Algoritmos y estructura de datos

Asignatura anual, código 082021

Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional FRBA

Dr. **Oscar Ricardo Bruno**,
Ing. **Pablo Augusto Sznajdleder**.
Ing. **Jose Maria Sola**

Tabla de contenido

Conceptos básicos

1

Objetivos de aprendizaje
Dominando los temas del presente capitulo Usted podrá.

1. Conocer la terminología propia de la disciplina.
2. Definir y comprender claramente conceptos específicos muchas veces mal definidos
3. Comprender el valor de la abstracción.
4. Dar valor a la eficiencia en las soluciones
5. Introducirse en la notación algorítmica y a la forma e encarar los problemas de programación

Introducción:

Se presenta el alcance del presente trabajo y se introducen conceptos fundamentales de algoritmia y programación, los que servirán de base para el desarrollo de los temas a tratar.

Informática

Disciplina del estudio sistematizado de los procesos algorítmicos que describen y transforman información, su teoría, análisis, diseño, eficiencia, implementación y aplicación.
La informática es una disciplina científica, matemática y una ingeniería; tiene tres formas de pensar propias: Teoría, abstracción y diseño.
Las tres se complementan para la resolución de la mayoría de los problemas.

- ✓ Teoría: Con el pensamiento teórico se describen y prueban relaciones.
- ✓ Abstracción: Recolección de datos y formulación de un modelo, se eliminan los detalles irrelevantes.
- ✓ Diseño: se tienen en cuenta requisitos, especificaciones y se diseñan o analizan mecanismos para resolver problemas. Supone llevar a la práctica los resultados teóricos.

Programación

La programación es una actividad transversal asociada a cualquier área de la informática, aunque es la ingeniería del software el área específica que se ocupa de la creación del software.
En principio la programación se veía como un arte, solo era cuestión de dominar un lenguaje de programación y aplicar habilidades personales a la resolución de problemas, casi en forma artesanal. El software era visto como algo desarrollado a través de la intuición sin la utilización de métodos de diseño con técnicas para proceder en forma sistemática y sin ningún control de su desarrollo. Con el reconocimiento de la complejidad del desarrollo del software nació la ingeniería del software.

Se considero que al igual que cualquier otra disciplina la creación de software debía ser reconocida como una actividad de ingeniería que requería la aplicación de sólidos principios científicos.
La ingeniería del software es la disciplina que se ocupa de la aplicación del conocimiento científico al diseño y construcción de programas de computación y a todas las actividades asociadas de documentación, operación y mantenimiento, lo que proporciona un enfoque sistemático.
La programación es una actividad en la que la creatividad juega un rol primordial
Programa:
Conjunto de instrucciones, ejecutables sobre una computadora, que permite cumplir una función especifica. Se asocia al programa con una determinada función o requerimiento a satisfacer por la ejecución del conjunto de instrucciones que lo forman. En general alcanzan su objetivo en tiempo finito, aunque hay excepciones, por ejemplo los programas de control de un sistema de alarma poseen requerimiento de tiempo infinito. Un programa sin errores que se ejecuta puede no ser correcto si no cumple con los requerimientos.

Definición
Programa: conjunto de instrucciones no activas almacenadas en un computador, se vuelve **tarea** a partir de que se selecciona para su ejecución y permite cumplir una función específica. Un **proceso** es un programa en ejecución.

En principio las tareas más importantes a la que se enfrenta quien debe escribir programas en computadoras son:

1. Definir el conjunto de instrucciones cuya ejecución ordenada conduce a la solución.
2. Elegir la representación adecuada de los datos del problema.

La función esencial del especialista informático es explotar el potencial de las computadoras para resolver situaciones del mundo real. Para esto debe analizar los problemas del mundo real, ser capaz de sintetizar sus aspectos principales y poder especificar la función objetivo que se desee. Posteriormente debe expresar la solución en forma de programa, manejando los datos del mundo real mediante una representación valida para una computadora.

Partes de un programa

Los componentes básicos son las instrucciones y los datos. Las instrucciones o sentencias representan las operaciones que se ejecutaran al interpretar el programa. Todos los lenguajes de programación tienen un conjunto mínimo de operaciones que son las de asignación, selección e iteración. Un lenguaje con solo estas tres instrucciones permite escribir cualquier algoritmo.
Los datos son valores de información de los que se necesita disponer, en ocasiones transformar para ejecutar la función del programa.
Los datos están representados simbólicamente por un nombre que se asocia con una dirección única de memoria.
El contenido de la dirección de memoria correspondiente a un dato constante se asigna solo una vez y solo puede ser modificado en una nueva compilación. En cambio el contenido o valor de la dirección de memoria correspondiente a un dato variable puede ser asignado o modificado en tiempo de ejecución.
Un programa se corresponde con una transformación de datos. A partir de un contexto determinado por las precondiciones.
El programa transforma la información debiendo llegar al resultado esperado produciendo el nuevo contexto caracterizado por las poscondiciones.

Dato

Representación de un objeto el mundo real mediante el cual se pueden modelizar aspectos de un problema que se desea resolver con un programa en una computadora.

Definición

Dato representación de un objeto el mundo real mediante el cual se pueden modelizar aspectos de un problema que se desea resolver con un programa en una computadora.
<dato> -> <objeto><atributo><valor>

Abstracción

Proceso de análisis del mundo real con el propósito de interpretar los aspectos esenciales de un problema y expresarlo en términos precisos.

Modelizacion

Abstraer un problema del mundo real y simplificar su expresión, tratando de encontrar los aspectos principales que se pueden resolver, requerimientos, los datos que se han de procesar y el contexto del problema.

Precondición

Información conocida como verdadera antes de iniciar el programa.

Poscondición

Información que debiera ser verdadera al cumplir un programa, si se cumple adecuadamente el requerimiento pedido.

Especificación

Proceso de analizar problemas del mundo real y determinar en forma clara y concreta el objetivo que se desea. Especificar un problema significa establecer en forma univoca el contexto, las precondiciones el resultado esperado, del cual se derivan las poscondiciones.

Lenguaje de programación

Conjunto de instrucciones permitidas y definidas por sus reglas sintácticas y su valor semántico para la expresión de soluciones de problemas.

Del problema real a su solución por computadoras

Analizando un problema del mundo real se llega a la *modelización* del problema por medio de la *abstracción*.

A partir del modelo se debe elaborar el análisis de la solución como sistema, esto significa la descomposición en módulos. Estos módulos deben tener una función bien definida.

La modularización es muy importante y no solo se refiere a los procesos a cumplir, sino también a la distribución de los datos de entrada, salida y los datos intermedios necesarios para alcanzar la solución.

Estudio de los datos del problema.

Cada módulo debe tener un proceso de refinamiento para expresar su solución en forma ordenada, lo que llevara a la construcción del algoritmo correspondiente.

A partir de los algoritmos se pueden escribir y probar programas en un lenguaje determinado y con un conjunto de datos significativos.

Etapas de resolución de problemas con computadoras.

1. Análisis del problema: en su contexto del mundo real.
2. Diseño de la solución: Lo primero es la modularización del problema, es decir la descomposición en partes con funciones bien definidas y datos propios estableciendo la comunicación entre los módulos.
3. Especificación del algoritmo: La elección adecuada del algoritmo para la función de cada modulo es vital para la eficiencia posterior.
4. Escritura del programa: Un algoritmo es una especificación simbólica que debe convertirse en un programa real sobre un lenguaje de programación concreto.

5. Verificación: una vez escrito el programa en un lenguaje real y depurado los errores sintácticos se debe verificar que su ejecución conduzca al resultado deseado con datos representativos del problema real.

Programación modular – programación estructurada

Se dice modular porque permite la descomposición del problema en módulos y estructurada solo permite la utilización de tres estructuras: Asignación, selección, repetición.

Algoritmo

El termino algoritmo es en honor del matemático árabe del siglo IX, *Abu Jafar Mohamed ibn Musa Al Khowârizmî*. Refiere conjunto de reglas, ordenadas de forma lógica, finito y preciso para la solución de un problema, con utilización o no de un computador.

En la actualidad al término se lo vincula fuertemente con la programación, como paso previo a la realización de un programa de computación aunque en realidad es una metodología de resolución presente en muchas de las actividades que se desarrolla a lo largo de la vida.

Desde los primeros años de escuela se trabaja con algoritmos, en especial en el campo de las matemáticas. Los métodos utilizados para sumar, restar, multiplicar y dividir son algoritmos que cumplen perfectamente las características de precisión, finitud, definición y eficiencia.

Para que el algoritmo pueda ser fácilmente traducido a un lenguaje de programación y luego ser ejecutado la especificación debe ser clara, precisa, que pueda ser interpretada con precisión y corresponda a pocas acciones, si esto no ocurre será necesario acudir a desarrollar un mayor nivel de refinamiento.

La utilización de refinamientos sucesivos es lo que permite alcanzar la solución modular que se propone.

Diseño modular, entonces, es la aplicación del criterio de refinamientos sucesivos, partiendo de un plan de acción, determinando que hacer, por aplicación de los conocimientos estratégicos de resolución pasando luego al como hacerlo con los conocimientos tácticos para la realización del algoritmo.

La programación de algoritmos representa un caso de resolución de problemas que requiere representación mental del mundo real, adaptación para tener una solución computable y criterio para elegir una alternativa eficiente de implementación.

Cuando se analiza un problema, particularmente de programación, y éste es difícil de describir, el plan de acción recomendable para alcanzar la solución es comenzar trazando un esbozo de las formas más gruesas, para que sirvan de andamio a las demás; aunque algunas de ellas se deban cambiar posteriormente. Después, se agregan los detalles, (obteniéndose el algoritmo refinado), para dotar a estos esqueletos de una estructura más realista.

Durante la tarea de integración final, se descartan aquellas primeras ideas provisionales que ya no encajan en la solución. Por lo que, hasta que no se haya visto el conjunto global es imposible encontrarle sentido a ninguna de las partes por sí solas.

Siempre es mejor explicar un misterio en términos de lo que se conoce, pero cuando esto resulta difícil de hacer, se debe elegir entre seguir tratando de aplicar las antiguas teorías, o de descartarlas y probar con otras nuevas. Siguiendo este análisis, se define como reduccionistas a aquellas personas que prefieren trabajar sobre la base de ideas existentes, y como renovadores a los que les gusta impulsar nuevas hipótesis. En programación debe encontrarse un equilibrio entre ambas posturas.

La programación como toda actividad que requiere creatividad necesita que se produzca un salto mental que se puede sintetizar como señala David Perkins en:

1. Larga búsqueda, se requiere esfuerzo en analizar y buscar.
2. Escaso avance aparente: el salto mental sobreviene tras un avance que parece escaso o no muy evidente, pero sobreviene.

3. Acontecimiento desencadenante: El típico proceso de hacer clic comienza con un acontecimiento que lo desencadena.
4. Chasquido cognitivo: De pronto aparece la solución la que sobreviene con rapidez que hace que las piezas encajen con precisión, aun cuando sea necesario todavía ajustar algunos detalles. Pero la idea generadora apareció.
5. Transformación. Este avance nos va modificando nuestro mundo mental.

En síntesis, la practica del salto de pensamiento requiere en primer lugar de buscar analogías, en segundo lugar juegan un papel importante las conexiones lógicas, formulación de una pregunta crucial ocupa un papel decisivo. El repertorio de acciones tras el salto del pensamiento se expande para incluir no solo la analogía sino una extrapolación lógica y la formulación de la pregunta adecuada

Para encontrar una solución muchas veces se necesita desplazarse bastante por el entorno adecuado. Conforme a esto Thomas Edison declaro que la invención significa 99% de transpiración y 1% de inspiración, en contraposición con Platón que sostenía que las soluciones aparecen por inspiración divina.

Muchos problemas son razonables, cabe razonarlos paso a paso para alcanzar la solución. Otros son irrazonables no se prestan a un a reflexión por etapas.

Definición

Algoritmo

Especificación rigurosa (debe expresarse en forma univoca) de la secuencia de pasos, instrucciones, a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito. Esto último supone que el algoritmo empieza y termina, en el caso de los que no son de tiempo finito (ej. Sistemas en tiempo real) deben ser de número finito de instrucciones.

En definitiva un algoritmo es una especificación ordenada de la solución a un problema de la vida real. Son el fundamento de la programación de computadores en el paradigma de programación imperativo.

Bajo este paradigma desarrollar un programa significa indicarle al computador, con precisión, sin ambigüedad y en un lenguaje que este pueda entender, todos y cada uno de los pasos que debe ejecutar para lograr el objetivo propuesto.

Previo a la traducción en un lenguaje de programación es necesario poder entender el problema, conocer las pre condiciones, establecer cual debe ser la pos condición, o aquello que debe ser cierto al finalizar la ejecución del algoritmo, en definitiva entender claramente *Que* es lo que se debe hacer para luego avanzar en *Como* hacerlo. Aquí debe utilizarse todas las herramientas al alcance de la mano para el desarrollo del algoritmo como paso previo a la solución del problema por el computador.

Existen varias técnicas para representar formalmente un algoritmo, una descriptiva llamada pseudo código, y otras graficas como los diagrama de flujo, diagrama Nassi Sneiderman, Diagramas de Lindsay, diagramas de Jackson, entre otros, en este caso se presentara una notación algorítmica similar a la presentada por Piere Scholl en el texto Esquemas algorítmicos fundamentales: Secuencia e iteración.

Definición

Algoritmo:

Secuencia finita de instrucciones, reglas o pasos que describen en forma precisa las operaciones que una computadora debe realizar para llevar a cabo una tarea en tiempo finito [Knuth, 1968].

Descripción de un esquema de comportamiento expresado mediante un repertorio finito de acciones y de informaciones elementales, identificadas, bien comprendidas y realizables a priori. Este repertorio se denomina léxico[Scholl, 1988].

Esta formado por reglas, pasos e instrucciones.
Las reglas especifican operaciones.
La computadora es el agente ejecutor.
La secuencia de reglas y la duración de la ejecución son finitas.

Características de un algoritmo

Un algoritmo debe tener al menos las siguientes características:

1. **Ser preciso**: esto significa que las operaciones o pasos del algoritmo deben desarrollarse en un orden estricto, ya que el desarrollo de cada paso debe obedecer a un orden lógico.
2. **Ser definido**. Ya que en el área de programación, el algoritmo es el paso previo fundamental para desarrollar un programa, es necesario tener en cuenta que el computador solo desarrollará las tareas programadas y con los datos suministrados; es decir, no puede improvisar y tampoco inventará o adivinará el dato que necesite para realizar un proceso. Por eso, el algoritmo debe estar plenamente definido; esto es, que cuantas veces se ejecute, el resultado depende estrictamente de los datos suministrados. Si se ejecuta con un mismo conjunto de datos de entrada, el resultado deberá ser siempre el mismo.
3. **Ser finito**: esta característica implica que el número de pasos de un algoritmo, por grande y complicado que sea el problema que soluciona, debe ser limitado. Todo algoritmo, sin importar el número de pasos que incluya, debe llegar a un final. Para hacer evidente esta característica, en la representación de un algoritmo siempre se incluyen los pasos inicio y fin.
4. **Presentación formal**: para que el algoritmo sea entendido por cualquier persona interesada es necesario que se exprese en alguna de las formas comúnmente aceptadas; pues, si se describe de cualquier forma puede no ser muy útil ya que solo lo entenderá quien lo diseñó. Las formas de presentación de algoritmos son: el pseudo código, diagrama de flujo y diagramas de Nassi/Schneiderman, entre otras. En esta publicación se propondrá una notación algorítmica y se darán las equivalencias entre la propuesta y las existentes y también con las sentencias de los lenguajes de programación, en particular Pascal y C.
5. **Corrección**: el algoritmo debe ser correcto, es decir debe satisfacer la necesidad o solucionar el problema para el cual fue diseñado. Para garantizar que el algoritmo logre el objetivo, es necesario ponerlo a prueba; a esto se le llama verificación o prueba de escritorio.
6. **Eficiencia**: hablar de eficiencia o complejidad de un algoritmo es evaluar los recursos de cómputo que requiere para almacenar datos y para ejecutar operaciones frente al beneficio que ofrece. En cuanto menos recursos requiere será más eficiente el algoritmo.

La vida cotidiana está llena de soluciones algorítmicas, algunas de ellas son tan comunes que no se requiere pensar en los pasos que incluye la solución. La mayoría de las actividades que se realizan diariamente están compuestas por tareas más simples que se ejecutan en un orden determinado, lo cual genera un algoritmo. Muchos de los procedimientos utilizados para desarrollar tareas cotidianas son algorítmicos, sin embargo, esto no significa que todo lo que se hace está determinado por un algoritmo.

El primer paso en el diseño de un algoritmo es conocer la temática a tratar, el segundo será pensar en las actividades a realizar y el orden en que deben ejecutarse para lograr el objetivo, el tercero y no menos importante es la presentación formal.

Propiedades de los algoritmos

1. Especificación precisa de la entrada: El algoritmo debe dejar claro el número y tipo de datos de entrada y las condiciones iniciales que deben cumplir esos valores de entrada para conseguir que las operaciones tengan éxito.
2. Especificación precisa de cada instrucción: cada etapa del algoritmo debe estar definida con precisión, no debe haber ambigüedades sobre las acciones que se deben ejecutar en cada momento.
3. Un algoritmo debe ser exacto y correcto: Un algoritmo se espera que resuelva un problema y se debe poder demostrar que eso ocurre. Si las condiciones de entrada se cumplen y se ejecutan todos los pasos el algoritmo entonces debe producir la salida deseada.
4. Un algoritmo debe tener etapas bien definidas y concretas, un número finito de pasos, debe terminar y debe estar claro la tarea que el algoritmo debe ejecutar.
5. Debe ser fácil de entender, codificar y depurar.
6. Debe hacer uso eficiente de los recursos de la computadora

Finitud: en longitud y duración.
Precisión: Determinar sin ambigüedad las operaciones que se deben ejecutar.
Efectividad: las reglas pueden ejecutarse sin el ordenador obteniéndose el mismo resultado.
Generalidad: Resolver una clase de problema y no un problema particular.
Entradas y salidas: puede tener varias entradas pero una sola salida, el resultado que se debe obtener.

Eficiencia de un algoritmo

Se pueden tener varias soluciones algorítmicas para un mismo problema, sin embargo el uso de recursos y la complejidad para cada una de las soluciones puede ser muy diferente.
La eficiencia puede definirse como una métrica de calidad de los algoritmos asociada con la utilización optima de los recursos del sistema de cómputo donde se ejecutara el algoritmo, su claridad y el menor grado de complejidad que se pueda alcanzar. *Hacer todo tan simple como se pueda, no más (Albert Einstein).*
La eficiencia como factor espacio temporal debe estar estrechamente relacionada con la buena calidad, el funcionamiento y la facilidad del mantenimiento.
Medidas de eficiencia para N = 10.0000

Eficiencia		Iteraciones	Tiempo estimado
Logarítmica	$Log_2\ N$	14	Microsegundos
Lineal	N	10.000	0.1 segundo
Logarítmica lineal	$N * Log_2\ N$	140.000	2 segundos
Cuadrática	N^2	10.000^2	15-20 minutos
Poli nómica	N^k	10.000^K	Horas
Exponencial	2^N	$2^{10.000}$	Inmedible

Complejidades más comunes

1. Complejidad constante: se expresa como O(1). Se encuentra en algoritmos sin ciclos, por ejemplo en un intercambio de variables.
2. Complejidad logarítmica: Es una complejidad eficiente, la búsqueda binaria tiene esta complejidad.
3. Complejidad lineal: se encuentra en los ciclos simples.

4. Complejidad logarítmica lineal: Los mejores algoritmos de ordenamiento tienen esta complejidad.
5. Complejidad cuadrática: Aparece en el manejo de matrices de dos dimensiones, generalmente con dos ciclos anidados.
6. Complejidad cúbica: Aparece en el manejo de matrices de tres dimensiones, generalmente con tres ciclos anidados.
7. Complejidad exponencial: es la complejidad de algoritmos recursivos.

Léxico y algoritmo

Para escribir un algoritmo deben seguirse un conjunto de pasos básicos

1. Comprender el problema
2. Identificar los elementos a incluir en el léxico: constantes, tipos, variables y acciones.
3. Encontrar la forma de secuenciar las acciones para obtener el resultado, esto es, alcanzar las poscondiciones a partir de un estado inicial que cumple con la precondición. Para establecer el orden de las acciones los lenguajes de programación proporcionan mecanismos de composición: Secuenciación, análisis de casos, iteración y recursion.
4. Al organizar las acciones en el tercer paso puede ocurrir que se detecte que faltan elementos en el léxico o que algún aspecto del problema no ha sido bien comprendido lo cual requeriría volver a los pasos anteriores.
5. Nunca la solución aparece en el primer intento, en general aparece en un proceso cíclico, entonces se debe:
6. Escribir el léxico,
    a. escribir la primera versión,
    b. incluir en el léxico nuevos elementos que faltaban,
    c. escribir la nueva versión del algoritmo y así sucesivamente

Estructura de un algoritmo

```
LEXICO {Léxico Global del algoritmo}
        {Declaración de tipos, constantes, variables y acciones}
        Acción 1
                PRE {Precondición de la acción 1}
                POS {Poscondición de la acción 1}
                LEXICO {Léxico local, propio de la acción 1}
                        Declaraciones locales
                ALGORITMO {Implementación de la acción 1}
                        {Secuencia de instrucciones de la acción 1}
                FIN {Fin implementación algoritmo de la acción 1}
ALGORITMO
        PRE {Precondición del algoritmo principal}
        POS {Poscondición del algoritmo principal}
                {Secuencia de instrucciones del algoritmo principal}
FIN {Fin del algoritmo principal}
```

Proceso Computacional

Se refiere a un algoritmo en ejecución. La ejecución de las instrucciones origina una serie de acciones sobre elementos de memoria que representan información manejada por el algoritmo. A nivel algorítmico se asigna un nombre a cada información de modo de manejar un par nombre-valor para cada información.

Una variable representa alguna entidad del mundo real, relevante para el problema que se quiere resolver. El efecto que producen las acciones del proceso sobre las variables produce cambio de estados o de sus valores.

Definiciones

Programa: Algoritmo escrito en un lenguaje cuyas instrucciones son ejecutables por una computadora y que están almacenados en un disco.

Tarea: Un programa se vuelve tarea a partir del momento que se lo selecciona para su ejecución y hasta que esta termina.

Proceso: programa en ejecución, se ha iniciado pero aún no ha finalizado.

Lenguajes de programación: notación que permite escribir programas a mayor nivel de abstracción que los lenguajes de máquina. Sus instrucciones deben ser traducidas a lenguaje de máquina.

Lenguaje de máquina: Instrucciones que son ejecutables por el hardware de una computadora.

Paradigmas de programación

Paradigma: Colección de conceptos que guían el proceso de construcción de un programa. Estos conceptos controlan la forma en que se piensan y formulan los programas.

Imperativo – Procedural – Objetos.

Declarativo – Funcional – Lógico.

Dato Información Conocimiento

Dato: <objeto><atributo><valor> sin interpretar.

Información: añade significado al dato.

Conocimiento: Añade propósito y capacidad a la información. Potencial para generar acciones.

Problema

Enunciado con una incógnita, la solución es encontrar el valor de esa incógnita.

Problema computacional o algorítmico: tarea ejecutada por una computadora con una especificación precisa de los datos de entrada y de los resultados requeridos en función de estos.

Clase de problemas

No computables: No existe un algoritmo.

Computables

Tratables: Existe un algoritmo eficiente.

Intratable: No existe algoritmo eficiente.

Expresiones Sentencias Léxico

Expresiones: secuencia de operadores y operandos que se reduce a un solo valor.

Sentencias: acción produce un efecto, puede ser primitiva o no primitiva.

Léxico: Descripción del conjunto de acciones e informaciones a partir de la cual se expresa el esquema de comportamiento del algoritmo.

Pasos para resolver un algoritmo

Comprender el problema.

Identificar información y acciones a incluir en el léxico (constantes, tipos, variables y acciones).

Encontrar un camino de secuenciar las acciones para obtener el resultado, es decir para alcanzar la poscondición a partir del estado inicial que cumple con la precondición.

Acciones primitivas y derivadas

Acciones primitivas: Incorporadas por el lenguaje.

Acciones derivadas: realizadas mediante la combinación de acciones primitivas con el objeto de desarrollar una tarea en particular. Son complementarias y pueden ser desarrolladas por el programador.

Estructura de un algoritmo

```
LEXICO {Léxico Global del algoritmo}
        {Declaración de tipos, constantes, variables y acciones}
        Acción 1
                PRE {Precondición de la acción 1}
                POS {Poscondición de la acción 1}
                LEXICO {Léxico local, propio de la acción 1}
                        Declaraciones locales
                ALGORITMO {Implementación de la acción 1}
                        {Secuencia de instrucciones de la acción 1}
                FIN {Fin implementación algoritmo de la acción 1}

ALGORITMO
        PRE {Precondición del algoritmo principal}
        POS {Poscondición del algoritmo principal}
                {Secuencia de instrucciones del algoritmo principal}
FIN {Fin del algoritmo principal}
```

Resumen:

En el presente capitulo se introdujeron términos y frases de la disciplina en estudio.
Se abordo el problema desde un punto de vista conceptual definiendo con precisión términos y frases para evitar ambigüedades. Se puso especial énfasis en la necesidad de pensar las soluciones antes de escribir código. Se establecieron cuales son los pasos que deben seguirse para una solución correcta de problemas y cuales son los problemas que pueden tratarse en forma algorítmica. Se definió cuales son las características deseables de los algoritmos y se introdujo el concepto de eficiencia de modo de hacer, las cosas tan simple como se pueda.
Se introdujo, además una representación semi formal para la descripción de los algoritmos

Acciones y funciones

2

Objetivos de aprendizaje
Dominando los temas del presente trabajo Usted podrá.

1. Entender la descomposición como forma de resolución de problemas.
2. Dar valor a la reusabilidad en búsqueda de la eficiencia en la escritura el código.
3. Establecer comunicación entre módulos.
4. Comprender las ventajas de la descomposición
5. Diferenciar acciones de funciones y los distintos tipos de parámetros y datos

Introducción

En este trabajo se analiza la descomposición como forma de alcanzar la solución de problemas. Una regla básica de la programación indica que si existe un programa de longitud L tal que L = L1 + L2, se dice que el esfuerzo de resolver L es mayor que la suma de los esfuerzos de resolución de L1 y L2, aun cuando haya que derivar esfuerzo para la integración de los módulos.

SI L = L 1 + L 2
Entonces
$Esfuerzo_{(L)} > Esfuerzo_{(L1)} + Esfuerzo_{(L2)}$

Antes de analizar las particularidades de las acciones y funciones es necesaria la definición de los términos que se utilizan.

Modularizacion

En general los problemas a resolver son complejos y extensos, puede incluso presentarse situaciones en que una parte del problema deba ser modificada para adaptarse a nuevos requerimientos. Se hace necesario conocer algunas herramientas que permitan facilitar la solución de estos problemas, la abstracción y la descomposición pueden ayudar a ello. La abstracción permitirá encontrar y representar lo relevante del problema y la descomposición se basa en el paradigma de ¨ dividir para vencer ¨. La descomposición tiene como objetivo dividir cada problema en subproblemas cada uno de los cuales será de más simple solución.
Es conveniente, e importante descomponer por varias razones:

- Favorece la comprensión. Una persona entiende un problema de características complejas partiendo la información, descomponiendolo. Por esto para comprender un problema complejo del mundo real es necesario dividirlo o modularizar.
- Favorece el trabajo en equipo. Cada programador recibe las especificaciones la tarea a realizar y las restricciones con las que debe manejarse.
- Favorece el mantenimiento. Las tareas involucradas en este mantenimiento, están vinculadas con detectar corregir errores, modificar y/o ampliar codigo. Esto es mucho mas sencillo si se hace un análisis y control de una porción o modulo y luego se lo relaciona que atendiendo de una sola vez la totalidad del problema.

- Permite la reusabilidad del código. Siempre es deseable, de ser posible, hacer uso de código ya escrito.
- Permite además, como veremos, separar la lógica de la algoritmia con el tipo o estructura de dato involucrada. Alcanzando reusabilidad y diversidad de tipos tendiente a un criterio de programación mas genérica centrada en la algoritmia.

Módulos

Un problema debe ser descompuesto en subproblemas que se denominan módulos en los que cada uno tendrá una tarea especifica, bien definida y se comunicaran entre si adecuadamente para conseguir un objetivo común. Un modulo es un conjunto de instrucciones mas un conjunto de datos que realizan una tarea lógica.

Alcance de los datos

El desarrollo de un programa complejo con muchos módulos en los que en cada uno de ellos se deben definir los propios tipos de datos puede ocasionar algunos de los inconvenientes siguientes:

- Demasiados identificadores.
- Conflictos entre los nombres de los identificadores de cada modulo.
- Integridad de los datos, lo que implica que puedan usarse datos que tengan igual identificador pero que realicen tareas diferentes.

La solución a estos problemas se logra con una combinación entre ocultamiento de datos y uso de parámetros.

Datos locales y globales

Unos se declaran en la sección de declaración del programa principal, los otros en la sección de declaración de cada modulo. Otros pueden ser declarados en un bloque determinado, por lo que solo tendrá visibisilidad en el mismo.

Local y global puede ser sido utilizado de manera absoluta pero los módulos pueden anidarse las reglas que gobiernan el alcance de los identificadores son:

- El alcance de un identificador es el bloque del programa donde se lo declara.
- Si un identificador declarado en un bloque es declarado nuevamente en un bloque interno al primero el segundo bloque es excluido del alcance de la primera sección. Algunos lenguajes permiten la incorporación de operadores de ámbito.

Ocultamiento y protección de datos

Todo lo relevante para un modulo debe ocultarse a los otros módulos. De este modo se evita que en el programa principal se declaren datos que solo son relevantes para un modulo en particular y se protege la integridad de los datos.

Parámetros

Son variables cuya característica principal es que se utilizan para transferir información entre módulos. En programas bien organizados toda información que viaja hacia o desde módulos se hace a través de parámetros.

Hay dos tipos de parámetros, los pasados por valor y los pasados por referencia o dirección.

Cuando existen datos compartidos entre módulos una solución es que un modulo pase una copia de esos datos al otro. En este caso el pasaje se denomina pasaje por valor. El modulo que recibe esta copia no puede efectuar ningún cambio sobre el dato original; el dato original se encuentra de este modo protegido de modificación.

Los parámetros pasados por referencia o dirección no envían una copia del dato sino envían la dirección de memoria donde el dato se encuentra por lo que tanto el proceso que lo llama como el proceso llamado pueden acceder a dicho dato para modificarlo.
Razones por la que es conveniente la utilización de parámetros sobre las variables globales.

Integridad de los datos

Es necesario conocer que datos utiliza con exclusividad cada modulo para declararlos como locales al modulo ocultándolo de los otros, si los datos pueden ser compartidos por ambos módulos debería conocerse cuales son, si el modulo los utiliza solo de lectura o puede modificarlos y es aquí donde se utilizan los parámetros.

Protección de datos

Si una variable es local a un modulo se asegura que cualquier otro modulo fuera del alcance de la variable no la pueda ver y por lo tanto no la pueda modificar. Dado que las variables globales pueden ser accedidas desde distintos módulos, la solución para evitar modificaciones no deseadas es el pasaje como parámetros valor.

Uso de parámetros para retornar valores

Si bien el pasaje por valor es útil para proteger a los datos, existen situaciones en las que se requiere hacer modificaciones sobre los datos y se necesitan conocer esas modificaciones. Para esto se deben utilizar parámetros pasados por referencia. Los parámetros enviados por referencia son aptos además para enviar datos en ambos sentidos.

Utilidad del uso de parámetros

El uso de parámetros independiza a cada modulo del nombre de los identificadores que utilizan los demás. En el caso de lenguajes fuertemente tipiados solo importa la correspondencia en cantidad tipo y orden entre los actuales del llamado y los formales de la implementación con independencia del nombre del identificador.

Reusabilidad

El uso de parámetros permite separar el nombre del dato, del dato en si mismo, lo que permite que el mismo código sea utilizado en distintas partes del programa simplemente cambiando la lista de parámetros actuales.

Acciones

El léxico establece el nivel de abstracción de un algoritmo. Es decir, introduce las variables, las constantes, los tipos de datos y las acciones con que se construye el algoritmo.
El concepto de acción está muy ligado al concepto de abstracción. Se analiza como abstracción por parametrización y abstracción por especificación.
Las acciones pueden desarrollar distintos cálculos y retornar cero, uno o mas resultados al programa que lo invoca. En general se los usa con una invocación a si mismo, no retornan valor en el nombre (solo podría retornar uno), en cambio retornan valores en los parámetros (los parámetros variables o enviados por referencia.

Utilización de acciones

Una *acción* es una secuencia de instrucciones que se identifica por un nombre y que puede ser invocada desde un algoritmo principal o desde otra acción. Cuando una acción es invocada desde algún punto de un algoritmo, el flujo de ejecución se traslada a la primera instrucción de la acción, entonces la acción se ejecuta hasta el final y cuando acaba, el flujo se traslada de nuevo a la instrucción del algoritmo que sigue a aquella que origino la invocación.
Una acción debe tener un efecto bien definido, lo que significa que debe ser cohesiva. El nombre de la acción es conveniente que evoque la tarea que realiza. Hay que definir acciones que sean aplicables a cualquier posible conjunto de valores de entrada y no a un valor concreto.

Entre una acción y el algoritmo que la invoca se debe producir una comunicación de valores: el algoritmo debe proporcionar los valores de entrada y la acción puede retornar el resultado, o puede modificar el estado de alguno de ellos.
Puede haber acciones en las que la comunicación se realice mediante variables globales definidas en el ámbito del algoritmo principal, que pueden ser manejadas por la acción. Pero esta no es la forma mas apropiada. Una acción, en principio, nunca debería acceder a variables globales.
Los *parámetros* son el mecanismo que posibilita escribir acciones generales, aplicables a cualquier valor de la entrada, e independientes del léxico del algoritmo.

Acciones con parámetros

Un *parámetro* es un tipo especial de variable que permite la comunicación entre una acción y el algoritmo que la invoca, ya sea que este pase a la acción un valor de entrada o bien que la acción devuelva el resultado al algoritmo, o que pasen ambas cosas simultáneamente.
A los parámetros que proporcionan un valor de entrada se los llaman *Parámetros dato*, y a los que, además de recoger un valor, retornan un resultado, se los llama *dato-resultado*. En la declaración de una acción, la lista de los parámetros se indica después del nombre de la acción entre paréntesis y separados por coma.

```
Nombre(dato tipo p1 ,dato-resultado tipo p2 )
```

Un parámetro también cuenta con una característica: la dirección de memoria en la que se realiza la transmisión de la información.
Se denomina *paso de parámetros* al modo en que se establece la comunicación entre los argumentos pasados a la acción desde el algoritmo y los parámetros de la acción; en la llamada se pasan los datos de entrada, y en el retorno se devuelven los resultados. Cada argumento se liga con el parámetro que ocupa la misma posición en la declaración de la acción y ambos deben coincidir en tipo.
En la invocación a una acción, el algoritmo debe proporcionar un valor para cada parámetro dato, mientras que debe indicar para cada parámetro dato-resultado (&) qué variable de su léxico recogerá el valor.

Abstracción y acciones

El término *abstracción* se refiere al proceso de eliminar detalles innecesarios en el dominio del problema y quedarse con aquello que es esencial para encontrar la solución. Como resultado de aplicar la abstracción, se obtiene un modelo que representa la realidad que nos ocupa. Este modelo no es la realidad, es una simplificación de esa realidad que establece un nivel de abstracción mas apropiado para razonar sobre el problema y encontrar la solución.
La abstracción también está relacionada con los lenguajes de programación. Estos ofrecen mecanismos de abstracción que determinan la forma de razonar de un programador.
El concepto de acción conjuga dos técnicas de abstracción que son la parametrización y la especificación. La parametrización es un mecanismo por el cual se generaliza una declaración para que no sea aplicado a un único caso, sino que sirva para cualquier valor que pueda tomar cierto parámetro.
La abstracción por especificación es la separación entre la especificación (el Qué) y la implementación (el cómo). En el caso de acciones, se refiere a distinguir entre la descripción de qué hace la acción y el cómo se la implementa. Una vez que se define una nueva acción, se las utiliza del mismo modo que si se tratase de una acción primitiva.
Del mismo modo que el algoritmo, las acciones se especifican mediante una precondición y una poscondición. La precondición establece las restricciones que satisfacen los parámetros (datos de entrada) para que se pueda ejecutar la acción, y la postcondición describe el resultado.

Cada acción debe ir siempre acompañada de una descripción textual de su efecto y de su precondición y postcondición. De esta forma, cualquier programador podría conocer qué hace y podría utilizarla sin conocer cómo lo hace.
El programador solo debe preocuparse por que se cumpla la precondición al invocar la acción y tendrá la certeza de que la acción cumplirá su objetivo.

Tipos de Parámetros

Una acción se comunica con el algoritmo que lo invoca a través de los parámetros. Es necesaria una comunicación en dos sentidos. El algoritmo debe proporcionar los datos de entrada que manipulará durante su ejecución y la acción debe retornar los resultados que obtiene.
El *tipo de parámetro* indica cómo los valores de los argumentos son ligados a los parámetros.
El tipo de parámetro *dato* solo permite que el parámetro pueda recibir el valor de un argumento mientras que el tipo de parámetro *dato-resultado* permite que el parámetro pueda recibir un valor y pueda retornar un resultado. En la declaración de una acción hay que indicar el tipo de cada parámetro. La declaración de una acción se la denomina *cabecera*.
Se denomina *paso por valor* al paso de parámetros que corresponda al tipo de parámetro dato y *paso por referencia* al que corresponda al tipo de parámetro dato-resultado.
Parámetro dato (o parámetro de entrada)
El valor del argumento es asignado al parámetro en el momento de la llamada. El argumento puede ser una expresión del mismo tipo de dato que el parámetro. Se trata de una comunicación unidireccional: solamente se transmite información desde el punto del llamado hacia la acción.
Una regla de buen estilo de programación es no modificar el valor de parámetros tipo dato, aunque ello no tenga efecto fuera de la acción.
Parámetro dato-resultado (o parámetro de entrada y salida)
El valor del argumento es asignado al parámetro en el momento de la llamada y al final de la ejecución el valor del parámetro es asignado al argumento. Se trata de una comunicación bidireccional. Si la ejecución de la acción provoca un cambio en el valor del parámetro, en el momento del retorno el argumento tendrá el valor del parámetro al finalizar la ejecución.
Un argumento para un parámetro dato-resultado debe ser una variable del mismo tipo de dato que el parámetro y no puede ser una expresión.
Acción para el intercambio de dos variables: La acción recibe dos identificadores con dos valores y debe retornar los identificadores con los valores cambiados

```
Intercambiar(dato-resultado a, b : Entero) : una acción
PRE { a, b : Entero, a = A, b = B }
POST { a = B, b = A }
LÉXICO
     t : Entero
ALGORITMO
t = a;
a = b;
b = t
FIN
```

Estos son los parámetros, que como deber ser modificados sus valores por la accion se definen como Dato-Resultado, son las variables que intercambian informacion entre el programa principal y el modulo. Son

Este identificador se necesita como variable auxiliar para poder hacer el intercambio. Su valor no lo necesita el programa que invoca a la accion, solo interesa su visibilidad en el modulo por tanto no es

Ejemplo en C++.

/* Funcion que no retorna valor en su nombre, el tipo de retorno void es lo que lo indica. La función tiene dos parámetros que son pasados por referencia o dirección, el & delante del

parámetro es el que lo indica. Al ser enviados por referencia el programa que lo invoca recibe los identificadores con los valores intercambiados */

```
void Intercambiar ( int &a, int &b)
{// comienzo del bloque de declaración de la funcion
        int t;    //declaración de una variable auxiliar t
        t = a;    //asignación a t del valor de a, para contenerlo
        a = b;    //asignación a a del valor de b para intercambiarlo
        b = t;    //asignación a b del valor que contenía originalmente a.
} // fin del bloque de la funcion
```

Beneficios del uso de acciones

En la actualidad, las clases de los lenguajes orientados a objetos son los mecanismos más adecuados para estructurar los programas. No obstante, las acciones no desaparecen con estos nuevos mecanismos porque dentro de cada clase se encuentran acciones.

Una acción tiene cuatro propiedades esenciales. Ellas son:

1. Generalidad
2. Ocultamiento de información
3. Localidad
4. Modularidad

De estas propiedades, se deducen una serie de beneficios muy importantes para el desarrollo de algoritmos.

1. Dominar la complejidad
2. Evitar repetir código
3. Mejorar la legibilidad
4. Facilitar el mantenimiento
5. Favorecer la corrección
6. Favorecer la reutilización

La función desarrollada tiene la propiedad de la reusabilidad ya mencionada, es decir, puede ser invocada en distintos puntos del programa con pares de identificadores de tipo entero y producirá el intercambio. Sin embargo para que pueda producirse el intercambio utilizando esta acción el requerimiento es que los parámetros sean de tipo int.

Algoritmos de UTNFRBA utilizara para las implementaciones C++. Este lenguaje agrega una característica de reutilización de código realmente poderosa que es el concepto de plantilla. Las plantillas de funciones, que son las que utilizaremos son de gran utilidad para evitar escribir código cuando las acciones deben realizar operaciones idénticas pero con distintos tipos de datos.

Las definiciones de las plantillas comienzan con la palabra template seguida de una lista de parámetros entre **<** y >, a cada parámetro que representa un tipo se debe anteponer la palabra

typename o class. Nosotros usaremos **typename**. Por ejemplo:

template <typename Tn > si solo involucra un tipo de dato

o

template < typename R, typename T> si involucra dos tipos de dato diferentes

La función del ejemplo tiene un solo tipo de dato que es int, si quisiéramos hacer el intercambio utilizando variables de otro tipo, por ejemplo doublé, también tendríamos un único tipo en toda la acción. En lugar de colocar un ti´po definido optaremos por un tipo genético al que llamaremos T.

La función quedaría:

```
//definicion de la plantilla de la función intercambiar
```

```
template < typename T>
void Intercambiar ( T &a, T &b) //con dos parámetros de tipo generico
{// comienzo del bloque de declaración de la funcion
        T t;      //declaración de una variable auxiliar t
        t = a;    //asignación a t del valor de a, para contenerlo
        a = b;    //asignación a a del valor de b para intercambiarlo
        b = t;    //asignación a b del valor que contenía originalmente a.
} // fin del bloque de la funcion
```

El tipo genérico es T por lo cual en cada aparición de int de la función original fue reemplado por el tipo genérico y ahora la misma acción es valida no solo para diferentes pares de valores (reusabilidad) sino además para distintos tipos de datos (polimorfismo)

Funciones

Si el propósito es calcular un valor a partir de otros que se pasan con argumentos y se utilizan acciones, habrá que definir un parámetro dato-resultado para que retorne el valor calculado. Las acciones que retornan valor en los parámetros no pueden ser utilizadas en una expresión.
Para resolver este problema, los lenguajes de programación incorporan el concepto de función. Las funciones devuelven un único valor. La función supone extender el conjunto de operadores primitivos. Se invocan en una expresión.
En cada declaración se especifica el nombre de la función, la lista de los parámetros y finalmente el tipo de valor que retorna la función.

Nombre_funcion (par_1 : td_1 ; ... ; par_n : td_n) : tr : una función

Una función no debe tener efectos laterales. Es decir, debe limitarse a calcular un valor y no modificar ninguno de los que se describen en el momento de la invocación.
Las acciones se utilizan para extender el conjunto de acciones primitivas. Las funciones permiten extender el conjunto de funciones u operadores primitivos. Siempre deben utilizarse dentro de una expresión.
Función que obtenga el mayor de tres números

```
Max(a, b, c : Entero) → Entero : una función
        ALGORITMO
          SI (a >= b) y (a >= c)
  ENTONCES
Max = a
  SINO SI (b >= a) y (b >= c)
  ENTONCES
        Max = b
  SINO
Max = c
        FIN_SI
        FIN_SI
      FIN.
```

Las funciones siempre retornan un unico valor. El identificador del nombre de la funcion se puede utilizar en una expresión como cualquier identificador del tipo que retorna. Las funciones pueden retornar un escalar un apuntadoro un registro

```
//definicion de la plantilla de la función Max
template < typename T>
T Max ( T a, T b, T b) \*       vea que la función retorna un valot de tipo T y los parámetros no
                                están pasados por referencia*\
```

```
{// comienzo del bloque de declaración de la funcion
        T t = a;  //declaración de una variable auxiliar t para contener el maximo
        if (b > t) t = b;
        if (c > t) t = c;
        return t;          //retorna en t el valor del maximo.
} // fin del bloque de la funcion
```

Concepto de recursividad:

Es un proceso que se basa en su propia definición. Una función puede invocarse a sí misma como parte de los tratamientos de cálculo que necesita para hacer su tarea

Parte de instancias complejas y las define en términos de instancias más simples del mismo problema, llegando a un punto donde las instancias más simples son definidas explícitamente.

Define el problema en términos de un problema más simple de la misma naturaleza.

Debe disminuir el espacio del problema en cada llamada recursiva

Hay una instancia particular que se conoce como caso base o caso degenerado

Divide el problema original en subproblemas más pequeños. Cuando es lo suficientemente chico se resuelve directamente y se combinan soluciones del subproblema hasta que queda resuelto el problema

Tiene:

- ✓ Una ecuación de recurrencia, en función de términos anteriores $T_n = F(T_{n-1}, T_{n-2}, T_0)$.
- ✓ Uno o varios términos particulares que no dependen de los anteriores. $Ti = G_{(i)}$ (base)

Funcion Factorial

- ✓ Ecuación de recurrencia : $n! = n * (n-1)!$
- ✓ Condiciones particulares: $0! = 1$

Instancias que permanecen en memoria:

Funcion PotenciaNatural

- ✓ Ecuación de recurrencia : $a^n = a^{(n-1)} * a$ si $n > 1$
- ✓ Condiciones particulares: $a^0 = 1$

Funcion DivisionNatural

Dados dos valores num y den, con den <> 0 se puede definir el cálculo del cociente y el resto del siguiente modo:

- ✓ Si num < den → el cociente es = y el resto num.
- ✓ Si num <= den y si c y r son el cociente y resto entre num-den y den → cociente = c+1 y resto r.

Sucesión de Fibonacci

Una pareja de conejos tarda un mes en alcanzar la edad fértil y a partir de aquí un mes en engendrar otra pareja que al alcanzar la fertilidad engendraran otra pareja, entonces ¿Cuántos conejos habrá al término de N meses? Si los animales no mueren cada mes hay tantas parejas como la suma de los dos meses anteriores

- ✓ Ecuación de recurrencia : $Fib(n) = Fib(n-1) * (n-1)!$
- ✓ Condiciones particulares: $Fib(0) = 1$ y $Fib(1) = 1$

Meses	Padres	Hijos	Nietos	Cant Parejas		
0	I			1		
1	M			1		
2	M	I		2		
3	M	M I		3		
4	M	M M I	I	5		
5	M	M M M I	M I I	8		

Diseño y verificación de algoritmos recursivos

PotenciaNatural (a: Real; n : Entero) → Real : una función

Especificación Formal:

Pre: {(n >=0) y (n = 0 → a <> 0}

Pos: {PotenciaNatural (a,n) = a^n}

Análisis de casos:

Caso trivial	Solucion caso Trivial	n = 0	$a^n = 1$
Caso no trivial	Solucion caso no trivial	N > 0	$a^n = a^{(n-1)} * a$

Composición

PotenciaNatural(a : Real; n : Entero) → Real : una función;

Pre: {(n >=0) y (n = 0 → a <> 0}

Pos: {PotenciaNatural (a,n) = a^n}

ALGORITMO

Según n

n = 0 : PotenciaNatural ← 1;

n > 0 : PotenciaNatural ← a * Potencianatural (a – 1)

FIN

Verificación Formal: debe demostrarse que:

- ✓ Para cualquier dato de entrada que satisfaga la precondición se cumple la condición booleana del caso trivial o no trivial.
  Q(x) → Bt(x) o Bnt(x)
- ✓ Para cualquier dato de entrada que satisfaga la precondición y la condición del caso no trivial, los datos de entrada correspondientes a la llamada recursiva respetan la precondición
  Q(x) y Bnt (x) → Q(S(x))
- ✓ La solución calculada en el caso trivial respeta la poscondicion
  Q(x) y Bt(x) → R(x, triv(x))

Recursividad Indirecta:

	Procedure B(); forward;
	Procedure A();
	Begin
	End.
		funcionA();
		funcionb();
		main(){
		funcionA();

Eliminación de la recursividad

Toda funcion recursiva puede ser transformada a la forma iterativa mediante una pila (LIFO).

La recursión es una abstracción, una construcción artificial que brindan los compiladores como herramienta extra para resolver problemas, sobretodo aquellos que son de naturaleza recursiva.

```
BBR (a; inferior, superior : Entero; clave: Info)
central: Entero
Si (inferior > Superior ) BBR ← -1
SI_NO
 Central ← (inferior + suoerior)/2
 Si (a[central] 0 clave BBR ← central
  SI_NO
    Si (-------
```

RecorrerEnOrdenArbol(Dato Arbol: TPNodo; Dato Valor: Tinfo) una accion

```
ALGORITMO
        SI (Arbol <> NULO)
        ENTONCES
                RecorrerEnOrdenArbol(Arbol^.izquierdo);
                Imprimir(Arbol*.info);
                RecorrerEnOrdenArbol(Arbol^.Derecho);
        FIN_SI;
```

InsertarNodoArbol(Dato_Resultado Arbol: TPNodo; Dato Valor: Tinfo) una accion

```
ALGORITMO
        SI (Arbol <> NULO)
        ENTONCES {Nuevo(Arbol); Arbol^← <NULO, Valor, NULO>;}
        SINO    SI (Valor<Arbol^.info)
                ENTONCES InsertarNodoArbol(Arbol^.izquierdo, Valor)
                SINO    SI (Valor>Arbol^.info)
                        ENTONCES InsertarNodoArbol(Arbol^.izquierdo, Valor)
                        SINO ERROR
FIN
```

Resumen:

En este trabajo se avanza sobre la necesidad de mayor abstracción procedural introduciendo conceptos claves de programación como acciones y funciones como la forma mas adecuada de estructurar problemas en el paradigma procedural. Es una introducción a la abstracción procedural y de datos que servirá de base para sustentar futuros conocimientos de estructuración de programas en clases cuando se aborde, en otra instancia, la programación orientada a objetos.
Se dio valor a términos como reusabilidad, ocultamiento de datos, polimorfismo y cohesión. Se introdujeron conceptos como parámetros actuales, argumentos, parámetros formales, pasaje por valor, pasaje por referencia. Si introdujo un concepto desde el punto de vista de la algoritmia importante como la programación genérica que permite el lenguaje que estamos abordando.

Arreglos y registros

3

Tipos de Datos

Identifica o determina un dominio de valores y el conjunto de operaciones aplicables sobre esos valores.

1. Primitivos.
2. Derivados.
3. Abstractos.

Los algoritmos operan sobre datos de distinta naturaleza, por lo tanto los programas que implementan dichos algoritmos necesitan una forma de representarlos.
Tipo de dato es una clase de objeto ligado a un conjunto de operaciones para crearlos y manipularlos, un tipo de dato se caracteriza por

1. Un rango de valores posibles.
2. Un conjunto de operaciones realizadas sobre ese tipo.
3. Su representación interna.

Al definir un tipo de dato se esta indicando los valores que pueden tomar sus elementos y las operaciones que pueden hacerse sobre ellos.
Al definir un identificador de un determinado tipo el nombre del identificador indica la localización en memoria, el tipo los valores y operaciones permitidas, y como cada tipo se representa de forma distinta en la computadora los lenguajes de alto nivel hacen abstracción de la representación interna e ignoran los detalles pero interpretan la representación según el tipo.

Como ya vimos, los tipos de datos pueden ser.

1. **Estáticos:** Ocupan una posición de memoria en el momento de la definición, no la liberan durante el proceso solamente la liberan al finalizar la aplicación.
    a. **Simples**: Son indivisibles en datos mas elementales, ocupan una única posición para un único dato de un único tipo por vez.
        i. **Ordinales**: Un tipo de dato es ordinal o esta ordenado discretamente si cada elemento que es parte del tipo tiene un único elemento anterior (salvo el primero) y un único elemento siguiente (salvo el ultimo).
            1. **Enteros**: Es el tipo de dato numérico mas simple.
            2. **Lógico** o booleano: puede tomar valores entre dos posibles: verdadero o falso.
            3. **Carácter**: Proporcionan objetos de la clase de datos que contienen un solo elemento como valor. Este conjunto de elementos esta establecido y normatizado por el estándar ASCII.

ii. **No ordinales**: No están ordenados discretamente, la implementación es por aproximación
    1. Reales: Es una clase de dato numérico que permite representar números decimales.

b. **Cadenas**: Contienen N caracteres tratados como una única variable.

c. **Estructuras**: Tienen un único nombre para mas de un dato que puede ser del mismo tipo o de tipo distinto. Permiten acceso a cada dato particular y son divisibles en datos mas elementales.

Una estructura es, en definitiva, un conjunto de variables no necesariamente del mismo tipo relacionadas entre si de diversas formas.

Si los datos que la componen son todas del mismo tipo son homogéneas, heterogéneas en caso contrario.

Una estructura es estática si la cantidad de elementos que contiene es fija, es decir no cambia durante la ejecución del programa

i. **Registro**: Es un conjunto de valores que tiene las siguientes características:

Los valores pueden ser de tipo distinto. Es una estructura heterogénea.

Los valores almacenados se llaman campos, cada uno de ellos tiene un identificador y pueden ser accedidos individualmente.

El operador de acceso a cada miembro de un registro es l operador punto (**.**)

El almacenamiento es fijo.

ii. **Arreglo**: Colección ordenada e indexada de elementos con las siguientes características:

Todos los elementos son del mismo tipo, un arreglo es una estructura homogénea.

Los elementos pueden recuperarse en cualquier orden, simplemente indicando la posición que ocupa dentro de la estructura, esto indica que el arreglo es una estructura indexada.

El operador de acceso es el operador []

La memoria ocupada a lo largo de la ejecución del programa es fija, por esto es una estructura estática.

El nombre del arreglo se socia a un área de memoria fija y consecutiva del tamaño especificado en la declaración.

El índice debe ser de tipo ordinal. El valor del índice puede verse como el desplazamiento respecto de la posición inicial del arreglo.

Los arreglos pueden ser de varias dimensiones. Esta dimensión indica la cantidad de índices necesarias para acceder a un elemento del arreglo.

El arreglo lineal, con un índice, o una dimensión se llama vector.

El arreglo con 2 o mas índices o dimensiones es una matriz. Un grupo de elementos homogéneo con un orden interno en el que se necesitan 2 o mas índices para referenciar a un elemento de la estructura.

iii. **Archivos**: Estructura de datos con almacenamiento físico en memoria secundaria o disco.

Las acciones generales vinculadas con archivos son

Asignar, abrir, crear, cerrar, leer, grabar, Cantidad de elementos, Posición del puntero, Acceder a una posición determinada, marca de final del archivo, definiciones y declaraciones de variables.

Según su organización pueden ser secuenciales, indexados.

1. **Archivos de texto**: Secuencia de líneas compuestas por cero uno o mas caracteres que finalizan con un carácter especial que indica el final de la línea. Los datos internos son representados en caracteres, son mas portables y en general mas extensos.
2. **Archivos de tipo o binarios**: secuencia de bytes en su representación interna sin interpretar. Son reconocidos como iguales si son leídos de la forma en que fueron escritos. Son menos portables y menos extensos.

2. **Dinámicos:** Ocupan direcciones de memoria en tiempo de ejecución y se instancian a través de punteros. Esta s instancias pueden también liberarse en tiempo de ejecución. El tema de puntadores y estructuras enlazadas (estructuras relacionadas con este tipo de dato se analizan en detalle en capítulos siguentes)
    a. **Listas simplemente enlazadas**: cada elemento sólo dispone de un puntero, que apuntará al siguiente elemento de la lista o valdrá NULL si es el último elemento.
    b. Pilas: son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila**.**
    c. **Colas**: otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en salir). Los elementos se almacenan en fila, pero sólo pueden añadirse por un extremo y leerse por el otro.
    d. **Listas circulares**: o listas cerradas, son parecidas a las listas abiertas, pero el último elemento apunta al primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último". Cualquier nodo puede ser el nodo de entrada y salida.
    e. **Listas doblemente enlazad**as: cada elemento dispone de dos punteros, uno a punta al siguiente elemento y el otro al elemento anterior. Al contrario que las listas abiertas anteriores, estas listas pueden recorrerse en los dos sentidos.
    f. **Árboles**: cada elemento dispone de dos o más punteros, pero las referencias nunca son a elementos anteriores, de modo que la estructura se ramifica y crece igual que un árbol.
    g. **Árboles binarios**: son árboles donde cada nodo sólo puede apuntar a dos nodos.
    h. **Árboles binarios de búsqueda** (ABB): son árboles binarios ordenados. Desde cada nodo todos los nodos de una rama serán mayores, según la norma que se haya seguido para ordenar el árbol, y los de la otra rama serán menores.
    i. **Árboles AVL**: son también árboles de búsqueda, pero su estructura está más optimizada para reducir los tiempos de búsqueda.
    j. **Árboles B**: son estructuras más complejas, aunque también se trata de árboles de búsqueda, están mucho más optimizados que los anteriores.
    k. **Tablas HASH**: son estructuras auxiliares para ordenar listas.
    l. **Grafos**: es el siguiente nivel de complejidad, podemos considerar estas estructuras como árboles no jerarquizados.
    m. **Diccionarios**.

Registros y vectores

Registro: Es un conjunto de valores que tiene las siguientes características:
Los valores pueden ser de tipo distinto. Es una estructura heterogénea.

Los valores almacenados se llaman campos, cada uno de ellos tiene un identificador y pueden ser accedidos individualmente.
El operador de acceso a cada miembro de un registro es l operador punto (.)
El almacenamiento es fijo.
Declaracion
Genérica
NombreDelTipo = TIPO < $TipoDato_1$ $Identificador_1$; …; $TipoDato_N$ $Identificador_N$>

TipoRegistro = TIPO <Entero N; Real Y> //declara un tipo
TipoRegistro Registro; // define una variable del tipo declarado
En C

```
 struct NombreTipo {
        Tipo Identificador;
        Tipo Identificador;
}
struct TipoRegistro {
        int N;
        double Y;
};                      //      declara un tipo
TipoRegistro Registro; //       define una variable
```

Ejemplo de estructuras anidadas en C

```
struct TipoFecha {
        int     D;
        int     M;
        int     A;
};                      //      declara un tipo fecha

struct TipoAlumno {
        int             Legajo;
        string          Nombre;
        TipoFecha       Fecha
};                      //      declara un tipo Alumno con un campo de tipo Fecha
TipoAlumno Alumno;
```

Alumno es un registro con tres miembros (campos) uno de los cuales es un registro de TipoFecha.
El acceso es:

Nombre	Tipo dato	
Alumno	Registro	Registro total del alumno
Alumno.Legajo	Entero	Campo legajo del registro alumno que es un entero
Alumno.Nombre	Cadena	Campo nombre del registro alumno que es una cadena
Alumno.Fecha	Registro	Campo fecha del registro alumno que es un registro
Alumno.Fecha.D	Entero	Campo dia del registro fecha que es un entero
Alumno.Fecha.M	Entero	Campo mes del registro fecha que es un entero
Alumno.fecha.A	Entero	Campo anio del registro alumno que es un entero

Arreglo: Colección ordenada e indexada de elementos con las siguientes características:

- Todos los elementos son del mismo tipo, un arreglo es una estructura homogénea.

- Los elementos pueden recuperarse en cualquier orden, simplemente indicando la posición que ocupa dentro de la estructura, esto indica que el arreglo es una estructura indexada.
- El operador de acceso es el operador []
- La memoria ocupada a lo largo de la ejecución del programa es fija, por esto es una estructura estática.
- El nombre del arreglo se socia a un área de memoria fija y consecutiva del tamaño especificado en la declaración.
- Al elemento de posición genérica i le sigue el de posición i+1 (salvo al ultimo) y lo antecedede el de posición i-1 (salvo al primeo.
- El índice debe ser de tipo ordinal. El valor del índice puede verse como el desplazamiento respecto de la posición inicial del arreglo. La posición del primer elemento en el caso particular de C es 0(cero), indica como se dijo el desplazamiento respecto del primer elemento, para este caso es nana. En un arreglo de N elemento, la posición del ultimo es N – 1, por la misma causa.
- Los arreglos pueden ser de varias dimensiones. Esta dimensión indica la cantidad de índices necesarias para acceder a un elemento del arreglo.
- El arreglo lineal, con un índice, o una dimensión se llama vector.
- El arreglo con 2 o mas índices o dimensiones es una matriz. Un grupo de elementos homogéneo con un orden interno en el que se necesitan 2 o mas índices para referenciar a un elemento de la estructura.
- En el caso de C, no hay control interno para evitar acceder a un índice superior al tamaño físico de la estructura, esta situación si tiene control en C++, mediante la utilización de at pata el acceso (se vera mas adelante).

Declaración

Genérica
Nombre del Tipo = TABLA [Tamaño] de Tipo de dato;
ListaEnteros = TABLA[10] de Enteros; // una tabla de 10 enteros
ListaRegistros = TABLA[10] de TipoRegistro; // una tabla de 10 registros
En C:

```
TipoDeDato Identificador[Tamaño];
int VectorEnteros[10]// declara un vector de 10 enteros
TipoAlumno VectorAlum[10] // declara un vector de 10 registros.
```

Accesos

Nombre	Tipo dato	
VectorAlum	Vector	Vector de 10 registros de alumnos
VectorAlum[0]	Registro	El registro que esta en la posición 0 del vector
VectorAlum[0].Legajo	Entero	El campo legajo del registro de la posición 0
VectoeAlum[0].Fecha	Registro	El campo fecha de ese registro, que es un registro
VectorAlum[0].Fecha.D	Entero	El campo dia del registro anterior

En C++, incluye la clase <array>

```
#include <array>

// Declaración generica array<tipo de dato, cantidad elementos> identificador;
array<int,10> ArrayEnteros; // declara un array de 10 enteros
array<TipoAlumno, 10> ArrayRegistros //declara array de 10 registros

Iteradores
begin           Return iterador al inicio
end             Return iterador al final
  std::array<int,5> miarray = { 2, 16, 77, 34, 50 };
  for ( auto it = myarray.begin(); it != myarray.end(); ++it )

Capacidad
size            Return tamaño
  std::array<int,5> miarray;
  std::cout << miarray.size() << std::endl;
  // retorna 5

Elementos de  acceso
operator[]      Acceso a un elemento
at              Acceso a un elemento
front           Acceso al primero de los elementos
back            Acceso al ultimo de los elementos
data

  std::array<int,4> myarray = {10,20,30,40};
    std::cout << myarray[1];//muestra 10
    std::cout << myarray.at(2);// muestra 20
    std::cout << myarray.front();// muestra 10
    std::cout << myaray.back(;// muestra 40
```

Acciones y funciones para vectores

BusqSecEnVector

(Dato V: Tvector; Dato N: Entero; Dato Clave:Tinfo; Dato_resultado Posic: Entero): una accion

Usar este algoritmo si alguna de las otras búsquedas en vectores mas eficientes no son posibles, recordando que búsqueda directa tiene eficiencia 1, búsqueda binaria es logarítmica y búsqueda secuencial es de orden N

```
 PRE:    V: Vector en el que se debe buscar
         Clave : Valor Buscado
         N: Tamaño lógico del vector
         U: = N -1 posición del ultimo, uno menos que el tamaño del vector
POS:     Posic: Posición donde se encuentra la clave, -1 si no esta.
LEXICO
j : Entero;
ALGORITMO
         Posic = 0;
         j = 0; //Pone el indice en la primera posición para recorrer el vector//
         MIENTRAS (j  <=  MAX_FIL  y  j <= U  y  V[j] <> Clave) HACER
                  Inc (j) //Incrementa el indice para avanzar en la estructura//
         FIN_MIENTRAS;
         SI (j > N)
         ENTONCES
                  Posic = -1 // No encontró la clave buscada
         SI_NO
                  Posic = j // Encontró la clave en la posición de índice j
FIN_SI;

FIN. // Búsqueda secuencial En Vector
```

Controla No superar el tamaño fisico del vector j<= MAX_FIL

No leer mas alla del ultimo elemento logico cargado j <= N

BusqMaxEnVector

(Dato V: Tvector; Dato N: Entero; Dato_resultado Maximo :Tinfo; Dato_resultado Posic: Entero): una acccion

```
PRE:     V: Vector en el que se debe buscar (sin orden)
         N : Tamaño lógico del vector
         U = N-1 Posicion del ultimo elemento
POS:     Posic: Posición donde se encuentra el máximo
Maximo : Valor máximo del vector.
LEXICO
j : Entero;
ALGORITMO
         Posic = 0;
         Maximo = V[1];
         PARA j [1, U] HACER
                  SI (v[j] > Maximo)
                  ENTONCES
                           Posic = j;
                           Maximo = v[j];
                  FIN_SI;
         FIN_PARA;
```

Supone que el maximo es el primer valor del vector por lo que le asigna ese valor a maximo y la posición 0 a la posición del maximo. Al haber leido solo un elemento supone ese como maximo

Recorre ahora las restantes posiciones del vector, a partir de la segunda y lcada vez que el valor leido supera al maximo contiene ese valor como maximo y el indice actual como posición del maximo

FIN. // Búsqueda máximo En Vector

BusqMinDistCeroEnVector

(Dato V: Tvector; Dato N: Entero; Dato_resultado Minimo :Tinfo; Dato_resultado Posic: Entero): una acccion

PRE: V: Vector en el que se debe buscar (sin orden)
N : Tamaño lógico del vector, existe al menos un valor <> de cero
U = N-1 Posicion del ultimo elemento

POS: Posic: Posición donde se encuentra el minimo distinto de cero

Minimo : Valor minimo distinto de cero del vector.

```
LEXICO
i,j  : Entero;
ALGORITMO
        //
        J = o;
Mientras (J<=U) Y (V[j] = 0) Hacer
        Incrementar[j];
Posic = J;
        Minimo = V[j];
        PARA j [Posic.+1 , N] HACER
                SI (v[j]<> 0 Y v[j] < Minimo)
                ENTONCES
                        Posic = j;
                        Minimo = v[j];
                FIN_SI;
        FIN_PARA;
```

Recorre el vector hasta encontrar el primero distinto de cero. Al encontrarlo supone ese valor como minimo y el valor del indice como posición del minimo

Recorre el vector desde la posición inmediata siguiente hasta la ultima desplazando el minimo solo si el valor es distinto de cero y, ademas, menor aue el minimo

FIN. // Búsqueda minimo distinto de cero En Vector

BusqMaxySiguienteEnVector

(Dato V: Tvector; Dato N: Entero; Dato_resultado Maximo :Tinfo; Dato_resultado Posic: Entero, Dato_resultado Segundo :Tinfo; Dato_resultado PosicSegundo: Entero): una accion

PRE: V: Vector en el que se debe buscar
N : Tamaño lógico del vector mayor o gual a 2
U = N - 1

POS: Posic: Posición donde se encuentra el máximo, PosicSegundo: Posición donde se encuentra el siguiente al máximo

Maximo : Valor máximo del vector. Segundo : Valor del siguiente al máximo del vector

```
LEXICO
j : Entero;
ALGORITMO
        SI V[0] > V[1]
ENTONCES
        Posic = 0;
        Maximo = V[0];
        PosicSegund = 1;
        Segundo = V[1];
```

Se tiene como precondicion que al menos hay dos valores. Se verifica el valor que esta en la primera posición y se lo compara con el que esta en la segunda posición, en el caso de ser mayor, el maximo es ese valor, posición del máximo es cero, el segundo el valor que esta en segundo lugar y posición del segundo es 1. En caso contrario se establece como maximo el valor de la segunda

```
SINO
        Posic = 1;
        Maximo = V[1];
        PosicSegund =0;
        Segundo = V[0];

FIN_SI

PARA j [2, U] HACER
        SI (v[j] > Maximo)
        ENTONCES
                Segundo = Maximo;
                PosicSegundo = Posic;
                Posic = j;
                Maximo = v[j];
        SINO
                SI Maximo>Segundo
                ENTONCES
                        Segundo = V[j];
                        PosicSegundo = j
                FIN_SI
        FIN_SI;
FIN_PARA;

FIN. // Búsqueda máximo En Vector
```

Se verifica luego desde la tercera posición hasta el final. En el caso que el nuevo valor sea mayor que el maximo, se debe contener el anterior maximo en el segundo y al maximo se le asigna el nuevo valor. Cosa similar hay que hacer con las posiciones. Si esto no ocurriera se debe verificar si el

CargaSinRepetirEnVectorV1

(Dato_Resultado V: Tvector; Dato_Resultado N: Entero; Dato Clave:Tinfo; Dato_resultado Posic: Entero; Dato_resultado Enc : Booleano): una accion

Utilizar este algoritmo si la cantidad de claves diferentes es fija, se dispone de memoria suficiente como para almacenar el vector y la clave no es posicional(es decir clave e índice no se corresponden directamente con posición única y predecible.Si la posición fuera unica y predecible la busqueda debe ser directa

PRE: V: Vector en el que se debe buscar
Clave : Valor Buscado
N : Tamaño lógico del vector
U = N-1 posicion del ultimo elemento

POS: Posic: Posición donde se encuentra la clave, o donde lo inserta si no esta. Retorna -1 (menos 1) en caso que el vector esta completo y no lo encuentra
Enc : Retorna True si estaba y False si lo inserto con esta invocación
Carga vector sin orden

```
LEXICO
j : Entero;
ALGORITMO/
        Posic = -1;
        J = 0;
        MIENTRAS (j  <=  MAX_FIL  y  j <= U  y  V[j] <> Clave) HACER
                Inc (j)
```

Controla No superar el tamaño fisico del vector j<= MAX_FIL

No leer mas alla del ultimo elemento logico cargado j <= N

```
        FIN_MIENTRAS;
        SI j > MAX_FIL
        ENTONCES
                Posic = -1
        SI_NO
                Posic = j:
                SI (j > N)
                ENTONCES
                        Enc =FALSE; // No encontró la clave buscada
                        Inc(N);
                        V[N] = Clave;
                SI_NO
                        Enc = True // Encontró la clave en la posición de índice j
                FIN_SI;
        FIN_SI
FIN. // Carga sin repetir en vector
```

Si debio superar el tamaño fisico maximo del vector no pudo cargarlo y retorna cero como señal de error

Si encontro un dato o lo debe cargar esto es en el indice j por lo que a pos se se asigna ese valor. En el caso que j sea mayor que n significa que recorrio los n elemntos cargados del vector, tiene etpacio por lo que debe cargar el nuevo en la posición j. En este caso, y al haber un elemento nuevo debe incrementar n que es el

BusquedaBinariaEnVectorV1(Dato V: Tvector; Dato N: Entero; Dato Clave:Tinfo; Dato_resultado Posic: Entero; Dato_resultado Pri : Entero): una accion

Utilizar este algoritmo si los datos en el vector están ordenados por un campo clave y se busca por ese campo. Debe tenerse en cuenta que si la clave es posicional se deberá utilizar búsqueda directa ya que la diferencia en eficiencia esta dada entre 1, para la búsqueda directa y log_2N para la binaria

PRE: V: Vector en el que se debe buscar con clave sin repetir
Clave : Valor Buscado
N : Tamaño lógico del vector

POS: Posic: Posición donde se encuentra la clave, o -1 (menos 1) si no esta
Pri : Retorna la posición del limite inferior

```
LEXICO
j : Entero;
u,m : Entero;
ALGORITMO
        Posic = -1;
        Pri = ;
        U = N-1;
        MIENTRAS (Pri < = U  y  Pos = -1) HACER
                M = (Pri + U ) / 2
                SI V[M] = Clave
                ENTONCES
                        Posic = M;
                SI_NO
                        SI Clave > V[M]
                        ENTONCES
                                Pri = M+1
                        SI_NO
                                U = M – 1
                        FIN_SI
```

Establece valores para las posiciones de los elementos del vector, Pri contiene el indice del primero, es decir el valor 1, U el indice del ultimo elemento logicio, es decir N. Ademas se coloca en Posic. El valor cero, utilizando este valor como bandera para salir del ciclo cuando encuentar el valor buscado

Permanece en el ciclo mientras no encuentre lo buscado, al encontrarlo le asigna a pos el indice donde lo encontro, como es un valor > que cero hace false la expresión logica y sale del ciclo. Si no lo encuentra, y para evirtar ciclo infinito verifica que el primero no tome un valor mayor que el ultimo. Si eso ocurre es que el dato buscado no esta y se debe

Si el dato buscado lo encuentra le asigna a posición el indice para salir. Si no lo encuentra verifica si esmayor el dato buscabo a lo que se encuentra revisa en la mitad de los mayores por lo que le asigna al primero el indice siguiente al de la mitad dado que alli no estab y vuelve a dividir el conjunto de datos en la mitas, de ser menor pone como tome

```
            FIN_SI
        FIN_MIENTRAS;

FIN. // Búsqueda binaria en vector
```

BusquedaBinariaEnVectorV2

(Dato V: Tvector; Dato N: Entero; Dato Clave:Tinfo; Dato_resultado Posic: Entero; Dato_resultado Pri : Entero): una acccion

PRE: V: Vector en el que se debe buscar clave puede estar repetida
Clave : Valor Buscado
N : Tamaño lógico del vector

POS: Posic: Posición donde se encuentra la primera ocurrencia de la clave.
0 (cero) si no esta.
Pri : Retorna la posición del limite inferior

```
LEXICO
j : Entero;
u,m : Entero;
ALGORITMO
        Posic = -1;
        Pri = 1;
        U = N-1;
        MIENTRAS (Pri <  U ) HACER
                M = (Pri + U ) / 2
                SI V[M] = Clave
                ENTONCES
                        Posic = M;
                        Pri = M;
                SI_NO
                        SI Clave > V[M]
                        ENTONCES
                                Pri = M+1
                        SI_NO
                                U = M – 1
                        FIN_SI
                FIN_SI
        FIN_MIENTRAS;

FIN. // Búsqueda binaria en vector
```

La busqueda es bastante parecida a lo desarrollado anteriormente, pero en pos debe tener la primera aparicion de la clave buscada que puede repetirse.

En la busqueda anterior utilizabamos esta pos como bandera, para saber cuando Sali si lo encontro. En este caso si lo utilizamos con el mismo proposito saldria cuando encuentra un valor oincidente con la clave que no necesariamente es el primero, por lo que esa condicion se elimina. Al encontrarlo en m se le asigna ese valoe a pos, alli seguro esta. No sabemos si mas arriba vuelve a estar por lo que se asigna tambien esa posición al ultimo para seguir iterando y ver si lo vuelve a encontrar. Debe modificarse el operador de relacion que compara primero con ultimo para evitar un ciclo infinito, esto se hace eliminando la relacion por igual. Insisto en el concepto de los valores de retorno de la busqueda binaria. Una particularidad de los datos es que si lo que se busca no esta puede retornal en el primero la posición donde esa clave deberia estar

CargaSinRepetirEnVectorV2

(Dato_Resultado V: Tvector; Dato_Resultado N: Entero; Dato Clave:Tinfo; Dato_resultado Posic: Entero; Dato_resultado Enc : Booleano): una acccion

PRE: V: Vector en el que se debe buscar ordenado por clave
Clave : Valor Buscado
N : Tamaño lógico del vector

La busqueda es bastante parecida a lo desarrollado anteriormente, pero en pos debe tener la primera aparicion de la clave buscada que puede repetirse.
En la busqueda anterior utilizabamos esta pos como bandera, para saber cuando Sali si lo encontro. En este caso si lo utilizamos con el mismo proposito saldria cuando encuentra un valor oincidente con la clave que no necesariamente es el primero, por lo que esa condicion se elimina. Al encontrarlo en m se le asigna ese valoe a pos, alli seguro esta. No sabemos si mas arriba vuelve a estar por lo que se asigna tambien esa posición al ultimo para seguir iterando y ver si lo vuelve a encontrar. Debe modificarse el operador de relacion que compara primero con ultimo para evitar un ciclo infinito, esto se hace eliminando la relacion por igual. Insisto en el concepto de los valores de retorno de la busqueda binaria. Una particularidad de los datos es que si lo que se busca no esta puede retornal en el primero la posición donde esa clave deberiaestar

POS: Posic: Posición donde se encuentra la clave, o donde lo inserta si no esta. Retorna -1 (menos 1) en caso que el vector esta completo y no lo encuentra
Enc : Retorna True si estaba y False si lo inserto con esta invocación
Carga vector Ordenado

Al estar el vector ordenadola busqueda puede ser binaria, si lo encuentra retorna en posición un valor mayor a cero. Si no lo encuentra el valor de posición sera -1. En este caso, se conoce que en pri es en la posición donde este valor debe estar

```
LEXICO
j : Entero;
U = N-1;
ALGORITMO
        Enc = True;
        BusquedaBinariaEnVectorV(V; N; Clave; Posic; Pri)
        SI (Posic = -1)
        ENTONCES
                Enc = False ;
                Posic = Pri;
                PARA j [U, Pri](-) HACER
                        V[j+1] = V[j];
                FIN_PARA;
                V[Pri] = Clave;
                Inc(N);
        FIN_SI
FIN. // Carga sin repetir en vector Versión 2. con vector ordenado
```

Se produce un desplazamiento de los valores desde el ultimo hasta el valor de pri corriendolos un lugar para poder insertar en la posición pri el nuevo valos. Al pasar por aquí se inserto un nuevo elemento por lo que n, que contiene la cantidad de elementos del vector debe incrementarse en uno

OrdenarVectorBurbuja

(Dato_Resultado V: Tvector; Dato N: Entero): una acccion

Pre: V: Vector en el que se debe ordenar, se supone dato simple
N : Tamaño lógico del vector
U = N-1 // posición del ultimo elemento del vector.

POS: Vector ordenado por clave creciente

Usar este algoritmo cuando los datos contenidos en un vector deben ser ordenados. Se podría por ejemplo cargar los datos de un archivo al vector, ordenar el vector recorrerlo y generar la estructura ordenada. Para esto la cantidad de elementos del archivo debe ser conocida y se debe disponer de memoria suficiente como para almacenar los datos. Una alternativa, si la memoria no alcanza para almacenar todos los datos podría ser guardar la clave de ordenamiento y la referencia donde encontrar los datos, por ejemplo, la posición en el archivo.

La idea general es ir desarrollando pasos sucesivos en cada uno de los cuales ir dejando el mayor de los elementos en el último lugar. En el primer paso se coloca el mayor en la ultima posición, en el paso siguiente se coloca el que le sigue sobre ese y asi hasta que queden dos elemntos. En ese caso al acomodar el segundo el otro queda acomodado el primer ciclo cuenta los pasos, son uno menos que la cantidad de elementos porque el ultimo paso permite acomodar 2 elementos, por eso el ciclo se hace entre 1 y U – 1 siendo n la cantidad de elementos

```
LEXICO
I,J, : Entero;
```

```
Aux : Tinfo;
ALGORITMO
        PARA i [1, U - 1] HACER
                PARA j [1, U - i] HACER
                 SI (v[j] > v[j + 1])
                   ENTONCES
                     Aux = v[j];
                     V[j] = v[j + 1];
                     V[j + 1] = Aux;
                 FIN_SI;
                FIN_PARA;
        FIN_PARA;
FIN
```

Para poder colocar el mayor al final es necesario hacer comparaciones. Se compara el primero con el segundo y si corresponde se intercambian. Asi hasta llegar al ante ultimo eleemento que se lo compara con el últim.

Al ir recorriendo los distintos pasos, y dado que en cada uno se acomoda un nuevo elemento corresponde hacer una comparación menos en cada avance, como los pasos los recorremos con i, las comparaciones seran U – i. disminuye en 1 en cada paso

OrdenarVectorBurbujaMejorado

(Dato_Resultado V: Tvector; Dato N: Entero): una acccion
PRE: V: Vector en el que se debe ordenar, se supone dato simple
N : Tamaño lógico del vector
U = N - 1
POS: Vector ordenado por clave creciente

```
LEXICO
I,J, : Entero;
Aux : Tinfo;
Ord : Boolean;
ALGORITMO
        I = 0;
REPETIR
        Inc(i);
                Ord = TRUE;
                PARA j [1, U - i] HACER
                        SI (v[j] > v[j + 1])
                        ENTONCES
                                Ord = False;
Aux = v[j];
                                V[j] = v[j + 1];
                                V[j + 1] = Aux;
                        FIN_SI;
                FIN_PARA;
        HASTA ( Ord   o I = U – 1); //si esta ordenado o llego al final
FIN
```

El algoritmo es similar al anterior, solo que el ciclo de repetición externo no lo hace si es que tiene la certeza, en el paso anterior que los datos ya estan ordenados. Es por eso que agrega una bandera para verificar si ya esta ordenado y cambia el cilo exactp por un ciclo pos condicional. Es decir reemplaza la composición para por la composición repetir hasta

OrdenarVectorInserion

(Dato_Resultado V: Tvector; Dato N: Entero): una accion
PRE: V: Vector en el que se debe ordenar, se supone dato simple
N : Tamaño lógico del vector
POS: Vector ordenado por clave creciente

Este algoritmo consta de los siguientes pasos
El primer elemento A[0] se lo considera ordenado; es decir se considera el array con un solo elemento.
Se inserta A[1] en la posicion correcta, delante o detras de A[0] segun sea mayor o menor.
Por cada iteracion, d i desde i=1 hasta n-1, se explora la sublista desde A[i-1] hasta A[0],buscando la posicion correcta de la insercion ; a la vez se mueve hacia abajouna posicion todos los elementos mayores que el elemento a insertar A[i] para dejar vacia la posicion.
Insertar el elemento en l posicion correcta.

```
LEXICO
I,J, : Entero;
Aux : Tinfo;

ALGORITMO
        PARA I[1..N-1]
                J = I;
                Aux = A[i] ;
                MIENTRAS (J >0 Y AUX < A[J - 1])HACER
                        A[J] = A[J - 1] ;
                        Dec(J) ;
                FIN MIENTRAS ;
        A[J] = Aux ;
        FIN PARA;
FIN
```

OrdenarVectorShell

(Dato_Resultado V: Tvector; Dato N: Entero): una acccion
PRE: V: Vector en el que se debe ordenar, se supone dato simple
N : Tamaño lógico del vector
POS: Vector ordenado por clave creciente
Este algoritmo consta de los siguientes pasos
Dividir la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos en n/2.
Analizar cada grupo por separado comparandolas parejas de elementos, y si no estan ordenados, se intercambian .
Se divide ahora la lista en la mitad n/4, con un incremento tambien en n/4 y nuevamente se clasifica cada grupo por separado.
Se sigue dividiendo la lista en la mitad de grupos que en el paso anterior y se clasifica cada grupo por separado.
El algoritmo termina cuando el tamaño del salto es 1.

```
ALGORITMO
        Intervalo = n / 2;

MIENTRAS Intervalo > 0 HACER
                PARA I [Intervalo + 1 .. N] HACER
                        MIENTRAS (J > 0) HACER
```

```
K = J + Intervalo;
                                    SI (A[J] <= A[K]
                                    ENTONCES
                                            J = -1
                                    SINO
                                            Intercambio(A[J] ,A[K]) ;
                                            J = J – Intervalo ;
                                    FINSI ;
                        FIN PARA ;
                FIN MIENTRAS;
FIN.
```

CorteDeControlEnVector

(Dato V:Tvector; Dato N: Entero): una acccion

Usar este procedimiento solo si se tienen los datos agrupados por una clave común y se requiere procesar emitiendo información por cada subconjunto correspondiente a cada clave.

PRE: V: Vector en el que se debe Recorrer con corte de control
Debe tener un elemento que se repite y estar agrupado por el.
N : Tamaño lógico del vector
U = N-1

POS: Recorre agrupando por una clave

```
LEXICO
I : Entero;

ALGORITMO
        I = 0;
        Anterior = TipoInfo;
        // Inicializar contadores generales
      MIENTRAS (I<=U) Hacer
          //inicializar contadores de cada sublote
         Anterior = V[i]
          MIENTRAS (I<=U Y Anterior = V[i] HACER
                  // Ejecutar acciones del ciclo

                  I = I+1 // avanza a la siguiente posición
         FIN_MIENTRAS
         // Mostrar resultados del sublote
      FIN_MIENTRAS
      // Mostrar resultados generales
FIN
```

ApareoDeVectores

(Dato V1,v2:Tvector; Dato N1,N2: Entero): una acccion

Utilizar este procedimiento si se tiene mas de una estructura con un campo clave por el que se los debe procesar intercalado y esas estructuras están ORDENADAS por ese campo común.

PRE: V1,V2: Vectores a Recorrer mezclados o intercalados

```
        Los vectores deben estar ordenados.
          N1,N2 : Tamaño lógico de los vectores
|         U1 = N1-1; U2 = N2 – 1 //posición de los últimos elementos de V1 y V2
POS:    Muestra la totalidad de los datos con el orden de las estructuras


LEXICO
I,J : Entero;
ALGORITMO
        I = 0;
        J = 0;

      MIENTRAS (I<=U1 o J<=U2) Hacer
        SI((J > U2) o ((I<=U1) y (V1[I]<V2[J])) HACER
        ENTONCES
                Imprimir (V1[I]);
                I = I + 1;
        SINO
                Imprimir(V2[J]);
                J = J + 1;
        FIN_SI;
      FIN_MIENTRAS

FIN
```

CargaNMejoresEnVector

```
(Dato_Resultado V: Tvector; Dato_Resultado N: Entero; Dato Clave:Tinfo): una acccion
PRE:    V: Vector en el que se debe buscar e insertar los mejores
        Clave: Valor Buscado
        N: Tamaño lógico del vector
        U = N-1 //p0sicion del ultimo elemento en el vector
POS:    Vector con los N mejores sin orden
LEXICO
j : Entero;
Maximo: Tinfo
Posic: Entero;
ALGORITMO
        SI (U < MAX-FIL)
        ENTONCES
                Inc (U);
                V[U] = Clave
        SI_NO
        BusqMaxEnVector(V;  N; Maximo :Tinfo; Posic: Entero);
                SI (Clave > Maximo)
                ENTONCES
                        V[Posic] = Clave;
                FIN_SI;
        FIN_SI;
```

FIN. // Carga los N mejores en vector

Archivos

4

Las estructuras de datos, a diferencia de los datos simples tienen un único nombre para más de un dato, son divisibles en miembros más elementales y dispones de operadores de acceso.
Por su forma de creación y permanencia en memoria pueden ser estáticas (creadas en tiempo de declaración, ejemplos registro, array) o dinámicas (creadas en tiempo de ejecución, ejemplos estructuras enlazadas con asignación dinámica en memoria)
Por su persistencia pueden ser de almacenamiento físico (archivos) o temporal (array, registros).
Nos ocuparemos aquí de estructuras de almacenamiento físico, archivos, estructura de dato que se utiliza para la conservación permanente de los datos. Desde el punto de vista de la jerarquía de los datos, una computadora maneja bits, que para poder manipularlos como caracteres (digitos, letras o caracteres especiales), se agrupan en bytes. Asi como los caracteres se componen de bits, los campos pueden componerse como un conjunto de bytes. Asi pueden conformarse los registros, o srtuct en C. Asi como un registro es un conjunto de datos relacionados, un archivo es un conjunto de registros relacionados. La organización de estos registros en un archivo de acceso secuebcial o en un archivo de acceso directo.

Estructura tipo registro:

Posiciones contiguas de memoria de datos no homogéneos, cada miembro se llama campo. Para su implementación en pascal se debe:

a) Declaración y definición de un registro

```
struct  NombreDelTipo {
            tipo de dato Identificador;
            tipo de dato Identificador;
} Nombre del identificador;
```

b) Operador de Acceso.
NombreDelIdentificador.Campo1 {Acceso al miembro campo1}

c) Asignación
   i) Interna: puede ser
      (1) Estructura completa Registro1←Registro2
      (2) Campo a campo Registro.campo ← Valor
   ii) Externa
      (1) Entrada
         (a) Teclado: campo a campo Leer(Registro.campo)
         (b) Archivo Binario: por registro completo Leer(Archivo, Registro)
      (2) Salida
         (a) Monitor: Campo a campo Imprimir(Registro.campo)

(b) Archivo Binario: por registro completo Imprimir(Archivo, Registro)

Estructura tipo Archivo

Estructura de datos con almacenamiento físico en disco, persiste mas allá de la aplicación y su procesamiento es lento. Según el tipo de dato se puede diferenciar en archivo de texto (conjunto de líneas de texto, compuestas por un conjunto de caracteres, que finalizan con una marca de fin de línea y una marca de fin de la estructura, son fácilmente transportables) archivos binarios (secuencia de bytes, en general mas compactos y menos transportables).

Para trabajar con archivos en C es necesario:

- Definir el tipo de la struct en caso de corresponder
- Declarar la variable (es un puntero a un FILE -definida en stdio.h- que contiene un descriptor que vincula con un índice a la tabla de archivo abierto, este descriptor ubica el FCB -File Control Blocx- de la tabla de archivos abiertos). Consideraremos este nombre como nombre interno o lógico del archivo.
- Vincular el nombre interno con el nombre físico o externo del archivo, y abrir el archivo. En la modalidad de lectura o escritura, según corresponda.

Archivos y flujos

Al comenzar la ejecución de un programa, se abren, automáticamente, tres flujos, stdin (estándar de entrada), stdout (estándar de salida), stderr (estándar de error).

Cuando un archivo se abre, se asocia a un flujo, que proporcionan canales de comunicación, entre al archivo y el programa.

FILE * F; asocia al identificador F que contiene información para procesar un archivo.

Cada archivo que se abre debe tener un apuntador por separado declarado de tipo FILE. Para luego ser abierto, la secuencia es:

FILE * Identificador;

Identificador = fopen(“nombre externo “, “modo de apertuta)”; se establece una línea de comunicación con el archivo.

Los modos de apertura son:

Modo	Descripción
R	Reset Abre archivo de texto para lectura
Rt	Idem anterior,explicitando t:texto
W	Write Abre archivo de texto para escritura, si el archivo existe se descarta el contenido sin advertencia
Wt	Idem anterior,explicitando t:texto
Rb	Reset abre archivo binario para lectura
Wb	Write Abre archivo binario para escritura, si el archivo existe se descarta el contenido sin advertencia
+	Agrega la otra modalidad a la de apertura

Asocia al flujo

FILE * F;

F = fopen(“Alumnos”, wb+); abre para escritura (crea) el arcivo binario alumnos, y agrega lectura.

Para controlar que la apertura haya sido correcta se puede:

If ((F = fopen(“Alumnos”, wb+)) == NULL) {error(1); return 0};

Si el apuntador es NULL, el archive no se pudo abrir.

Archivos de texto:

Secuencia de líneas compuestas por cero o mas caracteres, con un fin de línea y una marca de final de archivo.

Función	Descripción
FILE *f;	Define f como puntero a FILE
f = fopen (“archivo”, “w”);	Asocia f a un flujo
f = fropen(“archivo”, “w”);	Similar anterior, si esta abierto antes lo cierra
fclose(f);	Vacía el flujo y cierra el archivo asociado
fflush(f);	Produce el vaciado de los flujos
remove(“archivo”);	El archivo ya no queda accesible
rename(“viejo”, “nuevo”);	Renombra con nuevo el viejo nombre
fprintf(f, “%d”, valor);	Escritura con formato en un flujo
fscanf(f, “%d”, &valor);	Lectura con formato desde un flujo
c = getchar();	Lee un carácter desde stdin
c = getc(f);	Lee un carácter desde el flujo
c = fgetc(f);	Igual que el anterior
ungetc (c, f);	Retorna el carácter al flujo y retrocede
putchar(c) ;	Escribe un carácter en stdin
putc(c, f);	Escribe un carácter en el flujo
fputc(c,f);	Igual al anterior
gets(s);	Lee una cadena de stdin
fgets(s, n, f);	Lee hasta n-1 carácter del flujo en s
puts(s);	Escribe una cadena en stdin
fputs(s, f);	Escribe la cadena s en el flujo
feof(f)	Retorna no cero si el indicador de fin esta activo
ferror(f);	Retorna no cero si el indicador de error esta activo
clearerr(f);	Desactiva los indicadores de error

Operaciones simples

```
FILE * AbrirArchTextoLectura( FILE * f, char nombre[]){
// Asocia el flujo f  con el archivo nombre, lo abre en modo lectura
      return fopen(nombre, “r”);
}

FILE * AbrirArchTextoEscritura( FILE * f, char nombre[]){
// Asocia el flujo f  con el archivo nombre, lo abre en modo escritura
      return fopen(nombre, “w”);
}
```

Equivalencias que pueden utilizarse entre C y la representación algorítmica para flujos de texto Siendo int c; char s[n]; float r; FILE * f;	
Representacion:	Equivalente a:
LeerCaracter(f,c)	c = fgetc(f)
LeerCadena(f,s)	fgets(s,n,f)
LeerConFormato(f,c,r,s)	fscanf(f,”%c%f%s”,& c,&r,s)
GrabarCaracter(f,c)	fputc(c,f)
GrabarCadena(f,s)	fputs(f,s)
GrabarConFormato(f,c,r,s)	fprintf(f,”%c %f %s \n”, c,r,s)

Patrones algorítmicos con archivos de texto:

Dado un archivo de texto leerlo carácter a carácter y mostrar su contenido por pantalla.

C

```
main()
{
FILE *f1, f2;
int c;
f1 = fopen("entrada", "r");
f2 = fopen("salida","w");
c = fgetc(f1);
while (!feof(f1)) {
   fputc(c, f2);
  c = fgetc(f1);
}
fclose(f1);
fclose(f2);
return 0;
}
```

Solución algorítmica

```
AbrirArchTextoLectura(f1, "entrada")
AbrirArchTextoEscritura(f2,"salida")
LeerCaracter(f1,c)
Mientras(!feof(f1))
   GrabarCaracter(f2,c)
   LeerCaracter(f1,c)
FinMientras
Cerrar(f1)
Cerrar(f2)
```

Dado un archivo de texto con líneas de no más de 40 caracteres leerlo por línea y mostrar su contenido por pantalla.

C

```
main() {
FILE *f1;
char s[10 + 1]c;
f1 = fopen("entrada", "r");
fgets(s, 10 + 1, f1);
while (!feof(f1)) {
   printf("%s/n",s);
  fgets(s, 10 + 1,f1);
}
fclose(f1);
return 0;
}
```

Solución Algorítmica

```
AbrirArchTextoLectura(f1, "entrada")
LeerCadena(f1,s)
```

```
Mientras(!feof(f1))
   Imprimir(s)
   LeerCadena(f1,s)
FinMientras
Cerrar(f1)
```

Dado un archivo de texto con valores encolumnados como indica el ejemplo mostrar su contenido por pantalla. 10 123.45 Juan

C

```
main(){
FILE *f1;
int a;
float f;
char s[10 + 1]c;
f1 = fopen("entrada", "r");
fscanf(f1,"%  %f %s",&a,&f,s);
while (!feof(f1)) {
   printf("%10d%7.2f%s\n",a,f,s);
  fscanf(f1,"%d %f %s",&a,&f,s);
}
fclose(f1);
return 0;
}
```

Solución algorítmica

```
AbrirArchTextoLectura(f1, "entrada")
LeerConFormato(f1,a,f,s)          //lectura con formato de un archivo de texto
Mientras(!feof(f1))
   Imprimir(a,f,s)
   LeerConFormato(f1,a,f,s)
FinMientras
Cerrar(f1)
```

Archivos binarios:

Para trabajar en forma sistematizada con archivos binarios (trabajamos con archivos binarios, de acceso directo, de tipo, en particular de tipo registro) se requiere definir el tipo de registro y definir un flujo.

Tenga en cuenta que en algoritmos y estructura de datos trabajamos con distintos tipos de estructuras, muchas de ellas manejan conjunto de datos del mismo tipo a los efectos de resolver los procesos batch que nos presentan distintas situaciones problemáticas de la materia. Así estudiamos Arreglos, en este caso archivos (en particular de acceso directo) y luego veremos estructuras enlazadas.

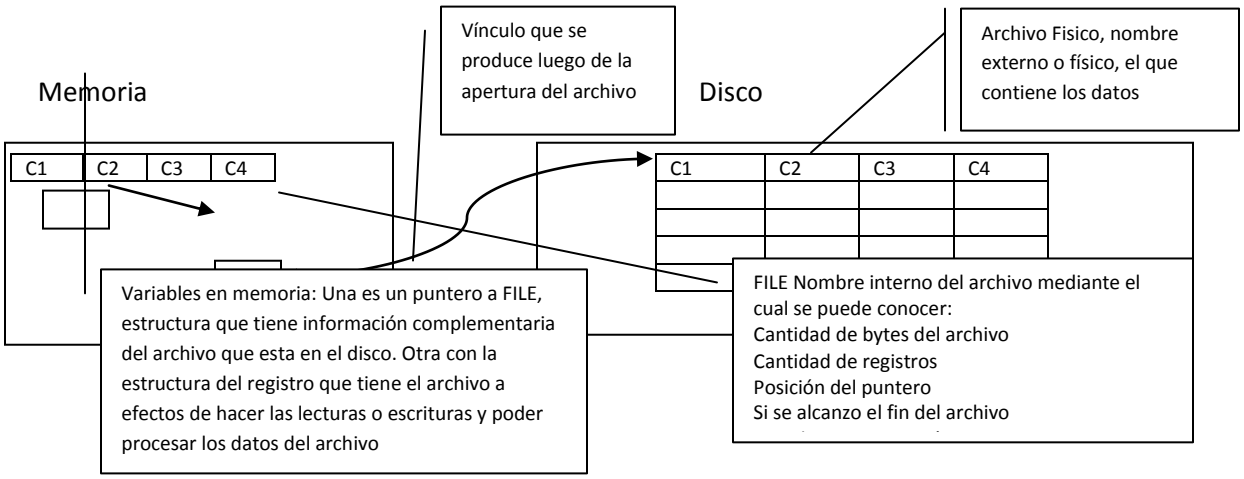
Cada una de estas estructuras tienen sus particularidades y uno de los problemas que debemos afrontar es la elección adecuada de las mismas teniendo en cuenta la situación particular que cada problema nos presente. A los efectos de hacer un análisis comparativo de las ya estudiadas, les muestro una tabla comparativa de arreglos y archivos, señalando algunas propiedades distintivas de cada una de ellas.

Análisis comparativo entre arreglos y archivos de registro de tamaño fijo		
Propiedad	Arreglo	Archivo
Tamaño Físico en Tiempo Ejecución	Fijo	Variable
Almacenamiento	Electrónico	Físico
- Persistencia	No	Si
- Procesamiento	Rápido	Lento
Búsquedas		
- Directa	Si	Si
- Binaria	Si (si esta ordenado)	Si
- Secuencial	Es posible	No recomendada
Carga		
- Secuencial	Si	Si (al final de la misma)
- Directa	Si	Solo con PUP
- Sin repetir clave	Si	No recomendada
Recorrido		
- 0..N	Si	Si
- N..0	Si	Si
- Con corte de control	Si	Si
- Con Apareo	Si	Si
- Cargando los N mejores	Si	No recomendada
Ordenamientos		
- Con PUP	Si	Si
- Método de ordenamiento	Si	No recomendado

En función de estas y otras características particulares de las estructuras de datos se deberá tomar la decisión de seleccionar la más adecuada según las características propias de la situación a resolver.

Como concepto general deberá priorizarse la eficiencia en el procesamiento, por lo que estructuras en memoria y con accesos directos ofrecen la mejor alternativa, lo cual hace a la estructura arreglo muy adecuada para este fin. Por razones varias (disponibilidad del recurso de memoria, desconocimiento a priori del tamaño fijo, necesidad que el dato persista mas allá de la aplicación, entre otras) nos vemos en la necesidad de seleccionar estructuras diferentes para adaptarnos a la solución que buscamos. El problema, aunque puede presentarse como complejo, se resuelve con ciertas facilidad ya que la decisión no es entre un conjunto muy grande de alternativas, simplemente son tres. Una ya la estudiamos, los arreglos, otra es el objeto de este apunte, los archivos, las que quedan las veremos en poco tiempo y son las estructuras enlazadas.

Observen que se pueden hacer cosas bastante similares a las que hacíamos con arreglos, recorrerlos, buscar, cargar, por lo que la lógica del procedimiento en general la tenemos, solo cambia la forma de acceso a cada miembro de la estructura. En realidad las acciones son mas limitadas dado que muchas cosas que si hacíamos con arreglos como por ejemplo búsquedas secuenciales, carga sin repetición, métodos de ordenamiento, los desestimaremos en esta estructura por lo costoso del procesamiento cuando el almacenamiento es físico (en el disco)



Definiciones y declaraciones:

En los ejemplos de archivos, dado que la estructura del registro no es relevante, se utilizará el modelo propuesto para archivos binarios y de texto, si algún patrón requiere modificación se aclarará en el mismo:

Archivo binario

Numero	Cadena	Caracter
int	char cadena[N]	char

```
struct   TipoRegistro {
         int Numero;
         char Cadena[30];
         char C;
} Registro;
FILE * F;
```

Operaciones simples

```
   FILE * AbrirArchBinLectura( FILE * f, char nombre[]){
   // Asocia el flujo f  con el archivo nombre, lo abre en modo lectura
         return fopen(nombre, "rb");
   }

   FILE * AbrirArchBinEscritura( FILE * f, char nombre[]){
   // Asocia el flujo f  con el archivo nombre, lo abre en modo escritura
         return fopen(nombre, "wb");
   }


   int  filesize( FILE * f){
   // retorna la cantidad de registros de un archivo
         TipoRegistro r;
         fseek(f, 0, SEEK_END); //pone al puntero al final del archivo
         return ftell(f)/sizeof(r); //cantidad de bytes desde el principio/tamaño del registro
   }
```

```
int  filepos( FILE * f){
// retorna el desplazamiento en registros desde el inicio
      TipoRegistro r;
      return ftell(f)/sizeof(r); //cantidad de bytes desde el principio/tamaño del registro
}

int  filepos( FILE * f){
// retorna el desplazamiento en registros desde el inicio
      TipoRegistro r;
      return ftell(f)/sizeof(r); //cantidad de bytes desde el principio/tamaño del registro
}

int  seek( FILE * f, int pos){
// Ubica el puntero en el registro pos (pos registros desplazados del inicio)
      TipoRegistro r;
      return fseek(f, pos * sizeof(r), SEEK_SET);
}

int  filepos( FILE * f){
// retorna el desplazamiento en registros desde el inicio
      TipoRegistro r;
      return ftell(f)/sizeof(r); //cantidad de bytes desde el principio/tamaño del registro
}

int LeerRegistro( FILE * f, TipoRegistro *r){
      //lee un bloque de un registro, retorna 1 si lee o EOF en caso de no poder leer.
      return fread(&r, sizeof(r) , 1, f);
}

int GrabarRegistro( FILE * f, TipoRegistro r){
      //Graba un bloque de un registro.
      return fwrite(&r, sizeof(r) , 1, f);
}

int LeerArchivo( FILE * f, TipoVector v){
      //lee un archivo completo y lo guarda en un vector.
      return fread(v, sizeof(v[0]) , filesize(f), f);
}
```

Función	Descripción
FILE *f;	Define f como puntero a FILE
f = fopen (“archivo”, “wb”);	Asocia f a un flujo
f = fropen(“archivo”, “wb”);	Similar anterior, si esta abierto antes lo cierra
fclose(f);	Vacía el flujo y cierra el archivo asociado
fflush(f);	Produce el vaciado de los flujos

remove(“archivo”);	El archivo ya no queda accesible
rename(“viejo”, “nuevo”);	Renombra con nuevo el viejo nombre
sizeof(tipo)	Retorna el tamaño de un tipo o identificador
SEEK_CUR	Constante asociada a fseek (lugar actual)
SEEK_END	Constante asociada a fseek (desde el final)
SEEK_SET	Constante asociada a fseek (desde el inicio)
size_t fread(&r, tam,cant, f)	Lee cant bloques de tamaño tam del flujo f
size_t fwrite(&r,tam,cant,f)	Graba cant bloques de tamaño tam del flujo f
fgetpos(f, pos)	Almacena el valor actual del indicador de posicion
fsetpos(f,pos)	Define el indicador de posicion del archive en pos
ftell(f)	El valor actual del indicador de posición del archivo
fseek(f, cant, desde)	Define indicador de posicion a partir de una posicion.

ConvertirBinario_Texto(Dato_Resultado B: TipoArchivo; Dato_resultado T: Texto): una acción

Usar este algoritmo para convertir un archivo binario a uno de texto, teniendo en cuenta la transportabilidad de los archivos de texto estas acciones pueden ser necesarias

```
PRE:    B: Archivo binario EXISTENTE
        T: Archivo de texto a crear
        R: TipoRegistro {Variable local para las lecturas de los registros}
POS:    Crea el archivo de texto con el formato requerido y encolumnado correctamente.
LEXICO
 TipoRegistro R;
ALGORITMO
        AbrirArcBinarioLectura(B);
        AbrirArchTextoEscritura(T);
        MIENTRAS (! feof(B) ) HACER
                LeerRegistro(B,R); {lee de B un registro completo y lo almacena en memoria en R}
                GrabarConFormato(T,r.numero, r.cadena,r.caracter);
                {graba en el archivo B datos del registro que está en memoria respetando la
                máscara de salida, vea que en texto lo hace campo a campo y en binario por
                registro completo}
        FIN_MIENTRAS
        Cerrar(B);
        Cerrar(T);
FIN. // Convertir archivo binario a texto
```

ConvertirTexto_Binario(Dato_Resultado B: TipoArchivo; Dato_resultado T: Texto): una acción

```
PRE:    B: Archivo binario a crear
        T: Archivo de texto Existente
        R: TipoRegistro {Variable local para las lecturas de los registros}
POS:    Crea el archivo de binario a partir de uno de texto con formato conocido.
LEXICO
R : TipoRegistro;
ALGORITMO
        AbrirArchBinarioEscritura(B);
        AbrirArchTextoLectura(T);
```

```
        MIENTRAS (! feof(T) ) HACER
                LeerConFormato(T,r.numero, r.cadena:30,r.caracter);
                {lee desde el archivo B  los datos y los lleva a un registro que está en memoria, vea
                que en texto lo hace campo}
                GrabarRegistro(B,R); {Graba en el archivo B el registro R completo}
        FIN_MIENTRAS
        Cerrar(T);
        Cerrar(B);
FIN. // Convertir archivo binario a texto
```

AgregarRegistrosABinarioNuevo(Dato_Resultado B: TipoArchivo): una acción

```
PRE:    B: Archivo binario a crear al que se le deben agregar registros
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Crea el archivo de binario y le agrega reagistros, r.numero>0 es la condición del ciclo.
LEXICO
R : TipoRegistro;
ALGORITMO
        AbrirArchBinarioEscritura(B);
        Leer(r.numero); {lee desde el teclado el valor del campo numero del registro}
        MIENTRAS (r.numero > 0) HACER {analiza la expresión lógica propuesta}
                Leer(r.cadena,r.caracter); {lee del teclado el resto de los campos}
                GrabarRegistro(B,R); {Graba en el archivo B el registro R completo}
                {vea que  la lectura de un registro la hicimos campo a campo al leer  desde un flujo
                de text. En cambio leimos el registro completo al hacerlo desde un archivo binario.
                Para el caso de la escritura el concepto es el mismo}
                Leer(r.numero); {lee desde el teclado el valor del proximo registro}
        FIN_MIENTRAS
        Cerrar(B);
FIN. // Agregar Registros
```

AgregarRegistrosABinarioExistente(Dato_Resultado B: TipoArchivo): una acción

```
PRE:    B: Archivo binario existente al que se le deben agregar registros
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Crea el archivo de binario y le agrega reagistros, r.numero>0 es la condición del ciclo.
LEXICO
R : TipoRegistro;
ALGORITMO
        AbrirArchBinarioLectura(B);
        Leer(r.numero);
        seek(B, filesize(B));{pone el puntero al final del archivo, posición donde se agregan}
        //se usaron las funciones de biblioteca desarrolladas anteriormente
        MIENTRAS (r.numero > 0) HACER {todo lo siguiente es similar al patrón anterior}
                Leer(r.cadena,r.caracter);
                GrabarRegistro(B,R);
                Leer(r.numero);
        FIN_MIENTRAS
        Cerrar(B);
FIN. // Agregar Registros
```

RecorrerBinarioV1(Dato_Resultado B: TipoArchivo): una acción
Recorre un archivo completo con un ciclo de repetición exacto, dado que filesize del archivo indica la cantidad de registros, si se lee con un ciclo exacto una cantidad de veces igual a filesize y por cada iterracion se lee un registro se leerían todos los registros del archivo

```
PRE:    B: Archivo binario existente que se desea imprimir
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Muestra el contenido completo del archivo por pantalla.
LEXICO
R : TipoRegistro; {para la lectura}
I  : Entero;
N: Entero
ALGORITMO
        AbrirArchBinarioLectura(B);
        N = filesize(B); {Contiene en N la cantidad de registros }

        PARA [I = 1 .. N] HACER {Itera tantas veces como cantidad de registros}
                LeerRegistro(B,R); {por cada iteración lee un registro diferente}
                Imprimir(R.numero:8,R,cadena:30, R.caracter);{muestra por pantalla}
        FIN_PARA
        Cerrar(B);
FIN. // Recorrido con ciclo exacto
```

RecorrerBinarioV2(Dato_Resultado B: TipoArchivo): una acción
Recorre un archivo completo con un ciclo de repetición mientras haya datos, es decir mientras fin de archivo sea falso, con una lectura anticipada.

```
PRE:    B: Archivo binario existente que se desea imprimir
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Muestra el contenido completo del archivo por pantalla.

LEXICO
R : TipoRegistro; {para la lectura}
ALGORITMO
        AbrirArchBinarioLectura(B);
        LeerRegistro(B,R);
        MIENTRAS (! feof(B)) HACER {Itera mientras haya datos}
                Imprimir(R.numero,R,cadena, R.caracter);{muestra por pantalla}
                Leer(B,R); {por cada iteración lee un registro diferente}

        FIN_MIENTRAS
        Cerrar(B);
FIN. // Recorrido con ciclo no exacto
```

RecorrerBinarioV3(Dato_Resultado B: TipoArchivo): una acción
Recorre un archivo completo con una lectura como condición del ciclo mientras.

```
PRE:    B: Archivo binario existente que se desea imprimir
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Muestra el contenido completo del archivo por pantalla.
LEXICO
```

R : TipoRegistro; {para la lectura}
F : Boolean {valor centinela que indica si se pudo leer el registro}
ALGORITMO

MIENTRAS (LeerRegistro(B,R) != feof(B)}
Imprimir(R.numero:8,R,cadena:30, R.caracter);*{muestra por pantalla}*
FIN_MIENTRAS
Cerrar(B);
FIN. //

CorteControlBinario(Dato_Resultado B: TipoArchivo): una acción
Recorre un archivo completo con corte de Control.
PRE: B: Archivo binario existente, ordenado, con una clave que se repite
R: TipoRegistro {Variable local para las lecturas de los registros}
POS: Muestra el contenido completo del archivo por pantalla agrupado por clave común.
Se hace una lectura anticipada para ver si hay datos. Se ingresa en un ciclo de repetición mientras haya datos, eso lo controla la variable F. Aquí se toma en una variable la clave de control. Se ingresa a un ciclo interno mientras haya datos y la clave leída sea igual al valor que se está controlando. Cuando se lee un registro cuyo valor no es igual al que se venía leyendo se produce un corte en el control que se estaba ejerciendo. Al salir del ciclo interno se verifica si sigue habiendo datos, si aún quedan, se toma la nueva clave como elemento de control y se sigue recorriendo, si no hubiera más se termina el proceso completo
LEXICO
R : TipoRegistro; {para la lectura}
Anterior : Dato del tipo de la clave por la que se debe agrupar
ALGORITMO
AbrirArchBinarioLectura(B);
LeerRegistro (B,R)
{*Inicializacion variables generales si corresponde*}
MIENTRAS (! feof (F)) HACER {*Itera mientras haya datos, es decir F = Falso*}
Anterior ← r.clave {*conserva en anterior la clave por la cual agrupar*}
{*inicialización de variables de cada grupo*}
MIENTRAS((! feof(F)) Y (Anterior = R.Clave) HACER
{*recorre en ciclo interno mientras haya datos y la clave sea la misma*}
{EJECUTAR LAS ACCIONES DE CADA REGISTRO}
LeerRegistro (B,R)
FIN_MIENTRAS
{*Acciones generales, si corresponden, de cada grupo*}
FIN_MIENTRAS
{*Acciones generales, si corresponde del total de los datos*}
Cerrar(B);
FIN. // *Recorrido con corte de control*
ApareoBinarioV1(Dato_Resultado A, B, C: TipoArchivo): una acción
Requiere al menos dos estructuras, con al menos un campo en común y se requiere procesarlos simultáneamente, intercalándolos por la clave común.
Se suponen los archivos ordenados crecientes, en caso de que el orden sea decreciente solo habrá que modificar los operadores de relación.
PRE: A, B: Archivos binarios existente, ordenado

C: Archivo Binario a crear
Ra,Rb: TipoRegistro {Variables locales para las lecturas de los registros}
POS: Intercala dos archivos manteniendo el orden de las claves

La primera versión presenta tres ciclos de repetición. Un primer ciclo mientras haya datos en ambos archivos y a continuación, cuando uno de ellos se termina, hacer un ciclo de repetición hasta agotar el que no se agoto. Como no se sabe a priori cual de los dos se agotara, se hacen los ciclos para ambos, desde luego que uno de ellos no se ejecutara. Mientras hay datos en ambos se requiere saber cual tiene la clave menor para procesarlo primero. Al agotarse uno de los archivos el otro archivo se lo procesa directamente sin necesidad de hacer ninguna comparación.

```
LEXICO
R a. Rb: TipoRegistro; {para la lectura}
F a, Fb: Boolean {valor centinela que indica si se pudo leer el registro}
ALGORITMO
    AbrirArchBinarioLectura(A);
    AbrirArchBinarioLectura(B);
    AbrirArchBinarioEscritura(C);
    LeerRegistro (A,Ra)
    LeerRegistro (A,Ra)
    {Inicializacion de variables generales}

    MIENTRAS ((! feof (A)) Y (!feof(B))) HACER { mientras haya datos en ambos archivos}
        SI (Ra.clave < Rb.clave) {si la clave del registro a es menor lo procesa y avanza}
        ENTONCES
            GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
            LeerRegistro (A,Ra)
        SI_NO
            Grabar(C, Rb) {Procesa el registro de B y avanza}
            LeerRegistro (B,Rb)
        FIN_SI
    FIN_MIENTRAS
    {agotar los que no se agotaron}
    MIENTRAS (!feof (A) ) HACER { agota A si es el que no termino}
        GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
        LeerRegistro (A,Ra)
    FIN_MIENTRAS

    MIENTRAS (!feof(B) ) HACER { agota B si es el que no termino}
        GrabarRegistro(C, Rb) {Procesa el registro de A y avanza}
        LeerRegistro (A,Rb)
    FIN_MIENTRAS
    Cerrar(A); Cerrar(B);     Cerrar(C);
FIN. // Recorrido apareo
```

ApareoBinarioV2(Dato_Resultado A, B, C: TipoArchivo): una acción

Requiere al menos dos estructuras, con al menos un campo en común y se requiere procesarlos simultaneamnte, intercalándolos por la clave común.

PRE: A, B: Archivos binarios existente, ordenado
C: Archivo Binario a crear
Ra,Rb: TipoRegistro {Variables locales para las lecturas de los registros}

POS: Intercala dos archivos manteniendo el orden de las claves

La segunda versión presenta un solo ciclo de repetició, mientras haya datos en alguno de los archivos. La expresión lógica de selección es un poco mas compleja, se coloca de a cuando b no tiene (Fb es verdadera) o cuando teniendo en ambos (Fa y Fb deben ser falsos) la clave de a es menor

```
LEXICO
R a. Rb: TipoRegistro; {para la lectura}
ALGORITMO
        AbrirBinarioLectura(A);
        AbrirBinarioLectura(B);
        AbrirBinarioEscritura(C);
        LeerRegistro (A,Ra)
        LeerRegistro (B,Rb)
        MIENTRAS ((! feof (A)) O (!feof(B))) HACER { mientras haya datos en algún  archivo}
                SI ((feof(B) O (( ! feof(A) Y(Ra.clave < Rb.clave)))
                {no hay datos en B, del otro lado del O se sabe que B tiene datos, como no se
                conoce si A los tiene,  se verifica, y si  hay también datos en a, habiendo en ambos
                se pregunta si el de a es menor en ese caso se procesa, es conceptualmente igual a
                la versión anterior aunque la expresión lógica parece un poco más compleja}
                ENTONCES
                        GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
                        LeerRegistro (A,Ra)
                SI_NO
                        GrabarRegistro(C, Rb) {Procesa el registro de B y avanza}
                FIN_SI
FIN_MIENTRAS
```

ApareoBinarioPorDosClaves(Dato_Resultado A, B, C: TipoArchivo): una acción

Requiere al menos dos estructuras, con al menos un campo en común y se requiere procesarlos simultaneamnte, intercalándolos por la clave común.

Se suponen los archivos ordenados crecientes, en este caso por dos campos de ordenamiento

PRE: A, B, : Archivos binarios existente, ordenado
C: Archivo Binario a crear
Ra,Rb: TipoRegistro {Variables locales para las lecturas de los registros}

POS: Intercala dos archivos manteniendo el orden de las claves

Se utiliza el criterio de la primera versión presentada

```
LEXICO
R a. Rb: TipoRegistro; {para la lectura}
ALGORITMO
        AbrirArchBinarioLectura(A);
        AbrirArchBinarioLectura(B);
        AbrirArchBinarioEscritura(C);
        LeerRegistro (A,Ra)
        LeerRegistro (B,Rb)

        MIENTRAS ((!feof(A)) Y (!feof(B))) HACER { mientras haya datos en ambos archivos}
                SI ((Ra.clave1 < Rb.clave1)O((Ra.clave1 0 Rb.clave1)Y(Ra.clave2 <Rb.clave2)))
```

```
                {si la primera clave del registro a es menor lo procesa y avanza, también procesa
                de a si la primera clave es igual y la segunda es menor}
                ENTONCES
                        GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
                        LeerRegistro (A,Ra)
                SI_NO
                        GrabarRegistro(C, Rb) {Procesa el registro de B y avanza}
                        LeerRegistro (B,Rb)
                FIN_SI
        FIN_MIENTRAS
        {agotar los que no se agotaron}
        MIENTRAS (! Feof(A) ) HACER { agota A si es el que no termino}
                GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
                LeerRegistro (A,Ra)
        FIN_MIENTRAS

        MIENTRAS (!feof(B) ) HACER { agota B si es el que no termino}
                GrabarRegistro(C, Rb) {Procesa el registro de A y avanza}
                LeerRegistro (A,Rb)
        FIN_MIENTRAS
        Cerrar(A); Cerrar(B);Cerrar(C);
FIN. // Recorrido apareo
```

BusquedaDirectaArchivo(Dato_Resultado B: TipoArchivo; Posic: Entero, Dato_Resultado R: TipoRegistro): una accion

Utilizar este algoritmo si los datos en el archivo están ordenados por un campo clave y la posición es conocida o es una Posición Única y Predecible

PRE: B: Archivo binario en el que se debe buscar con clave sin repetir
Posic: Posición donde se encuentra la clave buscada que puede ser PUP

POS: R: el registro completo de la clave buscada

```
LEXICO
        Seek (B, Posic);{ubica el puntero en la posición conocida o en la PUP}
        LeerRegistro (B,R);{lee el registro en la posición definida}
End.
```

BusquedaBinariaArchivo(Dato_Resultado B: TipoArchivo; Dato Clave:Tinfo; Dato_resultado Posic: Entero): una accion

Utilizar este algoritmo si los datos en el archivo están ordenados por un campo clave y se busca por ese campo. Debe tenerse en cuenta que si la clave es posicional se deberá utilizar búsqueda directa ya que la diferencia en eficiencia está dada entre 1, para la búsqueda directa y $log_2 N$ para la binaria

PRE: B: Archivo binario en el que se debe buscar con clave sin repetir
Clave : Valor Buscado

POS: Posic: Posición donde se encuentra la clave, ó (-1) si no esta

LEXICO

```
j : Entero;
u,m : Entero;
ALGORITMO
        Posic = -1;
        Pri = 0;
        U = filesize(B); //la función predefinida
        MIENTRAS (Pri < = U  y  Posic = -1) HACER
                M = (Pri + U ) div 2
                Seek(B, M); // la función predefinida
                LeerRegistro(B,R)
                SI R.clave = Clave
                ENTONCES
                        Posic = M;
                SI_NO
                        SI Clave > R.clave
                        ENTONCES
                                Pri = M+1
                        SI_NO
                                U = M – 1
                        FIN_SI
                FIN_SI
        FIN_MIENTRAS;

FIN. // Búsqueda binaria en archivo
```

BusquedaBinariaArchivoV2(Dato_Resultado B: TipoArchivo; Dato Clave:Tinfo; Dato_resultado Posic: Entero): una accion

Utilizar este algoritmo si los datos en el archivo están ordenados por un campo clave, la clave se repite y se desea encontrar la primera ocurrencia de la misma en el archivo

PRE: B: Archivo binario en el que se debe buscar con clave sin repetir
Clave : Valor Buscado

POS: Posic: Posición donde se encuentra la clave, ó (-1) si no esta

```
LEXICO
j : Entero;
u,m : Entero;
ALGORITMO
        Posic = -1;
        Pri = 0;
        U = filesize(B);
        MIENTRAS (Pri <  U  ) HACER
                M = (Pri + U ) div 2
                Seek(B, M);
                LeerRegistro(B,R)
                SI R.clave = Clave
                ENTONCES
                        Posic = M;
                        U = M{la encontró, verifica en otra iteración si también esta mas arriba}
```

```
            SI_NO
                    SI Clave > R.clave
                    ENTONCES
                            Pri = M+1
                    SI_NO
                            U = M – 1
                    FIN_SI
            FIN_SI
    FIN_MIENTRAS;

FIN. // Búsqueda binaria en archivo
```

El archivo como estructura auxiliar:

Como vimos en general es necesario la utilización de estructuras auxiliares con el propósito de acumular información según alguna clave particular, ordenar alguna estructura desordenada, buscar según una clave. Las estructuras más idóneas para este tipo de procesamiento son estructuras de almacenamiento electrónico por la eficiencia que brindan por la rápida velocidad de procesamiento, aquí la elección debe orientarse hacia estructuras tipo array, que permiten búsquedas directas, binarias y hasta secuencial, o estructuras enlazadas, listas, que permiten búsqueda secuencial, eficientes todas por la eficiencia que brinda el almacenamiento electrónico. De cualquier modo, es posible utilizar estructura tipo archivo como estructura auxiliar o paralela, en algunos casos particulares que se abordan a continuación.

Los casos pueden ser:

- ✓ Generar un archivo ordenado a partir de una estructura desordenada cuando es posible hacerlo por tener una clave de Posición Única Predecible.
- ✓ Generar un archivo paralelo a un archivo ordenado, con clave de PUP para acumular.
- ✓ Generar un archivo paralelo a un archivo ordenado, sin clave de PUP para acumular.
- ✓ Generar un archivo índice, que contenga la dirección en la que se encuentra una clave para poder hacer búsqueda directa.

Como se comentó, estas mismas opciones se pueden utilizar con estructuras en memoria, son mas eficientes, solo se agregan estas alternativas como variantes conceptuales para profundizar el conocimiento de las estructuras de datos.

Para estos ejemplos se toman archivos con las siguientes estructuras

Datos.dat: Archivo de datos personales: TipoArchivoDatosPersonales

Numero	DatosPersonales	OtrosDatos
Entero	Cadena	Cadena

Transacciones.dat: Archivo de transacciones: TipoArchivoTransacciones

Numero	Compras
Entero	Real

GenerarBinarioOrdenadoPUP(Dato_Resultado A,B: TipoArchivoDatosPersonales): una accion

Utilizar este algoritmo sólo si la clave es numérica, con valor inicial y final conocido y están todos los valores intermedios. En este caso es posible generar un archivo con una posición que es única y se la puede conocer a priori. La posición que ocupará esa clave en el archivo ordenad es:

PUP = valor de la clave – Valor de la clave inicial. Es decir si decimos que los valores de la clave en los registros están entre 1 y 999, por ejemplo, la PUP = Registro.clave – 1, si los valores estuvieran comprendidos entre 30001 y 31000 la PUP = Registro.clave – 30000.

PRE: A: Archivo Binario desordenado existente. Las condiciones de la clave permiten una PUP
B : Archivo Binario a crear ordenado

POS: Genera un archivo ordenado con PUP, con el mismo registro que el sin orden

```
LEXICO
R : TipoRegistro;
PUP: Entero;
ALGORITMO
        AbrirArchBinarioLectura(A);
        AbrirArchBinarioEscritura(B);
        MIENTRAS (!feof(A) ) HACER
                LeerRegistro(A,R) {leer el registro del archivo sin orden}
                PUP = R.clave – 1 {Calcular la PUP, recordar que se debe restar el valor de la primera de las claves posibles, para el ejemplo se toma 1}
                Seek(B, PUP);{acceder a la PUP calculada, usando la función que creamos}
                GrabarRegistro(B,R);{graba el registro en la PUP, vea que no es necesario leer para conocer cuál es el contenido previo ya que sea cual fuera será reemplazado por el nuevo}
        FIN_MIENTRAS;
        Close(A); Close(B);
FIN. // Generar archivo con PUP
```

Notas:

- ✓ Paralelo a archivo ordenado para agrupar: es posible que se plantee la situación problemática siguiente: Se dispone de dos archivos, uno con los datos personales, y otro con las transacciones y se busca informar cuanto compro cada cliente. Esto requeriría agrupar por clase. Si el archivo de datos personales esta ordenado, la clave numérica y con valores comprendidos entre 1 y 1000 y están todos. Estas características, por lo visto hasta aquí, supone que en ese archivo se puede hacer una búsqueda directa por la clave ya que puede ser, por las características una PUP. Por otro lado en el archivo de transacciones pueden ocurrir varias cosas:
    - Que el archivo de transacciones tenga un único registro por compra y este ordenado. Ambos ordenados, esto puede hacerse utilizando como patrón el apareo. Te invito a que lo hagas.
    - Que el archivo de transacciones tenga varios registros por compra y este ordenado. Ambos ordenados, esto puede hacerse utilizando como patrón el apareo (o similar), en combinación con el corte de control. Te invito a que lo hagas
    - Que el archivo de transacciones tenga un único registro por compra y este desordenado. Para ordenar las compras se puede generar un archivo paralelo al de datos personales con una PUP, en cada posición un dato de tipo real. Esto se puede realizar adaptando el patrón analizado, en este caso no es necesario leer pues no se debe acumular. Te invito también a que lo hagas.
    - Que el archivo de transacciones tenga varios registros por compra y este desordenado. Para ordenar las compras se puede generar un archivo paralelo al de datos personales con una PUP, en cada posición un dato de tipo real. Esto se puede realizar adaptando el patrón analizado, en este caso, como se debe

acumular es necesario *APUNTAR (a la PUP con Seek), LEER (el registro completo para llevarlo a memoria, recuerde que en este caso el puntero en el archivo avanza) MODIFICAR (el dato a acumular que está en memoria, APUNTAR (como el puntero avanzo se lo debe volver a la posición anterior, la tiene en la PUP o la puede calcular con filepos(Archivo) – 1, y finalmente GRABAR(llevar el registro modificado en memoria al disco para efectivizar la modificación. Hacelo!!!*

- *Que el archivo ordenado, este ordenado pero no pueda garantizarse la PUP, en este caso la búsqueda no puede ser directa con PUP. El criterio a utilizar es el mismo, con la salvedad que para poder determinar con precisión la posocion que la clave ocupa en el archivo auxiliar se lo debe buscar previamente con búsqueda binaria en el archivo de datos personales. Se vinculan por posición, por tanto cada posición del ordenado se supone la misma en el auxiliar.*
- *Si los datos están ordenados, como vimos, no es necesario generar estructuras auxiliares. Los datos tal como están deben ser mostrados, por tanto no se los debe retener para modificar el orden, que es esto lo que hacen las estructuras auxiliares. En caso de tener que generar una estructura auxiliar, primero debe generarse esta y luego se deben recorrer los dos archivos, el de datos y el auxiliar en forma paralela, uno proveerá información personal y el otro el valor comprado.*

- ✓ Creación de archivo índice: Si se tiene un archivo con una clave que puede ser PUP y esta desordenado y se requiere, por ejemplo, mostrarlo en forma ordenada por la clave, es posible generar un archivo ordenado con una PUP y mostrarlo. Puede ocurrir que no disponga de espacio en disco como para duplicar el archivo. Solo tiene espacio para las distintas posiciones de las claves y la referencia al archivo de datos. Esto permite generar un archivo con PUP que contenga la posición que efectivamente esa clave tiene en el archivo de datos. Esto es el concepto de estructura que estamos mencionando.
- ✓ Si tenemos un archivo desordenado y se quiere generar un archivo ordenado las alternativas son múltiples
  - Si el tamaño del archivo es conocido a priori y se dispone de memoria suficiente, es posible llevar el archivo a memoria, es decir, cargarlo en un vector, ordenar el vector y luego reconstruir el archivo reordenándolo según los datos cargados en el vector.
  - Si el tamaño es conocido y el recurso total no alcanza para llevarlo a memoria, se puede cargar la clave por la cual ordenar y una referencia al archivo para poder hacer luego un acceso directo al recorrer el vector ordenado.

GenerarIndicePUP(Dato_Resultado A: TipoArchivoDatosPersonales; B:TipoArchivoEnteros): una accion

Utilizar este algoritmo sólo si la clave es numérica, con valor inicial y final conocido y están todos los valores intermedios .No se tiene espacio para duplicar el archivo .

PRE: A: Archivo Binario desordenado existente. Las condiciones de la clave permiten una PUP
B : Archivo Binario a crear ordenado con la referencia al archivo sin orden

POS: Genera un archivo ordenado con PUP, con la posición del registro en el sin orden

```
LEXICO
R : TipoRegistro;
PUP: Entero;
ALGORITMO
        AbrirBinarioLectura(A);
```

```
        AbrirBinarioEscritura(B);
        MIENTRAS (Not feof(A) ) HACER
                LeerRegistro(A,R) {leer el registro del archivo sin orden}
                PUP = R.clave – 1 {Calcular la PUP, recordar que se debe restar el valor de la primera de las claves posibles, para el ejemplo se toma 1}
                Seek(B, PUP);{acceder a la PUP calculada}
                GrabarRegistro(B,filepos(A)-1);{graba la posición del registro en el archivo de datos en la PUP del archivo indice, vea que no es necesario leer para conocer  cuál es el contenido previo ya que sea cual fuera será reemplazado por el nuevo}
        FIN_MIENTRAS;
        Close(A); Close(B);
FIN. // Generar archivo indice con PUP
```

Archivos

5

Implementaciones C C++

Operaciones sobre arrays

Antes de comenzar

Este documento resume las principales operaciones que son generalmente utilizadas para la manipulación de arrays. Además busca inducir al alumno para que descubra la necesidad de trabajar con tipos de datos genéricos, implementados con templates, y también la importancia de poder desacoplar las porciones de código que son propias de un problema, de modo tal que el algoritmo pueda ser genérico e independiente de cualquier situación particular; delegando dichas tareas en la invocación de funciones que se reciben cómo parámetros (punteros a funciones).

Agregar un elemento al final de un array

La siguiente función agrega el valor `v` al final del array `arr` e incrementa su longitud `len`.

```
void agregar(int arr[], int& len, int v)
{
      arr[len]=v;
      len++;
      return;
}
```

Recorrer y mostrar el contenido de un array

La siguiente función recorre el array `arr` mostrando por consola el valor de cada uno de sus elementos.

```
void mostrar(int arr[], int len)
{
      for(int i=0; i<len; i++){
        cout << arr[i] << endl;
      }
  return;
}
```

Determinar si un array contiene o no un determinado valor

La siguiente función permite determinar si el array `arr` contiene o no al elemento `v`; retorna la posición que `v` ocupa dentro de `arr` o un valor negativo si `arr` no contiene a `v`.

```
int buscar(int arr[], int len, int v)
{
      int i=0;
      while( i<len && arr[i]!=v ){
       i++;
      }
  return i<len?i:-1;
```

```
}
```

Eliminar el valor que se ubica en una determinada posición del array

La siguiente función elimina el valor que se encuentra en la posición `pos` del array `arr`, desplazando al i-ésimo elemento hacia la posición i-1, para todo valor de i>`pos` y i<`len`.

```
void eliminar(int arr[], int& len, int pos)
{
      for(int i=pos; i<len-1; i++ ){
       arr[i]=arr[i+1];
      }
      // decremento la longitud del array
      len--;
  return;
}
```

Insertar un valor en una determinada posición del array

La siguiente función inserta el valor `v` en la posición `pos` del array `arr`, desplazando al i-ésimo elemento hacia la posición i+1, para todo valor de i que verifique: i>=`pos` e i<`len`.

```
void insertar(int arr[], int& len, int v, int pos)
{
      for(int i=len-1; i>=pos; i--){
        arr[i+1]=arr[i];
      }
      // inserto el elemento e incremento la longitud del array
      arr[pos]=v;
      len++;
  return;
}
```

Insertar un valor respetando el orden del array

La siguiente función inserta el valor `v` en el array `arr`, en la posición que corresponda según el criterio de precedencia de los números enteros. El array debe estar ordenado o vacío.

```
int insertarOrdenado(int arr[], int& len, int v)
{
      int i=0;
      while( i<len && arr[i]<=v ){
       i++;
      }
      // inserto el elemento en la i-esima posicion del array
      insertar(arr,len,v,i); // invoco a la funcion insertar
      // retorno la posicion en donde se inserto el elemento
return i;}
```

Más adelante veremos como independizar el criterio de precedencia para lograr que la misma función sea capaz de insertar un valor respetando un criterio de precedencia diferente entre una y otra invocación.

Insetar un valor respetando el orden del array, sólo si aún no lo contiene

La siguiente función busca el valor `v` en el array `arr`; si lo encuentra entonces asigna `true` a `enc` y retorna la posición que `v` ocupa dentro de `arr`. De lo contrario asigna `false` a `enc`, inserta a `v` en `arr` respetando el orden de los números enteros y retorna la posición en la que finalmente `v` quedó ubicado.

```
int buscaEInserta(int arr[], int& len, int v, bool& enc)
{
      // busco el valor
```

```
      int pos = buscar(arr,len,v);
      // determino si lo encontre o no
      enc = pos>=0;
      // si no lo encontre entonces lo inserto ordenado
      if( !enc ){
        pos = insertarOrdenado(arr,len,v);
      }
      // retorno la posicion en donde se encontro el elemento o en donde se inserto
return pos;}
```

Templates

Los templates permiten parametrizar los tipos de datos con los que trabajan las funciones, generando de este modo verdaderas funciones genéricas.

Generalización de las funciones agregar y mostrar

```
template <typename T> void agregar(T arr[], int& len, T v)
{
      arr[len]=v;
      len++;
      return;
}
```

```
template <typename T> void mostrar(T arr[], int len)
{
      for(int i=0; i<len; i++){
        cout << arr[i];
        cout << endl;
      }
  return;
}
```

Veamos como invocar a estas funciones genéricas.

```
int main()
{
      string aStr[10];
      int lens =0;
      agregar<string>(aStr,lens,"uno");
      agregar<string>(aStr,lens,"dos");
      agregar<string>(aStr,lens,"tres");
      mostrar<string>(aStr,lens);
      int aInt[10];
      int leni =0;
      agregar<int>(aInt,leni,1);
      agregar<int>(aInt,leni,2);
      agregar<int>(aInt,leni,3);
      mostrar<int>(aInt,leni);
  return 0;
}
```

Ordenamiento

La siguiente función ordena el array `arr` de tipo `T` siempre y cuando dicho tipo especifique el criterio de precedencia de sus elementos mediante los operadores relacionales `>` y `<`. Algunos tipos (y/o clases) válidos son: `int`, `long`, `short`, `float`, `double`, `char` y `string`.

```
template <typename T> void ordenar(T arr[], int len)
{
      bool ordenado=false;
      while(!ordenado){
        ordenado = true;
        for(int i=0; i<len-1; i++){
            if( arr[i]>arr[i+1] ){
              T aux = arr[i];
              arr[i] = arr[i+1];
              arr[i+1] = aux;
              ordenado = false;
            }
        }
      }
return;
}
```

Punteros a funciones

Las funciones pueden ser pasadas cómo parámetros a otras funciones para que éstas las invoquen.
Utilizaremos esta carácteristica de los lenguajes de programación para parametrizar el criterio de precedencia que queremos que la función `ordenar` aplique al momento de comparar cada par de elementos del array `arr`.
Observemos con atención el tercer parámetro que recibe la función `ordenar`. Corresponde a una función que retorna un valor de tipo `int` y recibe dos parámetros de tipo `T`, siendo `T` un tipo de datos genérico parametrizado por el template.
La función `criterio`, que debemos desarrollar por separado, debe comparar dos elementos e1 y e2, ambos de tipo `T`, y retornar un valor: negativo, positivo o cero según se sea: e1<e2, e1>e2 o e1=e2 respectivamente.

```
template <typename T>void ordenar(T arr[], int len, int (*criterio)(T,T))
{
      bool ordenado=false;
      while(!ordenado){
        ordenado=true;
        for(int i=0; i<len-1; i++){
        // invocamos a la funcion para determinar si corresponde o no permutar
            if( criterio(arr[i],arr[i+1])>0 ){
              T aux = arr[i];
              arr[i] = arr[i+1];
              arr[i+1] = aux;
              ordenado = false;
            } // cierra bloque if
        } // cierra bloque for
      }// cierra bloque while
return; }
```

Ordenar arrays de diferentes tipos de datos con diferentes criterios de ordenamiento

A continuación analizaremos algunas funciones que comparan pares de valores (ambos del mismo tipo) y determinan cual de esos valores debe preceder al otro.

Comparar cadenas, criterio alfabético ascendente:

```
int criterioAZ(string e1, string e2)
{
      return e1>e2?1:e1<e2?-1:0;
```

```
}
//Comparar cadenas, criterio alfabético descendente:
int criterioZA(string e1, string e2)
{
	return e2>e1?1:e2<e1?-1:0;
}
//Comparar enteros, criterio numérico ascendente:
	int criterio09(int e1, int e2)
{
	return e1-e2;
}
//Comparar enteros, criterio numérico descendente:
	int criterio90(int e1, int e2)
{
	return e2-e1;
}
```

Probamos lo anterior:

```
int main()
{
	int len = 4;
	// un array con 6 cadenas
	string x[] = {"Pablo", "Pedro", "Andres", "Juan"};
	// ordeno ascendentemente pasando como parametro la funcion criterioAZ
	ordenar<string>(x,len,criterioAZ);
	mostrar<string>(x,len);
	// ordeno descendentemente pasando como parametro la funcion criterioZA
	ordenar<string>(x,len,criterioZA);
	mostrar<string>(x,len);
	// un array con 6 enteros
	int y[] = {4, 1, 7, 2};
	// ordeno ascendentemente pasando como parametro la funcion criterio09
	ordenar<int>(y,len,criterio09);
	mostrar<int>(y,len);
	// ordeno ascendentemente pasando como parametro la funcion criterio90
	ordenar<int>(y,len,criterio90);
	mostrar<int>(y,len);
  return 0;
}
```

Arrays de estructuras

Trabajaremos con la siguiente estructura:

```
struct Alumno
{
	int legajo;
	string nombre;
	int nota;
};
// esta funcion nos permitira "crear alumnos" facilmente
Alumno crearAlumno(int le, string nom, int nota)
{
	Alumno a;
	a.legajo = le;
	a.nombre = nom;
	a.nota = nota;
  return a; }
```

Mostrar arrays de estructuras

La función `mostrar` que analizamos más arriba no puede operar con arrays de estructuras porque el objeto `cout` no sabe cómo mostrar elementos cuyos tipos de datos fueron definidos por el programador. Entonces recibiremos cómo parámetro una función que será la encargada de mostrar dichos elementos por consola.

```
template <typename T>
void mostrar(T arr[], int len, void (*mostrarFila)(T))
{
      for(int i=0; i<len; i++){
        mostrarFila(arr[i]);
      }
return;
}
```

Probemos la función anterior:

```
// desarrollamos una funcion que muestre por consola los valores de una estructura
void mostrarAlumno(Alumno a)
{
      cout << a.legajo << ", " << a.nombre << ", "<< a.nota << endl;
}

int main()
{
      Alumno arr[6];
      arr[0] = crearAlumno(30,"Juan",5);
      arr[1] = crearAlumno(10,"Pedro",8);
      arr[2] = crearAlumno(20,"Carlos",7);
      arr[3] = crearAlumno(60,"Pedro",10);
      arr[4] = crearAlumno(40,"Alberto",2);
      arr[5] = crearAlumno(50,"Carlos",4);
      int len = 6;
      // invoco a la funcion que muestra el array
      mostrar<Alumno>(arr,len,mostrarAlumno);
  return 0;
}
```

Ordenar arrays de estructuras por diferentes criterios

Recordemos la función `ordenar`:

```
template <typename T>void ordenar(T arr[], int len, int (*criterio)(T,T))
{
      bool ordenado=false;
      while(!ordenado){
        ordenado=true;
        for(int i=0; i<len-1; i++){
            if( criterio(arr[i],arr[i+1])>0 ){
              T aux = arr[i];
              arr[i] = arr[i+1];
              arr[i+1] = aux;
              ordenado = false;
            }
        }
      }
 return;
}
```

Definimos diferentes criterios de precedencia de alumnos:

```
//a1 precede a a2 si a1.legajo<a2.legajo:
int criterioAlumnoLegajo(Alumno a1, Alumno a2)
{
	return a1.legajo-a2.legajo;
}
//a1 precede a a2 si a1.nombre<a2.nombre:
int criterioAlumnoNombre(Alumno a1, Alumno a2)
{
	return a1.nombre<a2.nombre?-1:a1.nombre>a2.nombre?1:0;
}
```

a1 precede a a2 si a1.nombre<a2.nombre. A igualdad de nombres entonces precederá el alumno que tenga menor número de legajo:

```
int criterioAlumnoNomYLeg(Alumno a1, Alumno a2)
{
	if( a1.nombre == a2.nombre ){
	  return a1.legajo-a2.legajo;
	}
	else{
	  return a1.nombre<a2.nombre?-1:a1.nombre>a2.nombre?1:0;
	}
}
```

Ahora sí, probemos los criterios anteriores con la función ordenar.

```
int main()
{
	Alumno arr[6];
	arr[0] = crearAlumno(30,"Juan",5);
	arr[1] = crearAlumno(10,"Pedro",8);
	arr[2] = crearAlumno(20,"Carlos",7);
	arr[3] = crearAlumno(60,"Pedro",10);
	arr[4] = crearAlumno(40,"Alberto",2);
	arr[5] = crearAlumno(50,"Carlos",4);
	int len = 6;
	// ordeno por legajo
	ordenar<Alumno>(arr,len,criterioAlumnoLegajo);
	mostrar<Alumno>(arr,len,mostrarAlumno);
	// ordeno por nombre
	ordenar<Alumno>(arr,len,criterioAlumnoNombre);
	mostrar<Alumno>(arr,len,mostrarAlumno);
	// ordeno por nombre+legajo
	ordenar<Alumno>(arr,len,criterioAlumnoNomYLeg);
	mostrar<Alumno>(arr,len,mostrarAlumno);
  return 0;
}
```

..

