 **void** insertarAlInicio(**int** valor) {  
 Nodo\* nuevoNodo = **new** Nodo(valor);  
 nuevoNodo->siguiente = cabeza;  
 cabeza = nuevoNodo;  
 }  
  
 // Método para imprimir la lista  
 **void** imprimir() {  
 Nodo\* temp = cabeza;  
 **while** (temp != nullptr) {  
 std::cout << temp->dato << " -> ";  
 temp = temp->siguiente;  
 }  
 std::cout << "NULL" << std::endl;  
 }  
  
 // Método para eliminar un elemento de la lista dada una clave (valor)  
 **void** eliminar(**int** clave) {  
 Nodo\* temp = cabeza;  
 Nodo\* prev = nullptr;  
  
 // Buscar el nodo con la clave dada  
 **while** (temp != nullptr && temp->dato != clave) {  
 prev = temp;  
 temp = temp->siguiente;  
 }  
  
 // Si la clave no se encuentra en la lista, no hay nada que eliminar  
 **if** (temp == nullptr) {  
 std::cout << "**\n**El elemento " << clave << " no esta en la lista." << std::endl;  
 **return**;  
 }  
  
 // Eliminar el nodo encontrado  
 **if** (prev != nullptr)  
 prev->siguiente = temp->siguiente;  
 **else**  
 cabeza = temp->siguiente;  
  
 **delete** temp;  
 }  
  
 // Método para ordenar la lista usando el algoritmo de burbuja  
 **void** ordenarBurbuja() {  
 **if** (cabeza == nullptr || cabeza->siguiente == nullptr) {  
 // La lista está vacía o tiene solo un elemento, no es necesario ordenar  
 **return**;  
 }  
  
 **bool** intercambiado;  
 Nodo\* actual;  
 Nodo\* anterior = nullptr;  
  
 **do** {  
 intercambiado = false;  
 actual = cabeza;  
  
 **while** (actual->siguiente != anterior) {  
 **if** (actual->dato > actual->siguiente->dato) {  
 // Intercambiar los valores de los nodos  
 **int** temp = actual->dato;  
 actual->dato = actual->siguiente->dato;  
 actual->siguiente->dato = temp;  
 intercambiado = true;  
 }  
 actual = actual->siguiente;  
 }  
 anterior = actual;  
 } **while** (intercambiado);  
 }  
  
 // Destructor para liberar la memoria asignada a los nodos de la lista  
 ~Lista() {  
 Nodo\* temp = cabeza;  
 **while** (temp != nullptr) {  
 Nodo\* siguiente = temp->siguiente;  
 **delete** temp;  
 temp = siguiente;  
 }  
 cabeza = nullptr;  
 }  
};

***Validacion.h***

#pragma once  
#include <conio.h>  
#include <iostream>  
  
**int** **borrar**(**char**\* datos, **int**& i) {

**if** (i > **0**) {  
 printf("**\b** **\b**");  
 i--;  
 datos[i] = '\0';  
 **return** i;  
 }  
 **return** **0**;  
}  
  
**int** **ingresar\_enteros**(std::string msj) {  
 **char**\* datos = **new** **char**[**10**];  
 **char** c;  
 **int** i = **0**;  
  
 std::cout << msj;  
 **while** ((c = \_getch()) != **13** && i < **9**) {  
 **if** ((c >= '0' && c <= '9') || c == **8**) {  
 **if** (c == **8**) {  
 i = borrar(datos, i);  
 }  
 **else** {  
 printf("%c", c);  
 datos[i++] = c;  
 }  
  
 }  
 }  
 datos[i] = '\0';  
 **return** atoi(datos);  
}

1. **Conclusiones.**

El método de burbuja es fácil de entender e implementar, su rendimiento no es ideal para conjuntos de datos grandes. Otros algoritmos de ordenamiento son generalmente preferidos en la práctica para mejorar la eficiencia.

1. **Referencias**

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.

https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf