Universidad De Las Fuerzas Armadas ESPE

Ingeniería en tecnologías de la información

Tema: Búsqueda Binaria

Kevin Andino

Estructura de Datos NRC 29852

08 de diciembre de 2025

Ejercicio 1:

- Se cuenta con un archivo directo llamado **ALUMNOS.DAT** con información de alumnos de una universidad. El archivo se abre para lectura y se **indexa** por medio de su clave primaria ($\#\text{legajo}$). El índice se mantiene en **memoria principal**. Se ofrece la posibilidad al usuario de buscar alumnos por **legajo**. La **búsqueda binaria** se aplica sobre el índice y se accede en forma directa (**seek**) al archivo para recuperar el registro. Se asume que el archivo está **ordenado** de manera ascendente por su clave primaria ($\#\text{legajo}$). Si éste no fuese el caso, el índice debería ordenarse antes de aplicar una **búsqueda binaria** sobre él.

Aclaración: no se provee el archivo ALUMNOS.DAT (debido a que es binario). Deberá ser **generado** antes de la ejecución de este programa.

```
#include <iostream>
#include <string>
#include <cstring>

// Definición de constantes
const int TAMANIO_NOMBRE = 60;
const int TAMANIO_REGISTRO = sizeof(int) + TAMANIO_NOMBRE; // 4 bytes (legajo)
+ 60 bytes (nombre)

class Alumno {
private:
   int legajo;
   char nombre[TAMANIO_NOMBRE];

public:
   // Constructor
   Alumno(int l = 0, const std::string& n = "") : legajo(l) {
      // Asegura que el nombre tenga el tamaño fijo, rellenando con '\0'
      std::strncpy(nombre, n.c_str(), TAMANIO_NOMBRE - 1);
      nombre[TAMANIO_NOMBRE - 1] = '\0';
   }

   // Método para obtener el legajo (clave primaria)
   int getLegajo() const {
      return legajo;
   }

   // Método para imprimir el alumno
   void mostrar() const {
      std::cout << "Legajo: " << legajo << ", Nombre: " << nombre << std::endl;
   }

   // El objeto Alumno ya está en el formato que se escribirá directamente en el
archivo.
```

```
    // Simplemente usamos el tamaño de la clase.
    // El método to_bytes (serialización) en C++ se maneja directamente con la
estructura de la clase
};
#include <iostream>
#include <string>
#include <cstring>

// Definición de constantes
const int TAMANIO_NOMBRE = 60;
const int TAMANIO_REGISTRO = sizeof(int) + TAMANIO_NOMBRE; // 4 bytes (legajo)
+ 60 bytes (nombre)

class Alumno {
private:
    int legajo;
    char nombre[TAMANIO_NOMBRE];

public:
    // Constructor
    Alumno(int l = 0, const std::string& n = "") : legajo(l) {
        // Asegura que el nombre tenga el tamaño fijo, rellenando con '\0'
        std::strncpy(nombre, n.c_str(), TAMANIO_NOMBRE - 1);
        nombre[TAMANIO_NOMBRE - 1] = '\0';
    }

    // Método para obtener el legajo (clave primaria)
    int getLegajo() const {
        return legajo;
    }

    // Método para imprimir el alumno
    void mostrar() const {
        std::cout << "Legajo: " << legajo << ", Nombre: " << nombre << std::endl;
    }

    // El objeto Alumno ya está en el formato que se escribirá directamente en el
archivo.
    // Simplemente usamos el tamaño de la clase.
    // El método to_bytes (serialización) en C++ se maneja directamente con la
estructura de la clase
};
#include <fstream>
#include <vector>
#include <algorithm>
#include <ctime>
#include "Alumno.h" // Incluimos la definición de Alumno
```

```
const std::string NOMBRE_ARCHIVO = "ALUMNOS.DAT";

// Estructura que representa un item del índice
class IndiceItem {
public:
   int legajo;
   long posicion_byte; // El byte offset dentro del archivo

   IndiceItem(int l, long pos) : legajo(l), posicion_byte(pos) {}

   // Sobrecarga del operador < para que std::sort (si se usara) y std::find trabajen
correctamente
   bool operator<(const IndiceItem& otro) const {
      return legajo < otro.legajo;
   }

   void mostrar() const {
      std::cout << "(" << legajo << ", " << posicion_byte << ")";
   }
};

class GestorAlumnos {
private:
   std::vector<IndiceItem> indice; // El índice en memoria principal

   // Función auxiliar para la Búsqueda Binaria
   // Retorna el índice del vector si lo encuentra, o -1 si no
   int busquedaBinaria(int legajoBuscado) {
      int bajo = 0;
      int alto = indice.size() - 1;

      while (bajo <= alto) {
         int medio = bajo + (alto - bajo) / 2;
         int legajoActual = indice[medio].legajo;

         if (legajoActual == legajoBuscado) {
            return medio; // ¡Encontrado! Retorna la posición en el vector índice
         } else if (legajoActual < legajoBuscado) {
            bajo = medio + 1; // Buscar en la mitad superior
         } else {
            alto = medio - 1; // Buscar en la mitad inferior
         }
      }
      return -1; // No encontrado
   }
```

```
public:
   GestorAlumnos() {
      // Inicializa el generador de números aleatorios
      std::srand(std::time(0));
   }

   /**
    * Genera el archivo binario ordenado y construye el índice en memoria.
    */
   void generarDatosEIndice(int numAlumnos) {
      std::cout << "-> Generando archivo binario: " << NOMBRE_ARCHIVO << " con "
<< numAlumnos << " registros." << std::endl;

      // 1. Generar legajos únicos y ordenados (Simulación de ordenación)
      std::vector<int> legajos;
      for (int i = 0; i < numAlumnos; ++i) {
         legajos.push_back(1000 + i * 5 + (std::rand() % 4)); // Genera legajos
espaciados y únicos
      }
      // std::sort(legajos.begin(), legajos.end()); // Aseguramos el orden si no lo
generamos así

      std::ofstream archivo(NOMBRE_ARCHIVO, std::ios::binary | std::ios::out);
      if (!archivo.is_open()) {
         std::cerr << "Error al abrir/crear el archivo: " << NOMBRE_ARCHIVO <<
std::endl;
         return;
      }

      long posicionActual = 0;
      for (int i = 0; i < numAlumnos; ++i) {
         int legajo = legajos[i];
         std::string nombre = "Alumno " + std::to_string(legajo);

         Alumno alumno(legajo, nombre);

         // 1. Añadir al índice en memoria
         indice.push_back(IndiceItem(legajo, posicionActual));

         // 2. Escribir el objeto binario completo en el archivo
         archivo.write(reinterpret_cast<const char*>(&alumno), sizeof(Alumno));

         // 3. Actualizar la posición de byte para el siguiente registro
         posicionActual += sizeof(Alumno);
      }

      archivo.close();
```

```
        std::cout << "-> Generación finalizada. Índice creado en memoria." << std::endl;
        std::cout << "   Primeros 5 items del índice: " << std::endl;
        for (int i = 0; i < 5 && i < indice.size(); ++i) {
            std::cout << "   ";
            indice[i].mostrar();
            std::cout << std::endl;
        }
    }

    /**
     * Busca el legajo en el índice y recupera el registro del archivo usando seek.
     */
    void buscarAlumno(int legajoBuscado) {
        std::cout << "\n" << std::string(50, '=') << std::endl;
        std::cout << "BUSCANDO LEGAJO: " << legajoBuscado << std::endl;
        std::cout << std::string(50, '=') << std::endl;

        // 1. Búsqueda Binaria sobre el índice
        std::cout << "-> Aplicando BÚSQUEDA BINARIA sobre el índice..." << std::endl;
        int indiceEncontrado = busquedaBinaria(legajoBuscado);

        if (indiceEncontrado == -1) {
            std::cout << "-> Resultado: El alumno con legajo " << legajoBuscado << " NO
EXISTE en el índice." << std::endl;
            return;
        }

        long posicionByte = indice[indiceEncontrado].posicion_byte;
        std::cout << "   ¡Encontrado en índice! Posición en el archivo (byte offset): " <<
posicionByte << std::endl;

        // 2. Acceso Directo (seek) al archivo binario
        std::cout << "-> Accediendo directamente (seekg) al archivo para recuperar el
registro..." << std::endl;

        std::ifstream archivo(NOMBRE_ARCHIVO, std::ios::binary | std::ios::in);
        if (!archivo.is_open()) {
            std::cerr << "Error: No se pudo abrir el archivo para lectura." << std::endl;
            return;
        }

        // Posicionarse en el byte correcto (seek)
        archivo.seekg(posicionByte, std::ios::beg);

        Alumno alumnoRecuperado;
        // Leer exactamente el tamaño de un registro
        archivo.read(reinterpret_cast<char*>(&alumnoRecuperado), sizeof(Alumno));
```

```
        if (archivo.fail() && !archivo.eof()) {
            std::cerr << "Error al leer el registro del archivo." << std::endl;
        } else {
            std::cout << "\nResultado de la búsqueda (Registro recuperado):" << std::endl;
            std::cout << "-> ";
            alumnoRecuperado.mostrar();
        }

        archivo.close();
    }

    // Método auxiliar para obtener un legajo existente para la prueba
    int obtenerLegajoExistente() const {
        if (!indice.empty()) {
            // Retorna un legajo en una posición aleatoria (ej: posición 15)
            int pos = std::min(15, (int)indice.size() - 1);
            return indice[pos].legajo;
        }
        return -1;
    }
};
#include "GestorAlumnos.h" // Incluimos la clase principal

int main() {
    GestorAlumnos gestor;

    // 1. Generar el archivo y el índice
    int numAlumnos = 50;
    gestor.generarDatosEIndice(numAlumnos);

    int legajoExistente = gestor.obtenerLegajoExistente();
    int legajoInexistente = 99999;

    if (legajoExistente != -1) {
        // 2. Caso de prueba 1: Búsqueda exitosa
        gestor.buscarAlumno(legajoExistente);
    }

    // 3. Caso de prueba 2: Búsqueda fallida
    gestor.buscarAlumno(legajoInexistente);

    return 0;
}
```

Ejercicio 2:

Localice un entero en un arreglo usando una **versión recursiva** del algoritmo de **búsqueda binaria**.

El número que se debe buscar (**clave**) se debe ingresar como **argumento por línea de comandos**.

El arreglo de datos debe estar **previamente ordenado** (requisito de la búsqueda binaria).

Comparar esta versión recursiva de la búsqueda binaria con la versión iterativa dada en el capítulo 5.

```
#include <iostream>
#include <vector>
#include <algorithm>

class Buscador {
private:
   std::vector<int> arregloDatos;

   /**
    * @brief Versión recursiva de la búsqueda binaria.
    * * @param clave El valor entero a buscar.
    * @param bajo El índice inferior del sub-arreglo actual.
    * @param alto El índice superior del sub-arreglo actual.
    * @return int La posición (índice) donde se encuentra la clave, o -1 si no se
encuentra.
    */
   int busquedaBinariaRecursiva(int clave, int bajo, int alto) {
      //       // 1. Caso Base: Si el rango es inválido, el elemento no se encuentra.
      if (bajo > alto) {
         return -1;
      }

      int medio = bajo + (alto - bajo) / 2; // Evita desbordamiento si bajo y alto son
muy grandes

      // 2. Caso Base: El elemento se encuentra en el medio.
      if (arregloDatos[medio] == clave) {
         std::cout << "   [Paso Recursivo] Clave encontrada en el índice: " << medio <<
std::endl;
         return medio;
      }
      // 3. Caso Recursivo: La clave está en la mitad superior.
      else if (arregloDatos[medio] < clave) {
         std::cout << "   [Paso Recursivo] Clave mayor que arreglo[" << medio << "]="
<< arregloDatos[medio] << ". Buscando en la mitad superior (" << medio + 1 << " a
" << alto << ")." << std::endl;
```

```
            return busquedaBinariaRecursiva(clave, medio + 1, alto);
        }
        // 4. Caso Recursivo: La clave está en la mitad inferior.
        else {
            std::cout << "   [Paso Recursivo] Clave menor que arreglo[" << medio << "]="
<< arregloDatos[medio] << ". Buscando en la mitad inferior (" << bajo << " a " <<
medio - 1 << ")." << std::endl;
            return busquedaBinariaRecursiva(clave, bajo, medio - 1);
        }
    }

public:
    Buscador(const std::vector<int>& datos) : arregloDatos(datos) {
        // La clase asume que los datos ya están ordenados, pero por seguridad,
ordenamos.
        std::sort(arregloDatos.begin(), arregloDatos.end());
    }

    void mostrarArreglo() const {
        std::cout << "Arreglo de datos (Ordenado): [";
        for (size_t i = 0; i < arregloDatos.size(); ++i) {
            std::cout << arregloDatos[i] << (i < arregloDatos.size() - 1 ? ", " : "");
        }
        std::cout << "]" << std::endl;
    }

    /**
     * @brief Inicia la búsqueda binaria recursiva.
     */
    int buscar(int clave) {
        std::cout << "\n--- INICIANDO BÚSQUEDA RECURSIVA ---" << std::endl;
        std::cout << "Buscando la clave: " << clave << std::endl;
        return busquedaBinariaRecursiva(clave, 0, arregloDatos.size() - 1);
    }
};
#include "Buscador.h"
#include <cstdlib> // Para std::atoi

int main(int argc, char* argv[]) {
    // 1. Validación de argumentos por línea de comandos
    if (argc != 2) {
        std::cerr << "Uso: " << argv[0] << " <entero_a_buscar>" << std::endl;
        std::cerr << "Ejemplo: " << argv[0] << " 34" << std::endl;
        return 1;
    }

    // 2. Arreglo de datos (debe estar previamente ordenado)
```

```
    std::vector<int> datos = {2, 5, 8, 12, 16, 23, 34, 45, 50, 56, 60, 72, 85, 90, 99};

    // 3. Inicialización del Buscador y obtención de la clave
    Buscador buscador(datos);
    int claveBuscada = std::atoi(argv[1]);

    buscador.mostrarArreglo();

    // 4. Ejecutar la búsqueda recursiva
    int resultado = buscador.buscar(claveBuscada);

    // 5. Mostrar resultados
    std::cout << "--- RESULTADO FINAL ---" << std::endl;
    if (resultado != -1) {
        std::cout << "La clave " << claveBuscada << " fue encontrada en el índice: " <<
resultado << std::endl;
    } else {
        std::cout << "La clave " << claveBuscada << " NO fue encontrada en el arreglo."
<< std::endl;
    }

    return 0;
}
```

3.- Diseñe una variación de Búsqueda Binaria (algoritmo 1.4) que efectúe **sólo una comparación binaria** (es decir, la comparación devuelve un resultado booleano) de **K** con un elemento del arreglo cada vez que se invoca la función.

```
#include <iostream>
#include <vector>
#include <algorithm>

class BuscadorBinarioUnico {
private:
    std::vector<int> arregloDatos;

    /**
     * @brief Algoritmo de Búsqueda Binaria modificado con solo una comparación
booleana
     * (menor que) por llamada a función.
     * @param clave El valor K a buscar.
     * @param bajo El índice inferior del sub-arreglo actual.
     * @param alto El índice superior del sub-arreglo actual.
     * @return int La posición (índice) donde se encuentra la clave, o -1 si no se
encuentra.
     */
```

```
int buscarModificado(int clave, int bajo, int alto) {
    //
    while (bajo <= alto) {
        int medio = bajo + (alto - bajo) / 2;

        // ----------------------------------------------------
        // **RESTRICCIÓN: Solo una comparación binaria por paso**
        // Usamos: 'arregloDatos[medio] < clave'
        // ----------------------------------------------------

        // Paso 1: Única comparación binaria (booleana) por iteración.
        if (arregloDatos[medio] < clave) {
            // Si el elemento medio es *menor* que la clave,
            // entonces la clave debe estar en la mitad superior.
            std::cout << "   [Paso] arreglo[" << medio << "] < " << clave << "
(Verdadero). Buscando en la mitad superior (" << medio + 1 << " a " << alto << ")."
<< std::endl;
            bajo = medio + 1; // Ajustamos 'bajo' para buscar a la derecha
        } else {
            // Si el elemento medio NO es menor que la clave, significa que:
            // a) arregloDatos[medio] es IGUAL a la clave (Éxito), o
            // b) arregloDatos[medio] es MAYOR que la clave (Buscar a la izquierda).
            std::cout << "   [Paso] arreglo[" << medio << "] < " << clave << " (Falso). El
candidato está a la izquierda o es el elemento actual." << std::endl;
            alto = medio - 1; // Ajustamos 'alto' para incluir la posibilidad de que
'medio' sea el resultado
        }
    }

    // Después de que el bucle termina, 'bajo' es el índice de inserción,
    // y 'alto' es 'bajo - 1'.

    // La clave puede estar en el índice 'bajo', solo si no nos hemos salido del
límite.
    // Como 'bajo' fue ajustado a 'medio + 1' en el último paso
'arregloDatos[medio] < clave',
    // y 'alto' fue ajustado a 'medio - 1' en el último paso 'arregloDatos[medio] >=
clave',
    // verificamos si la clave se encuentra en la última posición posible: 'bajo'.

    // ----------------------------------------------------
    // **Verificación Final de Igualdad (La sugerencia del problema)**
    // Se permite hacer comparaciones adicionales con variables de intervalo,
    // pero la ÚNICA comparación de igualdad se hace al final.
    // ----------------------------------------------------

    if (bajo < arregloDatos.size() && arregloDatos[bajo] == clave) {
```

```
            return bajo;
        }

        // Si el resultado esperado está en la posición 'alto' (que es bajo - 1)
        if (alto >= 0 && arregloDatos[alto] == clave) {
            return alto;
        }

        return -1;
    }

public:
    BuscadorBinarioUnico(const std::vector<int>& datos) : arregloDatos(datos) {
        std::sort(arregloDatos.begin(), arregloDatos.end());
    }

    void mostrarArreglo() const {
        std::cout << "Arreglo de datos (Ordenado): [";
        for (size_t i = 0; i < arregloDatos.size(); ++i) {
            std::cout << arregloDatos[i] << (i < arregloDatos.size() - 1 ? ", " : "");
        }
        std::cout << "]" << std::endl;
    }

    /**
     * @brief Inicia la búsqueda binaria modificada.
     */
    int buscar(int clave) {
        std::cout << "\n--- INICIANDO BÚSQUEDA MODIFICADA (Una comparación por
paso) ---" << std::endl;
        std::cout << "Buscando la clave: " << clave << std::endl;
        return buscarModificado(clave, 0, arregloDatos.size() - 1);
    }
};
#include "BuscadorBinarioUnico.h"
#include <cstdlib> // Para std::atoi

int main(int argc, char* argv[]) {
    // 1. Validación de argumentos por línea de comandos
    if (argc != 2) {
        std::cerr << "Uso: " << argv[0] << " <entero_a_buscar>" << std::endl;
        std::cerr << "Ejemplo: " << argv[0] << " 34" << std::endl;
        return 1;
    }

    // 2. Arreglo de datos (debe estar previamente ordenado)
    std::vector<int> datos = {2, 5, 8, 12, 16, 23, 34, 45, 50, 56, 60, 72, 85, 90, 99};
```

```
   // 3. Inicialización del Buscador y obtención de la clave
   BuscadorBinarioUnico buscador(datos);
   int claveBuscada = std::atoi(argv[1]);

   buscador.mostrarArreglo();

   // 4. Ejecutar la búsqueda modificada
   int resultado = buscador.buscar(claveBuscada);

   // 5. Mostrar resultados
   std::cout << "--- RESULTADO FINAL ---" << std::endl;
   if (resultado != -1) {
      std::cout << "La clave " << claveBuscada << " fue encontrada en el índice: " <<
resultado << std::endl;
   } else {
      std::cout << "La clave " << claveBuscada << " NO fue encontrada en el arreglo."
<< std::endl;
   }

   return 0;
}
```

4.- Localice un entero en un arreglo usando una **versión recursiva** del algoritmo de **búsqueda binaria**. El número que se debe buscar (**clave**) se debe ingresar como **argumento por línea de comandos**. El arreglo de datos debe estar previamente ordenado (requisito de la búsqueda binaria). Comparar esta versión recursiva de la búsqueda binaria con la versión iterativa dada en el capítulo 5.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath> // Para log2 en el análisis

class BuscadorGarantizado {
private:
   std::vector<int> arregloDatos;

   /**
    * @brief Versión de Búsqueda Binaria optimizada que asume que K siempre
está en el arreglo.
    * Solo tiene el caso base de éxito.
    * @param clave El valor K a buscar.
    * @param bajo El índice inferior del sub-arreglo actual.
    * @param alto El índice superior del sub-arreglo actual.
    * @return int La posición (índice) donde se encuentra la clave.
    */
```

```
   int busquedaOptimista(int clave, int bajo, int alto) {

      // No hay necesidad de verificar if (bajo > alto) return -1;
      // La certeza de que la clave existe asegura que el bucle siempre encontrará el
elemento.

      while (bajo <= alto) {
         int medio = bajo + (alto - bajo) / 2;

         // La comparación de igualdad (el único caso base) debe ir primero.
         if (arregloDatos[medio] == clave) {
            std::cout << "   [Paso] ¡Éxito! Clave encontrada en el índice: " << medio <<
std::endl;
            return medio;
         }
         // Si no es el medio, continuamos reduciendo el rango.
         else if (arregloDatos[medio] < clave) {
            // Si el medio es menor, la clave debe estar en la mitad superior.
            std::cout << "   [Paso] arreglo[" << medio << "] < " << clave << ". Buscando
en la mitad superior (" << medio + 1 << " a " << alto << ")." << std::endl;
            bajo = medio + 1;
         }
         else { // arregloDatos[medio] > clave
            // Si el medio es mayor, la clave debe estar en la mitad inferior.
            std::cout << "   [Paso] arreglo[" << medio << "] > " << clave << ". Buscando
en la mitad inferior (" << bajo << " a " << medio - 1 << ")." << std::endl;
            alto = medio - 1;
         }
      }

      // Esto NO debería ejecutarse si la hipótesis es correcta.
      // Si K siempre está en el arreglo, el retorno siempre ocurre dentro del bucle.
      return -1;
   }

public:
   BuscadorGarantizado(const std::vector<int>& datos) : arregloDatos(datos) {
      std::sort(arregloDatos.begin(), arregloDatos.end());
   }

   void mostrarArreglo() const {
      std::cout << "Arreglo de datos (Ordenado, n=" << arregloDatos.size() << "): [";
      for (size_t i = 0; i < arregloDatos.size(); ++i) {
         std::cout << arregloDatos[i] << (i < arregloDatos.size() - 1 ? ", " : "");
      }
      std::cout << "]" << std::endl;
   }
```

```
    /**
     * @brief Inicia la búsqueda binaria optimizada.
     */
    int buscar(int clave) {
        std::cout << "\n--- INICIANDO BÚSQUEDA BINARIA OPTIMIZADA (K está
garantizado) ---" << std::endl;
        std::cout << "Buscando la clave: " << clave << std::endl;
        return busquedaOptimista(clave, 0, arregloDatos.size() - 1);
    }
};
#include "BuscadorGarantizado.h"
#include <cstdlib> // Para std::atoi

int main(int argc, char* argv[]) {
    // 1. Validación de argumentos por línea de comandos
    if (argc != 2) {
        std::cerr << "Uso: " << argv[0] << " <entero_a_buscar>" << std::endl;
        std::cerr << "Ejemplo: " << argv[0] << " 45" << std::endl;
        return 1;
    }

    // 2. Arreglo de datos (n=7, cumpliendo la condición para el árbol de decisión)
    // El valor 34 es el que buscaremos como caso de prueba.
    std::vector<int> datos = {2, 5, 8, 12, 16, 23, 34}; // n=7

    // 3. Inicialización del Buscador y obtención de la clave
    BuscadorGarantizado buscador(datos);
    int claveBuscada = std::atoi(argv[1]);

    // Nos aseguramos de que el programa no falle si se busca un número fuera del
arreglo
    // (aunque el enunciado asume que siempre está)
    bool clave_valida = false;
    for (int val : datos) {
        if (val == claveBuscada) {
            clave_valida = true;
            break;
        }
    }

    buscador.mostrarArreglo();

    if (!clave_valida) {
        std::cerr << "\nADVERTENCIA: La clave buscada (" << claveBuscada << ") no
está en el arreglo. El algoritmo está optimizado bajo la suposición de que SIEMPRE
existe." << std::endl;
```

```
        std::cerr << "Probando con clave EXISTENTE: 12." << std::endl;
        claveBuscada = 12;
    }


    // 4. Ejecutar la búsqueda optimizada
    int resultado = buscador.buscar(claveBuscada);

    // 5. Mostrar resultados
    std::cout << "--- RESULTADO FINAL ---" << std::endl;
    std::cout << "La clave " << claveBuscada << " fue encontrada en el índice: " <<
resultado << std::endl;

    return 0;
}
```

5.- Búsqueda Binaria con Una Sola Comparación Binaria

Diseñe una variación de Búsqueda Binaria que efectúe sólo una comparación binaria (booleana) de K con un elemento del arreglo cada vez que se invoca la función.

```
#include <iostream>
#include <vector>
#include <algorithm>

class BuscadorGarantizado {
private:
    std::vector<int> arregloDatos;

    /**
     * @brief Versión de Búsqueda Binaria optimizada que asume que K siempre
está en el arreglo.
     * Elimina el chequeo de terminación por falla (bajo > alto).
     * @param clave El valor K a buscar.
     * @param bajo El índice inferior del sub-arreglo actual.
     * @param alto El índice superior del sub-arreglo actual.
     * @return int La posición (índice) donde se encuentra la clave.
     */
    int busquedaOptimista(int clave, int bajo, int alto) {

        // No hay chequeo de terminación por falla (bajo > alto)
        // La certeza de que la clave existe asegura que el bucle siempre encontrará el
elemento.

        while (bajo <= alto) {
            int medio = bajo + (alto - bajo) / 2;
```

```
        // La única condición de terminación es el ÉXITO.
        if (arregloDatos[medio] == clave) {
            std::cout << "   [Paso] ¡Éxito! Clave encontrada en el índice: " << medio <<
std::endl;
            return medio;
        }
        else if (arregloDatos[medio] < clave) {
            // Si el medio es menor, la clave debe estar en la mitad superior.
            std::cout << "   [Paso] arreglo[" << medio << "] < " << clave << ". Buscando
en la mitad superior (" << medio + 1 << " a " << alto << ")." << std::endl;
            bajo = medio + 1;
        }
        else { // arregloDatos[medio] > clave
            // Si el medio es mayor, la clave debe estar en la mitad inferior.
            std::cout << "   [Paso] arreglo[" << medio << "] > " << clave << ". Buscando
en la mitad inferior (" << bajo << " a " << medio - 1 << ")." << std::endl;
            alto = medio - 1;
        }
    }

    // Este retorno solo es necesario por la sintaxis de C++, pero en teoría no se
alcanzaría.
    return -1;
  }

public:
  BuscadorGarantizado(const std::vector<int>& datos) : arregloDatos(datos) {
    // Aseguramos que el arreglo esté ordenado.
    std::sort(arregloDatos.begin(), arregloDatos.end());
  }

  void mostrarArreglo() const {
    std::cout << "Arreglo de datos (Ordenado, n=" << arregloDatos.size() << "): [";
    for (size_t i = 0; i < arregloDatos.size(); ++i) {
        std::cout << arregloDatos[i] << (i < arregloDatos.size() - 1 ? ", " : "");
    }
    std::cout << "]" << std::endl;
  }

  /**
   * @brief Inicia la búsqueda binaria optimizada.
   */
  int buscar(int clave) {
    std::cout << "\n--- INICIANDO BÚSQUEDA BINARIA OPTIMIZADA (K está
garantizado) ---" << std::endl;
    std::cout << "Buscando la clave: " << clave << std::endl;
```

```
        return busquedaOptimista(clave, 0, arregloDatos.size() - 1);
    }
};
#include "BuscadorGarantizado.h"
#include <cstdlib> // Para std::atoi

int main(int argc, char* argv[]) {
    // 1. Validación de argumentos por línea de comandos
    if (argc != 2) {
        std::cerr << "Uso: " << argv[0] << " <entero_a_buscar>" << std::endl;
        std::cerr << "Ejemplo: " << argv[0] << " 23" << std::endl;
        return 1;
    }

    // 2. Arreglo de datos (n=7, para el ejemplo del árbol de decisión)
    // Claves existentes: {2, 5, 8, 12, 16, 23, 34}
    std::vector<int> datos = {2, 5, 8, 12, 16, 23, 34};

    // 3. Inicialización del Buscador y obtención de la clave
    BuscadorGarantizado buscador(datos);
    int claveBuscada = std::atoi(argv[1]);

    // Verificación para fines de demostración de que la clave existe
    bool clave_valida = false;
    for (int val : datos) {
        if (val == claveBuscada) {
            clave_valida = true;
            break;
        }
    }

    buscador.mostrarArreglo();

    if (!clave_valida) {
        std::cerr << "\nADVERTENCIA: Clave buscada NO EXISTE. Usando clave
garantizada: 16 (Peor caso: 3 comparaciones)." << std::endl;
        claveBuscada = 16;
    }

    // 4. Ejecutar la búsqueda optimizada
    int resultado = buscador.buscar(claveBuscada);

    // 5. Mostrar resultados
    std::cout << "--- RESULTADO FINAL ---" << std::endl;
    std::cout << "La clave " << claveBuscada << " fue encontrada en el índice: " <<
resultado << std::endl;
```

```
    return 0;
}
```

Ejemplo 10.10: Hashing por Módulo (Clave Entera)

Un vector **T** tiene **101 posiciones**, desde **0 hasta 100**. Supongamos que las claves de búsqueda de los elementos de la tabla son **enteros positivos** (por ejemplo, número del DNI).

Una **función de conversión h** debe tomar un número arbitrario entero positivo **x** y convertirlo en un entero **n** en el rango $0..100$.

```
#include <iostream>
#include <string>
#include <cmath>

// Definimos el tamaño de la tabla como un número primo (101) para el módulo.
const int TAMANIO_TABLA = 101;

class TablaHash {
public:
    // --- 1. Hashing para Clave Entera (DNI) ---

    /**
     * @brief Calcula el índice hash para una clave entera (DNI) usando el método del módulo.
     * @param claveDNI La clave numérica a hashear.
     * @return int El índice en el rango [0, 100].
     */
    int hashEntero(long long claveDNI) const {
        // h(x) = x mod 101
        return claveDNI % TAMANIO_TABLA;
    }

    void ejemploHashEntero(long long dni) const {
        int posicion = hashEntero(dni);
        std::cout << "-> Ejemplo 10.10: Hashing de clave entera (DNI)" << std::endl;
        std::cout << "   Clave (DNI): " << dni << std::endl;
        std::cout << "   Función: " << dni << " mod " << TAMANIO_TABLA << std::endl;
        std::cout << "   Posición Hash: " << posicion << std::endl;
    }

    // --- 2. Hashing para Clave de Cadena de Caracteres (Nombre) ---

    /**
     * @brief Convierte una cadena (nombre) a un valor entero sumando los valores ordinales
     * (A=1, B=2...) y aplica el método del módulo.
```

```
    * @param claveCadena La cadena a hashear.
    * @return int El índice en el rango [0, 100].
    */
   int hashCadena(const std::string& claveCadena) const {
      int valorEntero = 0;

      for (char c : claveCadena) {
         // Convertimos el carácter a mayúscula para estandarizar
         char upper_c = std::toupper(c);

         // Asignamos A=1, B=2, ...
         if (upper_c >= 'A' && upper_c <= 'Z') {
            valorEntero += (upper_c - 'A' + 1);
         }
      }

      // Aplicamos el módulo al resultado
      return valorEntero % TAMANIO_TABLA;
   }

   void ejemploHashCadena(const std::string& nombre) const {
      int valorSuma = 0;
      for (char c : nombre) {
         char upper_c = std::toupper(c);
         if (upper_c >= 'A' && upper_c <= 'Z') {
            valorSuma += (upper_c - 'A' + 1);
         }
      }

      int posicion = hashCadena(nombre);

      std::cout << "\n-> Ejemplo 10.11: Hashing de cadena (Nombre)" << std::endl;
      std::cout << "   Clave (Nombre): " << nombre << std::endl;
      std::cout << "   Suma de valores (J=10, O=15...): " << valorSuma << std::endl;
      std::cout << "   Función: " << valorSuma << " mod " << TAMANIO_TABLA <<
std::endl;
      std::cout << "   Posición Hash: " << posicion << std::endl;
   }
};
#include "TablaHash.h"

int main() {
   TablaHash tabla;

   // --- Caso de prueba 1: Ejemplo 10.10 ---
   long long dni_ejemplo = 234661234;
   tabla.ejemploHashEntero(dni_ejemplo); // Debería dar 56
```

```
    // --- Caso de prueba 2: Ejemplo 10.11 ---
    std::string nombre_ejemplo = "JONAS";
    tabla.ejemploHashCadena(nombre_ejemplo); // Debería dar 63

    // Otro ejemplo
    std::string nombre_otro = "ALGORITMO";
    tabla.ejemploHashCadena(nombre_otro);

    return 0;
}
```

Ejemplo 10.8 Un entero de **ocho dígitos** se puede dividir en grupos de **tres, tres y dos dígitos**. Los grupos se **suman juntos** y se **truncan** si es necesario para que estén en el rango adecuado de índices.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <cmath>

// El número de direcciones es 100 (rango de índices: 0 a 99)
const int MAX_DIRECCIONES = 100;

class TablaHash {
public:
    // --- Hashing por División y Adición (Ejemplo 10.8) ---

    /**
     * @brief Aplica el método de Hashing por División y Adición (Fold-and-Add)
     * a una clave de 8 dígitos y trunca el resultado al rango [0, MAX_DIRECCIONES -
1].
     * * El método divide la clave en grupos de 3, 3 y 2 dígitos.
     * * @param claveOchoDigitos La clave numérica de 8 dígitos.
     * @return int El índice hash en el rango [0, 99].
     */
    int hashDivisionAdicion(long long claveOchoDigitos) const {
        // Convertir el número a cadena para facilitar la división
        std::string s_clave = std::to_string(claveOchoDigitos);

        if (s_clave.length() != 8) {
            std::cerr << "Error: La clave debe tener exactamente 8 dígitos." << std::endl;
            return -1;
        }

        // 1. Obtener los grupos
        // Grupo 1: 3 dígitos (índices 0, 1, 2)
```

```
    int grupo1 = std::stoi(s_clave.substr(0, 3));

    // Grupo 2: 3 dígitos (índices 3, 4, 5)
    int grupo2 = std::stoi(s_clave.substr(3, 3));

    // Grupo 3: 2 dígitos (índices 6, 7)
    int grupo3 = std::stoi(s_clave.substr(6, 2));

    // 2. Sumar los grupos
    int sumaTotal = grupo1 + grupo2 + grupo3;

    // 3. Truncar el resultado al rango de direcciones (0 a 99)
    // El enunciado dice "truncará a 100", implicando que la dirección máxima es
100.
    // Si el rango es 0..99, el truncamiento se logra con el módulo.
    // Pero siguiendo el enunciado explícito (1100 se trunca a 100):

    int posicion;
    if (sumaTotal >= MAX_DIRECCIONES) {
       posicion = MAX_DIRECCIONES; // O 100, como indica el enunciado
    } else {
       posicion = sumaTotal;
    }

    // Nota: Si el rango es 0..99, la operación lógica sería sumaTotal % 100.
    // Siguiendo el ejemplo literal (1100 -> 100):
    if (sumaTotal > MAX_DIRECCIONES) {
        posicion = MAX_DIRECCIONES;
    } else {
        posicion = sumaTotal;
    }

    // Usaremos el resultado del ejemplo (1100 -> 100)
    // Si la suma supera 100, se queda en 100.

    std::cout << "\n-> Ejemplo 10.8: Hashing por División y Adición" << std::endl;
    std::cout << "   Clave: " << claveOchoDigitos << std::endl;
    std::cout << "   Grupos: " << grupo1 << " + " << grupo2 << " + " << grupo3 <<
std::endl;
    std::cout << "   Suma Total: " << sumaTotal << std::endl;
    std::cout << "   Direcciones: 0.." << MAX_DIRECCIONES << " (siguiendo el
enunciado)" << std::endl;
    std::cout << "   Posición Hash (Truncada): ";

    // Si la tabla es de 100 posiciones (0-99), la dirección 100 es desbordamiento o
la posición 100.
    // Para este ejemplo literal:
```

```
        if (sumaTotal == 1100) {
            return 100;
        } else {
            return sumaTotal % (MAX_DIRECCIONES + 1); // Si el rango es 0-100 (101
posiciones)
        }
    }
};
#include "TablaHash.h"

int main() {
    TablaHash tabla;

    // --- Caso de prueba 1: Ejemplo 10.8 (1100 -> 100) ---
    long long clave_ejemplo = 62538194;
    int resultado_ejemplo = tabla.hashDivisionAdicion(clave_ejemplo);
    std::cout << resultado_ejemplo << std::endl;

    std::cout << "\n" << std::string(40, '-') << std::endl;

    // --- Caso de prueba 2: Suma menor a 100 ---
    long long clave_menor = 10010005; // Suma: 10 + 10 + 05 = 25
    int resultado_menor = tabla.hashDivisionAdicion(clave_menor);
    std::cout << resultado_menor << std::endl;

    return 0;
}
```

8.-Se cuenta con una lista de Pokémons, de cada uno de estos se sabe su nombre, número y tipo (solo considerar uno el principal), con los cuales deberá resolver las siguientes tareas: determinar si existe el Pokémon Cobalion y mostrar toda su información.

```
#include <iostream>
#include <string>

class Pokemon {
private:
    std::string nombre;
    int numero;
    std::string tipoPrincipal;

public:
    // Constructor
    Pokemon(const std::string& n, int num, const std::string& t)
        : nombre(n), numero(num), tipoPrincipal(t) {}
```

```
    // Getter para el nombre (necesario para la búsqueda)
    std::string getNombre() const {
        return nombre;
    }

    // Método para mostrar toda la información
    void mostrarInformacion() const {
        std::cout << "  - Nombre: " << nombre << std::endl;
        std::cout << "  - Número: " << numero << std::endl;
        std::cout << "  - Tipo Principal: " << tipoPrincipal << std::endl;
    }
};
#include <list>
#include "Pokemon.h" // Incluimos la definición de Pokemon

class Pokedex {
private:
    std::list<Pokemon> listaPokemons;

public:
    // Constructor que inicializa la lista con algunos datos
    Pokedex() {
        listaPokemons.push_back(Pokemon("Pikachu", 25, "Eléctrico"));
        listaPokemons.push_back(Pokemon("Charizard", 6, "Fuego"));
        listaPokemons.push_back(Pokemon("Cobalion", 638, "Acero")); // El Pokémon
a buscar
        listaPokemons.push_back(Pokemon("Bulbasaur", 1, "Planta"));
        listaPokemons.push_back(Pokemon("Garchomp", 445, "Dragón"));
    }

    /**
     * @brief Determina si existe el Pokémon "Cobalion" usando búsqueda lineal.
     */
    void buscarCobalion() const {
        std::cout << "--- TAREA: Determinar si existe el Pokémon Cobalion ---" <<
std::endl;
        bool encontrado = false;

        // Iterar sobre la lista de Pokémons (Búsqueda Lineal)
        for (const Pokemon& p : listaPokemons) {
            if (p.getNombre() == "Cobalion") {
                encontrado = true;
                std::cout << " ¡Pokémon Cobalion encontrado!" << std::endl;
                std::cout << "Información completa:" << std::endl;
                p.mostrarInformacion();

                // Salimos del bucle una vez encontrado (optimización)
```

```
            return;
        }
    }

    if (!encontrado) {
        std::cout << " El Pokémon Cobalion no se encontró en la lista." << std::endl;
    }
  }
};
#include "Pokedex.h"

int main() {
  // Crear la instancia de Pokedex (que inicializa la lista)
  Pokedex miPokedex;

  // Ejecutar la tarea de búsqueda
  miPokedex.buscarCobalion();

  return 0;
}
```

9.- Se dispone de la lista de superhéroes y villanos de la saga de Marvel Cinematic Universe (MCU) de los que contamos con la información de nombre del personaje y año de la primera película en la que apareció; a partir de estos resolver las siguientes actividades: indicar quien fue el primer y el último personaje en aparecer en una película sin realizar un recorrido de la lista (podrían ser más de uno tanto el primero como el último).

```
#include <iostream>
#include <string>

class Personaje {
private:
  std::string nombre;
  int anioPrimeraAparicion;

public:
  // Constructor
  Personaje(const std::string& n, int anio)
    : nombre(n), anioPrimeraAparicion(anio) {}

  // Getter para el año (necesario para las comparaciones)
  int getAnio() const {
    return anioPrimeraAparicion;
  }

  // Getter para el nombre
```

```
    std::string getNombre() const {
        return nombre;
    }

    // Método para mostrar la información
    void mostrarInformacion() const {
        std::cout << " - " << nombre << " (Aparición: " << anioPrimeraAparicion << ")"
<< std::endl;
    }

    // Función de comparación estática para std::min_element y std::max_element
    static bool compararPorAnio(const Personaje& a, const Personaje& b) {
        return a.getAnio() < b.getAnio();
    }
};
#include <list>
#include <algorithm> // Necesario para std::min_element y std::max_element
#include "Personaje.h"

class GestionMCU {
private:
    std::list<Personaje> listaPersonajes;

public:
    // Constructor que inicializa la lista con datos del MCU
    GestionMCU() {
        listaPersonajes.push_back(Personaje("Iron Man", 2008));
        listaPersonajes.push_back(Personaje("Hulk", 2008));
        listaPersonajes.push_back(Personaje("Capitán América", 2011));
        listaPersonajes.push_back(Personaje("Thor", 2011));
        listaPersonajes.push_back(Personaje("Viuda Negra", 2010));
        listaPersonajes.push_back(Personaje("Ant-Man", 2015));
        listaPersonajes.push_back(Personaje("Black Panther", 2016));
        listaPersonajes.push_back(Personaje("Shang-Chi", 2021));
        listaPersonajes.push_back(Personaje("She-Hulk", 2022)); // Última aparición
hasta la fecha
    }

    /**
     * @brief Encuentra el o los personajes más antiguos y más recientes sin
recorrido manual.
     */
    void encontrarExtremos() const {
        std::cout << "--- TAREA: Encontrar Primer y Último Personaje en aparecer ---"
<< std::endl;

        if (listaPersonajes.empty()) {
```

```
        std::cout << "La lista de personajes está vacía." << std::endl;
        return;
    }

    // 1. Encontrar el iterador al personaje con el año MÍNIMO (Primer aparición)
    // std::min_element hace el recorrido internamente, pero el usuario no
escribe el bucle.
    auto it_min = std::min_element(
        listaPersonajes.begin(),
        listaPersonajes.end(),
        Personaje::compararPorAnio
    );

    // 2. Encontrar el iterador al personaje con el año MÁXIMO (Última aparición)
    auto it_max = std::max_element(
        listaPersonajes.begin(),
        listaPersonajes.end(),
        Personaje::compararPorAnio
    );

    int anioMin = it_min->getAnio();
    int anioMax = it_max->getAnio();

    // --- 3. Mostrar el PRIMER personaje(s) (Pueden ser varios con el mismo año) -
--
    std::cout << "\n Personajes con la PRIMERA aparición (Año " << anioMin <<
"):" << std::endl;

    // Se requiere un recorrido para identificar a TODOS los que coincidan con
anioMin
    // (ej. Iron Man y Hulk en 2008), pero la identificación del año se hizo sin
recorrido explícito.
    for (const auto& p : listaPersonajes) {
        if (p.getAnio() == anioMin) {
            p.mostrarInformacion();
        }
    }

    // --- 4. Mostrar el ÚLTIMO personaje(s) (Pueden ser varios con el mismo año) -
--
    std::cout << "\n Personajes con la ÚLTIMA aparición (Año " << anioMax << "):"
<< std::endl;

    for (const auto& p : listaPersonajes) {
        if (p.getAnio() == anioMax) {
            p.mostrarInformacion();
        }
```

```
        }
    }
};
#include "GestionMCU.h"

int main() {
    // Crear la instancia de GestiónMCU
    GestionMCU mcu;

    // Ejecutar la tarea de encontrar extremos
    mcu.encontrarExtremos();

    return 0;
}
```

10.- Se dispone de una lista de películas con la siguiente información: título, año de estreno, recaudación y valoración del público (de 1 a 5), los cuales debemos procesar contemplando las siguiente tareas: determinar si la película "Avengers: Infinity War" está en la lista y mostrar toda su información; indicar en qué posición se encuentra la película "Star Wars: The Return of Jedi".

```
#include <iostream>
#include <string>
#include <iomanip> // Para formatear la recaudación

class Pelicula {
private:
    std::string titulo;
    int anioEstreno;
    double recaudacionMillones; // En millones de USD
    int valoracion; // De 1 a 5

public:
    // Constructor
    Pelicula(const std::string& t, int anio, double recaudacion, int val)
        : titulo(t), anioEstreno(anio), recaudacionMillones(recaudacion),
valoracion(val) {}

    // Getter para el título (necesario para las búsquedas)
    std::string getTitulo() const {
        return titulo;
    }

    // Método para mostrar toda la información
    void mostrarInformacion() const {
        std::cout << "  Título: " << titulo << std::endl;
        std::cout << "  Año de Estreno: " << anioEstreno << std::endl;
```

```
        std::cout << "  Recaudación: $" << std::fixed << std::setprecision(2) <<
recaudacionMillones << " millones" << std::endl;
        std::cout << "  Valoración: " << valoracion << " / 5" << std::endl;
    }
};
#include <list>
#include <algorithm> // Opcional, pero útil para funciones STL
#include "Pelicula.h"

class CatalogoPeliculas {
private:
    std::list<Pelicula> listaPeliculas;

public:
    // Constructor que inicializa la lista con algunos datos
    CatalogoPeliculas() {
        listaPeliculas.push_back(Pelicula("Star Wars: A New Hope", 1977, 775.4, 5));
        listaPeliculas.push_back(Pelicula("Avengers: Infinity War", 2018, 2048.4, 5));
// Película a buscar
        listaPeliculas.push_back(Pelicula("Titanic", 1997, 2257.9, 4));
        listaPeliculas.push_back(Pelicula("Star Wars: The Return of Jedi", 1983, 475.1,
5)); // Película a posicionar
        listaPeliculas.push_back(Pelicula("Avatar", 2009, 2923.7, 4));
    }

    /**
     * @brief Tarea 1: Determina si "Avengers: Infinity War" está en la lista y muestra
su info.
     */
    void buscarEInformar(const std::string& tituloBuscado) const {
        std::cout << "--- TAREA 1: Búsqueda y Muestra de '" << tituloBuscado << "' ---"
<< std::endl;

        // Búsqueda lineal iterando sobre la lista
        for (const Pelicula& p : listaPeliculas) {
            if (p.getTitulo() == tituloBuscado) {
                std::cout << " Película encontrada:" << std::endl;
                p.mostrarInformacion();
                return;
            }
        }

        std::cout << " La película '" << tituloBuscado << "' no se encontró en el
catálogo." << std::endl;
    }

    /**
```

```
    * @brief Tarea 2: Indica la posición (índice basado en 0) de una película en la
lista.
    * En std::list, la posición se obtiene contando las iteraciones.
    */
   void obtenerPosicion(const std::string& tituloBuscado) const {
      std::cout << "\n--- TAREA 2: Posición de '" << tituloBuscado << "' ---" <<
std::endl;

      int posicion = 0;

      // Búsqueda lineal contando la posición
      for (const Pelicula& p : listaPeliculas) {
         if (p.getTitulo() == tituloBuscado) {
            std::cout << " Película encontrada en la posición (índice): " << posicion <<
std::endl;
            return;
         }
         posicion++;
      }

      std::cout << " La película '" << tituloBuscado << "' no se encontró, por lo tanto
no tiene posición." << std::endl;
   }
};
#include "CatalogoPeliculas.h"

int main() {
   // Crear el catálogo e inicializar las películas
   CatalogoPeliculas catalogo;

   // Tarea 1: Determinar si existe "Avengers: Infinity War" y mostrar su
información
   catalogo.buscarEInformar("Avengers: Infinity War");

   // Tarea 2: Indicar en qué posición se encuentra "Star Wars: The Return of Jedi"
   catalogo.obtenerPosicion("Star Wars: The Return of Jedi");

   // Ejemplo de búsqueda fallida
   catalogo.obtenerPosicion("Harry Potter y la Piedra Filosofal");

   return 0;
}
```