Resumen de plantillas

Función agregar.

Descripción: Agrega el valor v al final del array arr e incrementa su longitud.

```
template <typename T> void agregar(T arr[], int& len, T v)
{
	arr[len]=v;
	len++;
	return;
}
```

Función `buscar`.
Descripción: Busca la primer ocurrencia de `v` en `arr`; retorna su posición o un valor negativo si `arr` no contiene a `v`.

```
template <typename T, typename K>
int buscar(T arr[], int len, K v, int (*criterio)(T,K))
{
      Int i=0;
      while( i<len && criterio(arr[i],v)!=0 ){
        i++;
      }
      return i<len?i:-1;
}
```

Función `eliminar`:.
Descripción: Elimina el valor ubicado en la posición `pos` del array `arr`, decrementando su longitud.

```
template <typename T>
void eliminar(T arr[], int& len, int pos)
{
      int i=0;
      for(int i=pos; i<len-1; i++ )
      {
        arr[i]=arr[i+1];
      }
      len--;
return;}
```

Función `insertar`.
Descripción: Inserta el valor `v` en la posición `pos` del array `arr`, incrementando su longitud.

```
template <typename T>
void insertar(T arr[], int& len, T v, int pos)
{
      for(int i=len-1; i>=pos; i--)
      {
        arr[i+1]=arr[i];
      }
      arr[pos]=v;
      len++;
  return;
}
```

Función `insertarOrdenado`.
Descripción: Inserta el valor `v` en el array `arr` en la posición que corresponda según el criterio `criterio`.

```
template <typename T>
int insertarOrdenado(T arr[], int& len, T v, int (*criterio)(T,T))
{
      int i=0;
      while( i<len && criterio(arr[i],v)<=0 ){
        i++;
      }
      insertar<T>(arr,len,v,i);
  return i; }
```

Función `buscaEInserta`.
Descripción: Busca el valor `v` en el array `arr`; si lo encuentra entonces retorna su posición y asigna `true` al parámetro `enc`. De lo contrario lo inserta donde corresponda según el criterio `criterio`, asigna `false` al parámetro `enc` y retorna la posición en donde finalmente quedó ubicado el nuevo valor.

```
template <typename T>
int buscaEInserta(T arr[], int& len, T v, bool& enc, int
(*criterio)(T,T))
{
      // busco el valor
      int pos = buscar<T,T>(arr,len,v,criterio);
      // determino si lo encontre o no
      enc = pos>=0;
      // si no lo encontre entonces lo inserto ordenado
      if( !enc ){
        pos = insertarOrdenado<T>(arr,len,v,criterio);
      }
  return pos;
}
```

Función ordenar
Descripción: Ordena el array `arr` según el criterio de precedencia que indica la función `criterio`.

```
template <typename T>
void ordenar(T arr[], int len, int (*criterio)(T,T))
{
      boolordenado=false;
      while(!ordenado){
        ordenado=true;
        for(int i=0; i<len-1; i++){
            if( criterio(arr[i],arr[i+1])>0 ){
              T aux = arr[i];
              arr[i] = arr[i+1];
              arr[i+1] = aux;
              ordenado = false;
            }
        }
      }
return;
}
```

Función `busquedaBinaria`.
Descripción: Busca el elemento `v` en el array `arr` que debe estar ordenado según el criterio `criterio`. Retorna la posición en donde se encuentra el elemento o donde este debería ser insertado.

```
template<typename T, typename K>
int busquedaBinaria(T a[], int len, K v, int (*criterio)(T, K), bool&
enc)
{
      int i=0;
      int j=len-1;
      int k=(i+j)/2;
      enc=false;
      while( !enc && i<=j ){
        if( criterio(a[k],v)>0 ){
```

```
            j=k-1;
        }
        else {
            if( criterio(a[k],v)<0 ){
              i=k+1;
            }
            else {
             enc=true;
            }
        }
       k=(i+j)/2;
      } return criterio(a[k],v)>=0?k:k+1;
}
```

Operaciones sobre archivos

Antes de comenzar

Este documento resume las principales operaciones que son generalmente utilizadas para la manipulación de archivos. Cubre el uso de todas las funciones primitivas que provee el lenguaje C y explica también como desarrollar templates que faciliten el uso de dichas funciones.

Introducción

Las funciones que analizaremos en este documento son[1]:

`fopen` - Abre un archivo.
`fwrite` - Graba datos en el archivo.
`fread` - Lee datos desde el archivo.
`feof` - Indica si quedan o no más datos para ser leidos desde el archivo.
`fseek` - Permite reubicar el indicador de posición del archivo.
`ftell` - Indica el número de byte al que está apuntando el indicador de posición del archivo.
`fclose` - Cierra el archivo.

Todas estas funciones están declaradas en el archivo `stdio.h` por lo que para utilizarlas debemos agregar la siguiente línea a nuestros programas:

```
#include <stdio.h>
```

Además, basándonos en las anteriores desarrollaremos las siguientes funciones que nos permitirán operar con archivos de registros:

`seek` – Mueve el indicador de posición de un archivo al inicio de un determinado registro.
`fileSize` – Indica cuantos registros tiene un archivo.
`filePos` – Retorna el número de registro que está siendo apuntado por el indicador de posición.
`read` – Lee un registro del archivo.
`write` – Graba un registro en el archivo.

Comenzando a operar con archivos

Grabar un archivo de caracteres

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
      // abro el archivo; si no existe => lo creo vacio
      FILE* arch = fopen("DEMO.DAT","wb+");
      char c = 'A';
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'A' contenido en c
      c = 'B'; // C provee también fputc(c, arch);
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'B' contenido en c
      c = 'C';
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'C' contenido en c
      fclose(arch);
  return 0;}
```

Leer un archivo caracter a caracter

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
// abro el archivo para lectura
```

[1] El detalle de las funciones de C se muestran en el anexo 1

```
      FILE* arch = fopen("DEMO.DAT","rb+");
      char c;
      // leo el primer caracter grabado en el archivo
      fread(&c,sizeof(char),1,arch);
      // mientras no llegue el fin del archivo...
      while( !feof(arch) ){
        // muestro el caracter que lei
        cout << c << endl;
        // leo el siguiente caracter
        fread(&c,sizeof(char),1,arch);
      }
      fclose(arch);
  return 0;
}
```

Archivos de registros

Las mismas funciones que analizamos en el apartado anterior nos permitirán operar con archivos de estructuras o archivos de registros. La única consideración que debemos tener en cuenta es que la estructura que vamos a grabar en el archivo no debe tener campos de tipos `string`. En su lugar debemos utilizar arrays de caracteres, al más puro estilo C.
Veamos un ejemplo:

```
struct Persona
{
      int dni;
      char nombre[25];
      double altura;
};
```

Grabar un archivo de registros

El siguiente programa lee por consola los datos de diferentes personas y los graba en un archivo de estructuras `Persona`.

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      FILE* f = fopen("PERSONAS.DAT","w+b");
      int dni;
      string nom;
      double altura;
      // el usuario ingresa los datos de una persona
      cout << "Ingrese dni, nombre, altura: ";
      cin >> dni;
      cin >> nom;
      cin >> altura;
      while( dni>0 ){
        // armo una estructura para grabar en el archivo
        Persona p;
        p.dni = dni;
        strcpy(p.nombre,nom.c_str()); // la cadena hay que copiarla con strcpy
        p.altura = altura;
        fwrite(&p,sizeof(Persona),1,f); // grabo la estructura en el archivo
        cout << "Ingrese dni, nombre, altura: ";
        cin >> dni;
        cin >> nom;
        cin >> altura;
```

```
      }
      fclose(f);
  return 0;
}
```

En este programa utilizamos el método `c_str` que proveen los objetos `string` de C++ para obtener una cadena de tipo `char*` (de C) tal que podamos pasarla cómo parámetro a la función `strcpy` y así copiar el nombre que ingresó el usuario en la variable `nom` al campo `nombre` de la estructura `p`.

Leer un archivo de registros

A continuación veremos un programa que muestra por consola todos los registros del archivo `PERSONAS.DAT`.

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      FILE* f = fopen("PERSONAS.DAT","rb+");
      Persona p;
      // leo el primer registro
      fread(&p,sizeof(Persona),1,f);
      while( !feof(f) ){
        // muestro cada campo de la estructura
        cout << p.dni << ", " << p.nombre <<", " << p.altura << endl;
        // leo el siguiente registro
        fread(&p,sizeof(Persona),1,f);
      }
      fclose(f);
 return 0; }
```

Acceso directo a los registros de un archivo

La función `fseek` que provee C/C++ permite mover el indicador de posición del archivo hacia un determinado byte. El problema surge cuando queremos que dicho indicador se desplace hacia el primer byte del registro ubicado en una determinada posición. En este caso la responsabilidad de calcular el número de byte que corresponde a dicha posición será nuestra. Lo podemos calcular de la siguiente manera:

```
void seek(FILE* arch, int recSize, int n)
{
      // SEEK_SET indica que la posicion n es absoluta respecto del inicio del archivo
      fseek(arch, n*recSize,SEEK_SET);
}
```

El parámetro `recSize` será el `sizeof` del tipo de dato de los registros que contiene el archivo. En el siguiente ejemplo accedemos directamente al tercer registro del archivo `PERSONAS.DAT` y mostramos su contenido.

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      FILE* f = fopen("PERSONAS.DAT","r+b");
      Persona p;
      // muevo el indicador de posicion hacia el 3er registro (contando desde 0)
      seek(f,sizeof(Persona), 2);
      // leo el registro apuntado por el indicador de posicion
      fread(&p,sizeof(Persona),1,f);
      // muestro cada campo de la estructura leida
```

```
      cout << p.dni << ", " << p.nombre <<", " << p.altura << endl;
  return 0;
}
```

Cantidad de registros de un archivo

En C/C++ no existe una función que nos permita conocer cuántos registros tiene un archivo. Sin embargo podemos programarla nosotros mismos utilizando las funciones `fseek` y `ftell`.

```
long fileSize(FILE* f, int recSize)
{
      // tomo la posicion actual
      long curr=ftell(f);
      // muevo el puntero al final del archivo
      fseek(f,0,SEEK_END); // SEEK_END hace referencia al final del archivo
      // tomo la posicion actual (ubicado al final)
      long ultimo=ftell(f);
      // vuelvo a donde estaba al principio
      fseek(f,curr,SEEK_SET);
      return ultimo/recSize;
}
```

Probamos ahora las funciones `seek` y `fileSize`.

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      FILE* f = fopen("PERSONAS.DAT","r+b");
      Persona p;
      // cantidad de registros del archivo
      long cant = fileSize(f,sizeof(Persona));
      for(int i=cant-1; i>=0; i--){
        // acceso directo al i-esimo registro del archivo
        seek(f,sizeof(Persona),i);
        fread(&p,sizeof(Persona),1,f);
        // muestro el registro leido
        cout << p.dni <<", "<<p.nombre<<", "<< p.altura << endl;
      }
      fclose(f);
```

return 0;}

Identificar el registro que está siendo apuntando por el identificador de posición del archivo

Dado un archivo y el tamaño de sus registros podemos escribir una función que indique cual será el próximo registro será afectado luego de realizar la próxima lectura o escritura. A esta función la llamaremos: `filePos`.

```
long filePos(FILE* arch, int recSize)
{
      return ftell(arch)/recSize;
}
```

Templates

Como podemos ver, las funciones `fread` y `fwrite`, y las funciones `seek` y `fileSize` que desarrollamos más arriba realizan su tarea en función del `sizeof` del tipo de dato del valor que vamos a leer o a escribir en el archivo. Por esto, podemos parametrizar dicho tipo de dato mediante un template lo que nos permitirá simplificar dramáticamente el uso de todas estas funciones.

Template: read

```
template <typename T> T read(FILE* f)
```

```
{
	T buff;
	fread(&buff,sizeof(T),1,f);
	return buff;
}
```

Template: write

```
template <typename T> void write(FILE* f, T v)
{
	fwrite(&v,sizeof(T),1,f);
	return;
}
```

Template: seek

```
template <typename T> void seek(FILE* arch, int n)
{
	// SEEK_SET indica que la posicion n es absoluta respecto del inicio del archivo
	fseek(arch, n*sizeof(T),SEEK_SET);
}
```

Template: fileSize

```
template <typename T> long fileSize(FILE* f)
{
	// tomo la posicion actual
	long curr=ftell(f);
	// muevo el puntero al final del archivo
	fseek(f,0,SEEK_END); // SEEK_END hace referencia al final del archivo
	// tomo la posicion actual (ubicado al final)
	long ultimo=ftell(f);
	// vuelvo a donde estaba al principio
	fseek(f,curr,SEEK_SET);
	return ultimo/sizeof(T);}
```

Template: filePos

```
template <typename T> long filePos(FILE* arch)
{
	return ftell(arch)/sizeof(T);
}
```

Template: busquedaBinaria

El algoritmo de la búsqueda binaria puede aplicarse perfectamente para emprender búsquedas sobre los registros de un archivo siempre y cuando estos se encuentren ordenados.
Recordemos que en cada iteración este algoritmo permite descartar el 50% de los datos; por esto, en el peor de los casos, buscar un valor dentro de un archivo puede insumir $\log_2(n)$ accesos a disco siendo n la cantidad de registros del archivo.

```
template <typename T, typename K>
int busquedaBinaria(FILE* f, K v, int (*criterio)(T,K))
{
	// indice que apunta al primer registro
	int i = 0;
	// indice que apunta al ultimo registro
	int j = fileSize<T>(f)-1;
	// calculo el indice promedio y posiciono el indicador de posicion
	int k = (i+j)/2;
	seek<T>(f,k);
	// leo el registro que se ubica en el medio, entre i y j
	T r = leerArchivo<T>(f);
	while( i<=j && criterio(r,v)!=0 ){
	  // si lo que encontre es mayor que lo que busco...
	  if( criterio(r,v)>0 ){
```

```
            j = k-1;
        }
        Else {
            // si lo que encontre es menor que lo que busco...
            if( criterio(r,v)<0 ){
              i=k+1;
            }
        }
        // vuelvo a calcular el indice promedio entre i y j
        k = (i+j)/2;
        // posiciono y leo el registro indicado por k
        seek<T>(f,k);
        // leo el registro que se ubica en la posicion k
        r = leerArchivo<T>(f);
      }
      // si no se cruzaron los indices => encontre lo que busco en la posicion k
return i<=j?k:-1;
}
```

Ejemplos

Leer un archivo de registros usando el template `read`.

```
      f = fopen("PERSONAS.DAT","rb+");
      // leo el primer registro
      Persona p = read<Persona>(f);
      while( !feof(f) ){
        // muestro
        cout << p.dni<<", "<<p.nombre<<", "<<p.altura << endl;
        p = read<Persona>(f);
      }
      fclose(f);
```

Escribir registros en un archivo usando el template `write`.

```
      f = fopen("PERSONAS.DAT","w+b");
      // armo el registro
      Persona p;
      p.dni = 10;
      strcpy(p.nombre,"Juan");
      p.altura = 1.70;
      // escribo el registro
      write<Persona>(f,p);
      fclose(f);
```

Acceso directo a los registros de un archivo usando los templates `fileSize`, `seek` y `read`.

```
      f = fopen("PERSONAS.DAT","r+b");
      // cantidad de registros del archivo
      long cant = fileSize<Persona>(f);
      for(int i=cant-1; i>=0; i--){
        // acceso directo al i-esimo registro del archivo
        seek<Persona>(f,i);
        Persona p = read<Persona>(f);
        cout << p.dni<<", "<<r.nombre<<", "<< r.altura << endl;
      }
      fclose(f);
```

Operaciones sobre estructuras dinámicas

Antes de comenzar

Este documento resume las principales operaciones que generalmente son utilizadas para la manipulación de las estructuras dinámicas: lista, pila y cola. Cubre además los conceptos de "puntero" y "dirección de memoria" y explica también cómo desarrollar templates para facilitar y generalizar el uso de dichas operaciones.

Punteros y direcciones de memoria

Llamamos "puntero" es una variable cuyo tipo de datos le provee la capacidad de contener una dirección de memoria. Veamos:

```
int a = 10; // declaro la variable a y le asigno el valor 10
int* p = &a; // declaro la variable p y le asigno "la direccion de a"
cout << *p << endl; // muestro "el contenido de p"
```

En este fragmento de código asignamos a la variable `p`, cuyo tipo de datos es "puntero a entero", la dirección de memoria de la variable `a`. Luego mostramos por pantalla el valor contenido en el espacio de memoria que está siendo referenciado por `p`; en otras palabras: mostramos "el contenido de `p`".
El operador `&` aplicado a una variable retorna su dirección de memoria. El operador `*` aplicado a un puntero retorna "su contenido".

Asignar y liberar memoria dinámicamente

A través de los comandos `new` y `delete` respectivamente podemos solicitar memoria en cualquier momento de la ejecución del programa y luego liberarla cuando ya no la necesitemos.
En la siguiente línea de código solicitamos memoria dinámicamente para contener un valor entero. La dirección de memoria del espacio obtenido la asignamos a la variable `p`, cuyo tipo es: `int*` (puntero a entero).

```
int* p = new int();
```

Luego, asignamos el valor 10 en el espacio de memoria direccionado por `p`.

```
*p = 10;
```

Ahora mostramos por pantalla el contenido asignado en el espacio de memoria al que `p` hace referencia:

```
cout << *p << endl;
```

Finalmente liberamos la memoria que solicitamos al comienzo.

```
delete p;
```

Nodo

Un nodo es una estructura autoreferenciada que, además de contener los datos propiamente dichos, posee al menos un campo con la dirección de otro nodo del mismo tipo.
Para facilitar la comprensión de los algoritmos que estudiaremos en este documento trabajaremos con nodos que contienen un único valor de tipo `int`; pero al finalizar reescribiremos todas las operaciones como templates de forma tal que puedan ser aplicadas a nodos de cualquier tipo.

```
struct Nodo{
      int info; // valor que contiene el nodo
      Nodo* sig; // puntero al siguiente nodo
      };
```

Punteros a estructuras: operador "flecha"

Para manipular punteros a estructuras podemos utilizar el operador `->` (llamado operador "flecha") que sim-plifica notablemente el acceso a los campos de la estructura referenciada.
Veamos. Dado el puntero `p`, declarado e inicializado como se observa en la siguiente línea de código:

```
Nodo* p = new Nodo();
```

Podemos asignar un valor a su campo `info` de la siguiente manera:

```
(*p).info = 10;
```

La línea anterior debe leerse así: "asigno el valor 10 al campo `info` del nodo referenciado por `p`". Sin embargo, el operador "flecha" facilita la notación anterior, así:

```
p->info = 10;
```

Luego, las líneas: `(*p).info = 10` y `p->info = 10` son equivalentes,

Listas enlazadas

Dados los nodos: n_1, n_2,..., n_{i-1}, n_i, n_{i+1},...n_{n-1}, n tales que: para todo i>=1 e i<n se verifica que: n_i contiene la dirección de: n_{i+1} y n contiene la dirección nula `NULL`, entonces: el conjunto de nodos n_i constituye una lista en-lazada. Si `p` es un puntero que contiene la dirección de n_1 (primer nodo) entonces diremos que: "`p` es la lista".
Las operaciones que veremos a continuación nos permitirán manipular los nodos de una lista enlazada.

Agregar un nodo al final de una lista enlazada

La función `agregar` agrega un nuevo nodo con el valor `x` al final de la lista referenciada por `p`.

```
void agregarNodo(Nodo*& p, int x)
{
      Nodo* nuevo = new Nodo();
      nuevo->info = x;
      nuevo->sig = NULL;
      if( p==NULL ){
        p = nuevo;
      }
      Else {
       Nodo* aux = p;
        while(aux->sig!=NULL ){
            aux = aux->sig;
        }
        aux->sig = nuevo;
      }
}
```

Mostrar el contenido de una lista enlazada

La función `mostrar` recorre la lista `p` y muestra por pantalla el valor que contienen cada uno de sus nodos.

```
void mostrar(Nodo* p)
{
      Nodo* aux = p;
      while( aux!=NULL ) {
        cout << aux->info << endl;
        aux = aux->sig;
      }
}
```

Liberar la memoria que ocupan los nodos de una lista enlazada

La función `liberar` recorre la lista `p` liberando la memoria que ocupan cada uno de sus nodos.

```
void liberar(Nodo*& p)
{
      Nodo* aux;
      while( p!=NULL ){
        aux = p;
        p = p->sig;
        delete aux;
      }
}
```

Probar las funciones anteriores

El siguiente programa agrega valores en una lista enlazada; luego muestra su contenido y finalmente libera la memoria utilizada.

```
int main()
{
	// declaro la lista (o, mejor dicho: el puntero al primer nodo
	Nodo* p = NULL;
	// agrego valores
	agregarNodo(p,1);
	agregarNodo(p,2);
	agregarNodo(p,3);
	agregarNodo(p,4);
	mostrar(p);
	// libero la memoria utilizada
	liberar(p);
  return 0;
}
```

Determinar si una lista enlazada contiene o no un valor especificado

La función `buscar` permite determinar si alguno de los nodos de la lista `p` contiene el valor `v`. Retorna un puntero al nodo que contiene dicho valor o `NULL` si ninguno de los nodos lo contiene.

```
Nodo* buscar(Nodo* p, int v)
{
	Nodo* aux = p;
	while( aux!=NULL && aux->info!=v ){
	  aux=aux->sig;
	}
	return aux;
}
```

Eliminar de la lista al nodo que contiene un determinado valor

La función `eliminar` permite eliminar de la lista `p` al nodo que contiene el valor `v`.

```
void eliminar(Nodo*& p, int v)
{
	Nodo* aux = p;
	Nodo* ant = NULL;
	while( aux!=NULL && aux->info!=v ){
	  ant = aux;
	  aux = aux->sig;
	}
	if( ant!=NULL ){
	  ant->sig = aux->sig;
	}
	Else {
	  p = aux->sig;
	}
	delete aux;
}
```

Eliminar el primer nodo de una lista

La función `eliminarPrimerNodo` elimina el primer nodo de la lista y retorna el valor que este contenía.

```
int eliminarPrimerNodo(Nodo*& p)
{
	int ret = p->info;
	Nodo* aux = p;
```

```
      p = p->sig;
      delete aux;

return ret;
}
```

Insertar un nodo manteniendo el orden de la lista

La función `insertarOrdenado` permite insertar el valor `v` respetando el ctriterio de ordenamiento de la lista `p`; se presume que la lista está ordenada o vacía. Retorna la dirección de memoria del nodo insertado.

```
Nodo* insertarOrdenado(Nodo*& p, int v)
{
      Nodo* nuevo = new Nodo();
      nuevo->info = v;
      nuevo->sig = NULL;
      Nodo* ant = NULL;
      Nodo* aux = p;
      while( aux!=NULL && aux->info<=v ){
        ant = aux;
        aux = aux->sig;
      }
      if( ant==NULL ){
        p = nuevo;
      }
      Else {
        ant->sig = nuevo;
      }
      nuevo->sig = aux;
return nuevo;}
```

Ordenar una lista enlazada

La función `ordenar` ordena la lista direccionada por `p`. La estrategia consiste en eliminar uno a uno los nodos de la lista e insertarlos en orden en una lista nueva; finalmente hacer que `p` apunte a la nueva lista.

```
void ordenar(Nodo*& p)
{
      Nodo* q = NULL;
      while( p!=NULL ){
        int v = eliminarPrimerNodo(p);
        insertarOrdenado(q,v);
      }
      p = q;
}
```

Insertar en una lista enlazada un valor sin repetición

La función `buscaEInsertaOrdenado` busca el valor `v` en la lista `p`. Si no lo encuentra entonces lo inserta respetando el criterio de ordenamiento. Retorna un puntero al nodo encontrado o insertado y asigna el valor `true` o `false` al parámetro `enc` según corresponda.

```
Nodo* buscaEInsertaOrdenado(Nodo*& p, int v, bool& enc)
{
      Nodo* x = buscar(p,v);
      enc = x!=NULL;
      if( !enc ){
        x = insertarOrdenado(p,v);
      }
  return x;
}
```

Templates

Reprogramaremos todas las funciones que hemos analizado de forma tal que puedan ser utilizadas con listas enlazadas de nodos de cualquier tipo de datos.

El nodo

```
template <typename T> struct Nodo {
      T info; // valor que contiene el nodo
      Nodo<T>* sig; // puntero al siguiente nodo
      };
```

Función: agregarNodo

```
template <typename T> void agregar(Nodo<T>*& p, T x)
{
      // creo un nodo nuevo
      Nodo<T>* nuevo = new Nodo<T>();
      nuevo->info =
      x; nuevo->sig = NULL;
      if( p==NULL ){
        p = nuevo;
      }
      Else {
        Nodo<T>* aux = p;
        while( aux->sig!=NULL ){
            aux = aux->sig;
        }
        aux->sig = nuevo;
      }
}
```

Función: liberar

```
template <typename T> void liberar(Nodo<T>*& p)
{
      Nodo<T>* aux;
      while( p!=NULL ){
        aux = p;
        p = p->sig;
        delete p;
      }
}
```

Ejemplo de uso

En el siguiente código creamos dos listas; la primera con valores enteros y la segunda con cadenas de caracteres.

```
int main()
{
      // creo la lista de enteros
      Nodo<int>*p1 = NULL;
      agregar<int>(p1,1);
      agregar<int>(p1,2);
      agregar<int>(p1,3);
      // la recorro mostrando su contenido
      Nodo<int>* aux1 = p1;
      while( aux1!=NULL ){
```

```
        cout << aux1->info << endl;
        aux1 = aux1->sig;
      }
      // libero la memoria
      liberar<int>(p1);
      // creo la lista de cadenas
      Nodo<string>* p2 = NULL;
      agregar<string>(p2,"uno");
      agregar<string>(p2,"dos");
      agregar<string>(p2,"tres");
      // la recorro mostrando su contenido
      Nodo<string>* aux2 = p2;
      while( aux2!=NULL ){
        cout << aux2->info << endl;
        aux2 = aux2->sig;
      }
      // libero la memoria
      liberar<string>(p2);
  return 0;
}
```

Ejemplo con listas de estructuras

```
struct Alumno {
      int leg;
      string nom;
      };

Alumno crearAlumno(int leg, string nom)
{
      Alumno a;
      a.leg = leg;
      a.nom = nom;
      return a;
}
int main()
{
      Nodo<Alumno>* p = NULL;
      agregar<Alumno>(p,crearAlumno(10,"Juan"));
      agregar<Alumno>(p,crearAlumno(20,"Pedro"));
      agregar<Alumno>(p,crearAlumno(30,"Pablo"));
      Nodo<Alumno>* aux = p;
      while( aux!=NULL ){
        cout << aux->info.leg << ", " << aux->info.nom << endl;
        aux = aux->sig;
      }
      liberar<Alumno>(p);
return 0;}
```

Función: buscar

```
template <typename T, typename K>
Nodo<T>* buscar(Nodo<T>* p, K v, int (*criterio)(T,K) )
{
      Nodo<T>* aux = p;
      while( aux!=NULL && criterio(aux->info,v)!=0 ){
        aux = aux->sig;
      }
 return aux;
}
```

Veamos un ejemplo de como utilizar esta función. Primero las funciones que permiten comparar alumnos

```
int criterioAlumnoLeg(Alumno a,int leg)
{
      return a.leg-leg;
}


int criterioAlumnoNom(Alumno a,string nom)
{
      return strcmp(a.nom.c_str(),nom);
}
```

Ahora analicemos el código de un programa que luego de crear una lista de alumnos le permite al usuario buscar alumnos por legajo y por nombre.

```
int main()
{
      Nodo<Alumno>* p = NULL;
      agregar<Alumno>(p,crearAlumno(10,"Juan"));
      agregar<Alumno>(p,crearAlumno(20,"Pedro"));
      agregar<Alumno>(p,crearAlumno(30,"Pablo"));
      int leg;
      cout << "Ingrese el legajo de un alumno: ";
      cin >> leg;
      // busco por legajo
      Nodo<Alumno>* r = buscar<Alumno,int>(p,leg,criterioAlumnoLeg);
      if( r!=NULL ){
        cout << r->info.leg << ", " << r->info.nom << endl;
      }
      string nom;
      cout << "Ingrese el nombre de un alumno: ";
      cin >> nom;
      // busco por nombre
      r = buscar<Alumno,string>(p,nom,criterioAlumnoNom);
      if( r!=NULL ){
        cout << r->info.leg << ", " << r->info.nom << endl;
      }
      liberar<Alumno>(p);
  return 0;}
```

Función: eliminar

```
template <typename T, typename K>
void eliminar(Nodo<T>*& p, K v, int (*criterio)(T,K))
{
      Nodo<T>* aux = p;
      Nodo<T>* ant = NULL;
      while( aux!=NULL && criterio(aux->info,v)!=0 ){
        ant = aux;
        aux = aux->sig;
      }
      if( ant!=NULL ){
        ant->sig = aux->sig;
      }
      else {
        p = aux->sig;
      }
      delete aux;}
```

Función: eliminarPrimerNodo

```
template <typename T>
T eliminarPrimerNodo(Nodo<T>*& p)
{
      T ret = p->info;
      Nodo<T>* aux = p->sig;
      delete p;
      p = aux;
  return ret;
}
```

Función: insertarOrdenado

```
template <typename T>
Nodo<T>* insertarOrdenado(Nodo<T>*& p, T v, int (*criterio)(T,T) )
{
      Nodo<T>* nuevo = new Nodo<T>();
      nuevo->info = v;
      nuevo->sig = NULL;
      Nodo<T>* aux = p;
      Nodo<T>* ant = NULL;
      while( aux!=NULL && criterio(aux->info,v)<=0 ){
        ant = aux;
        aux = aux->sig;
      }
      if( ant==NULL ){
        p = nuevo;
      } else {
          ant->sig = nuevo;
        }
      nuevo->sig = aux;
  return nuevo;
}
```

Función: ordenar

```
template <typename T>
void ordenar(Nodo<T>*& p, int (*criterio)(T,T))
{
      Nodo<T>* q = NULL;
      while( p!=NULL ) {
        T v = eliminarPrimerNodo<T>(p);
        insertarOrdenado<T>(q,v,criterio);
      }
      p = q;
}
```

Función: buscaEInsertaOrdenado

```
template <typename T>
Nodo<T>* buscaEInsertaOrdenado(Nodo<T>*&p,T v,bool&enc,int
(*criterio)(T,T))
{
      Nodo<T>* x = buscar<T,T>(p,v,criterio);
      enc = x!=NULL;
      if( !enc ) {
        x = insertarOrdenado<T>(p,v,criterio);
      }
```

```
  return x;
}
```

Pilas

Una pila es una estructura restrictiva que admite dos únicas operaciones: poner y sacar o, en inglés: push y pop. La característica principal de la pila es que el primer elemento que ingresa a la estructura será el último elemento en salir; por esta razón se la denomina LIFO: Last In First Out.

Función: push (poner)

```
template <typename T> void push(Nodo<T>*& p, T v)
{
      Nodo<T>* nuevo = new Nodo<T>();
      nuevo->info = v;
      nuevo->sig = p;
      p = nuevo;
}
```

Función: pop (sacar)

```
template <typename T> T pop(Nodo<T>*& p)
{
      T ret = p->info;
      Nodo<T>* aux = p;
      p = aux ->sig;
      delete aux;
  return ret;
}
```

Ejemplo de cómo usar una pila

```
int main()
{
      Nodo<int>* p = NULL;
      push<int>(p,1);
      push<int>(p,2);
      push<int>(p,3);
      while( p!=NULL ) {
        cout << pop<int>(p) << endl;
      }
  return 0;
}
```

Colas

Una cola es una estructura restrictiva que admite dos únicas operaciones: encolar y desencolar. La característica principal de la cola es que el primer elemento que ingresa a la estructura será también el primero en salir; por esta razón se la denomina FIFO: First In First Out.

Cola sobre una lista enlazada con dos punteros

Podemos implementar una estructura cola sobre una lista enlazada con dos punteros `p` y `q` que hagan referencia al primer y al último nodo respectivamente. Luego, para encolar valor simplemente debemos agregarlo a un nodo y referenciarlo como "el siguiente" de `q`. Y para desencolar siempre debemos tomar el valor que está siendo referenciado por `p`.

Función: agregar

```
template <typename T>
void agregar(Nodo<T>*& p, Nodo<T>*& q, T v)
```

```
{
      Nodo<T>* nuevo = new Nodo<T>();
      nuevo->info = v;
      nuevo->sig = NULL;
      if( p==NULL ){
        p = nuevo;
      }
      Else {
        q->sig = nuevo;
      }
      q = nuevo;
}
```

Función: suprimir

```
template <typename T>
T desencolar(Nodo<T>*& p, Nodo<T>*& q)
{
      T ret = p->info;
      Nodo<T>* aux = p;
      p = p->sig;
      if( p == NULL) q = NULL;
      delete aux;
  return ret; }
```

Veamos un ejemplo de como utilizar una cola implementada sobre una lista con dos punteros.

```
int main()
{
      Nodo<int>* p = NULL;
      Nodo<int>* q = NULL;
      agregar<int>(p,q,1);
      agrergar<int>(p,q,2);
      agragar<int>(p,q,3);
      cout << suprimir<int>(p,q) << endl;
      cout << suprimir<int>(p,q) << endl;
      agregar<int>(p,q,4);
      agregar<int>(p,q,5);
      while( p!=NULL ){
        cout << desencolar<int>(p,q) << endl;
      }
  return 0;
}
```

Biblioteca de templates

Operaciones sobre arrays

Función: agregar

Agrega el valor `v` al final del array `arr` e incrementa su longitud `len`.

```
template <typename T> void agregar(T arr[], int& len, T v);
```

Función: buscar

Busca la primer ocurrencia de `v` en `arr`; retorna su posición o un valor negativo si `arr` no contiene a `v`. La función `criterio` debe recibir dos parámetros t y k de tipo `T` y `K` respectivamente y retornar un valor negativo, cero o positivo según t sea menor, igual o mayor que k.

```
template <typename T, typename K>
int buscar(T arr[], int len, K v, int (*criterio)(T,K));
```

Función: eliminar

Elimina el valor ubicado en la posición `pos` del array `arr`, decrementando su longitud `len`.

```
template <typename T> void eliminar(T arr[], int& len, int pos);
```

Función: insertar

Inserta el valor `v` en la posición `pos` del array `arr`, incrementando su longitud `len`.

```
template <typename T> void insertar(T arr[], int& len, T v, int pos);
```

Función: insertarOrdenado

Inserta el valor `v` en el array `arr` en la posición que corresponda según el criterio de precedencia que establece la función `criterio`; esta función recibe dos parámetros v1, y v2 ambos de tipo `T` y retorna un valor negativo, cero o positivo según v1 sea menor, igual o mayor que v2 respectivamente.

```
template <typename T>
int insertarOrdenado(T arr[], int& len, T v, int (*criterio)(T,T));
```

Función: buscaEInsertaOrdenado

Busca el valor `v` en el array `arr`; si lo encuentra entonces retorna su posición y asigna `true` al parámetro `enc`. De lo contrario lo inserta donde corresponda según el criterio `criterio`, asigna `false` al parámetro `enc` y retorna la posición en donde finalmente quedó ubicado el nuevo valor.

```
template <typename T>
int buscaEInsertaOrdenado(T arr[],int& len,T v,bool& enc,int
(*criterio)(T,T));
```

Función: ordenar

Ordena el array `arr` según el criterio de precedencia que establece la función `criterio`.

```
template <typename T> void ordenar(T arr[], int len, int
(*criterio)(T,T));
```

Función: busquedaBinaria

Busca el elemento `v` en el array `arr` que debe estar ordenado según el criterio `criterio`. Retorna la posición en donde se encontró el elemento (si se encontró) o la posición en donde dicho elemento podría ser insertado manteniendo el criterio que establece la función `criterio` que recibe cómo parámetro.

```
template<typename T, typename K>
```

```
int busquedaBinaria(T a[], int len, K v, int (*criterio)(T, K), bool&
enc);
```

Operaciones sobre estructuras dinámicas con punteros

El Nodo

```
template <typename T>
struct Nodo
{
      T info;            // valor que contiene el nodo
      Nodo<T>* sig;      // puntero al siguiente nodo
};
```

Operaciones s/Listas

Función: agregarAlFinal

Agrega un nodo con valor `v` al final de la lista direccionada por `p`.

```
template <typename T> void agregar(Nodo<T>*& p, T v);
```

Función: `liberar`

Libera la memoria que insumen todos los nodos de la lista direccionada por `p`; finalmente asigna `NULL` a `p`.

```
template <typename T> void liberar(Nodo<T>*& p);
```

Función: `buscar`

Retorna un puntero al primer nodo de la lista direccionada por `p` cuyo valor coincida con `v`, o `NULL` si ninguno de los nodos contiene a dicho valor. La función `criterio` debe comparar dos elementos t y k de tipo `T` y `K` respectivamente y retornar un valor: menor, igual o mayor que cero según: t<k, t=k o t>k.

```
template <typename T, typename K>
Nodo<T>* buscar(Nodo<T>* p, K v, int (*criterio)(T,K));
```

Función: `eliminar`

Elimina el primer nodo de la lista direccionada por `p` cuyo valor coincida con `v`. La función asume que existe al menos un nodo con el valor que se desea eliminar.

```
template <typename T, typename K>
void eliminar(Nodo<T>*& p, K v, int (*criterio)(T,K));
```

Función: eliminarPrimerNodo

Elimina el primer nodo de la lista direccionada por `p` y returna su valor. Si la lista contiene un único nodo entonces luego de eliminarlo asignará `NULL` a `p`.

```
template <typename T> T eliminarPrimerNodo(Nodo<T>*& p);
```

Función: insertarOrdenado

Inserta un nodo en la lista direccionada por `p` respetando el criterio de ordenamiento que establece la función `criterio`.

```
template <typename T>
Nodo<T>* insertarOrdenado(Nodo<T>*& p, T v, int (*criterio)(T,T));
```

Función: `ordenar`

Ordena la lista direccionada por `p`; el criterio de ordenamiento será el que establece la función `criterio`.

```
template <typename T> void ordenar(Nodo<T>*& p, int (*criterio)(T,T));
```

Función: buscaEInsertaOrdenado

Busca en la lista direccionada por `p` la ocurrencia del primer nodo cuyo valor sea `v`. Si existe dicho nodo entonces retorna su dirección; de lo contrario lo inserta respetando el criterio de ordenamiento establecido por la función `criterio` y retorna la dirección del nodo insertado. Finalmente asigna `true` o `false` al parámetro `enc` según el valor `v` haya sido encontrado o insertado.

```
template <typename T>
Nodo<T>* buscaEInsertaOrdenado(Nodo<T>*& p,T v,bool& enc,int
(*criterio)(T,T));
```

Operaciones sobre pilas

Función: `push` (poner)

Apila el valor `v` a la pila direccionada por `p`.

```
template <typename T> void push(Nodo<T>*& p, T v);
```

Función: `pop` (sacar)

Remueve y retorna el valor que se encuentra en la cima de la pila direccionada por `p`.

```
template <typename T> T pop(Nodo<T>*& p);
```

Operaciones sobre colas (implementación con dos punteros

Función agregar

Agrega el valor `v` en la cola direccionada por `p` y `q`.

```
template <typename T> void encolar(Nodo<T>*& p, Nodo<T>*& q, T v);
```

Función suprimir

Elimina el primer nodo de la cola direccionada por `p` y `q` implementada sobre una lista enlazada.

```
template <typename T> T desencolar(Nodo<T>*& p, Nodo<T>*& p);
```

Opeaciones sobre colas (implementación: lista enlazada circular)

Función encolar

Encola el valor `v` en la cola direccionada por `p`, implementada sobre una lista circular.

```
template <typename T> void encolar(Nodo<T>*& p, T v);
```

Función desencolar

Desencola y retorna el próximo valor de la cola direccionada por `p` e implementada sobre una lista circular.

```
template <typename T> T desencolar(Nodo<T>*& p);
```

Operaciones sobre archivos

Función `read`

Lee un registro de tipo `T` desde el archivo `f`.

```
template <typename T> T read(FILE* f);
```

Función `write`

Escribe el contenido del registro `v`, de tipo `T`, en el archivo `f`.

```
template <typename T> void write(FILE* f, T v);
```

Función `seek`

Mueve el indicador de posición del archivo `f` hacia el registro número `n`.

```
template <typename T> void seek(FILE* f, int n);
```

Función fileSize

Retorna la cantidad de registros que tiene el archivo `f`.

```
template <typename T> long fileSize(FILE* f);
```

Función filePos

Retorna el número de registro que está siendo apuntado por el indicador de posición del archivo.

```
template <typename T> long filePos(FILE* f);
```

Función busquedaBinaria

Busca el valor `v` en el archivo `f`; retorna la posición del registro que lo contiene o -1 si no se encuentra el valor.

```
template <typename T, typename K>
int busquedaBinaria(FILE* f, K v, int (*criterio)(T,K));
```

Operaciones sobre estructuras dinámicas con TADS en struct

El Nodo

```
template <typename T>
struct Nodo
{
	T info;			// valor que contiene el nodo
	Nodo<T>* sig;		// puntero al siguiente nodo
};
```

Operaciones s/Listas

Lista Implementada en struct

```
template <typename T>
struct Lista
{
	Nodo<T>* inicio;		// Puntero al inicio de la lista
	Nodo<T>* final;			// puntero al final de la lista
	Nodo<T>* actual;		// puntero al nodo actual encontrado o insertado
	int cantidad;			// cantidad de nodos en la lista
};
```

Función: agregarAlFinal

Agrega un nodo con valor `v` al final de la lista. Actualiza final, actual, cantidad e inicio si corresponde

```
template <typename T> void agregar(Lista<T> & l, T v);
```

Función: `liberar`

Libera la memoria que insumen todos los nodos de la lista direccionada por `p`; finalmente asigna `NULL` a los punteros de la lista: inicio, actual, final y coloca cero en cantidad.

```
template <typename T> void liberar(Lista<T> & l);
```

Funciones para estructuras enlazadas

Sin Plantilla	Con plantilla

El Nodo
 	
struct Nodo {
int info;
Nodo* sig;
};
Nodo * P = new Nodo();
 	
int * p1 = NULL;
agregarAlFinal1(p1,1);
string* p2 = NULL;
agregarAlFinal2(p2,"Juan");
Alumno * p3 = NULL;
agregarAlFinal3(p1,crear("juan", 123)
Requiere una función diferente para cada tipo de dato.
	El Nodo
	template <typename T>
	struct Nodo {
	T info;
	Nodo**<T>*** sig;
	};
	Nodo **<T>*** P = new Nodo**<T>**();
	
	Nodo<int>* p1 = NULL;
	agregarAlFinal<int>(p1,1);
	Nodo<string>* p2 = NULL;
	agregarAlFinal<string>(p2,"Juan");
	Nodo<Alumno>* p3 = NULL;
	agregarAlFinal<Alumno>(p1,crear("juan", 123);
	Utiliza la misma función para tipos diferentes.

Funciones para pilas	

void push (Nodo* &p, int v)
{
Nodo* nuevo = new Nodo();
nuevo->info = v;
nuevo->sig = p;
p = nuevo;
}
 	
int pop (Nodo* &p)
{
Nodo* aux = p;
int v = aux->info;
p = aux->sig;
delete aux;
return v;
}
	template <typename T>
	void push(Nodo**<T>*** & p, T v)
	{
	Nodo<T>* nuevo = new Nodo<T>();
	nuevo->info = v;
	nuevo->sig = p;
	p = nuevo;
	}
	template <typename T>
	T pop(Nodo**<T>*** & p)
	{
	Nodo<T>* aux = p;
	T v = aux->info;
	P = aux->sig;
	delete aux;
	return v;
	}

Funciones para colas	
void agregar (Nodo* &p, Nodo*& q, int v) int suprimir (Nodo* &p, Nodo* &q, int v)	**template <typename T>** void agregar (Nodo<T>* &p, Nodo<T>*& q, T v) **T** suprimir (Nodo<T>* &p, Nodo<T>* &q, T v)
Funciones para listas	
Nodo* buscarNodo(Nodo* p, int v) Nodo* InsertarOrdenado(Nodo*&p, int v)	**template <typename T, typename K>** Nodo<T>* buscar(Nodo<T>* p, K v, int (*criterio)(T,K)) **template <typename T>** Nodo<T>* insertarOrdenado(Nodo<T>*& p, T v, int (*criterio)(T,T))

Nodo* insertarPrimero(Nodo*& p, int v)	**template <typename T>** Nodo<T>* insertarPrimero(Nodo<T>*& p, T v)
Nodo* insertaDelante(Nodo* &p, int v)	**template <typename T>** Nodo<T>* insertarDelante(Nodo<T>*& p, T v)
Nodo* insertarEnMedio(Nodo* &p, int v)	**template <typename T>** Nodo<T>* insertarEnMedio(Nodo<T>*& p, T v, int (*criterio)(T,T))
Nodo* insertarAlFinal(Nodo* &p, int v)	**template <typename T>** Nodo<T>* insertarDelante(Nodo<T>*& p, T v)
Nodo *InsertarSinRepetir(Nodo* &p, int v)	**template <typename T>** Nodo<T>* insertarSinRepetir(Nodo<T>*& p, T v, int (*criterio)(T,T))
void eliminar(Nodo* &p)	**template <typename T, typename K>** void eliminar(Nodo<T>*& p, K v, int (*criterio)(T,K))
void eliminarPrimerNodo(Nodo* &p)	**template <typename T>** void eliminarPrimerNodo(Nodo<T>*& p)
void liberar(Nodo* &p)	**template <typename T>** void liberar(Nodo<T>*& p)
void ordenar(Nodo * &p)	**template <typename T>** void ordenar(Nodo<T>*& p, int (*criterio)(T,T))

ANEXO 1 Biblioteca Standard C

6

1.1. Definiciones Comunes <stddef.h>

Define, entre otros elementos, el tipo `size_t` y la macro `NULL`. Ambas son definidas, también en otros encabezados, como en `<stdio.h>`.

`size_t`

Tipo entero sin signo que retorna el operador `sizeof`. Generalmente `unsigned int`.

`NULL`

Macro que se expande a un puntero nulo, definido por la implementación. Generalmente el entero cero `0` ó `0L`, ó la expresión constante `(void*)0` .

1.2. Manejo de Caracteres <ctype.h>

`int isalnum (int);`

Determina si el carácter dado `isalpha` o `isdigit` Retorna (ok ? ≠0 : 0).

`int isalpha (int);`

Determina si el carácter dado es una letra (entre las 26 minúsculas y mayúsculas del alfabeto inglés). Retorna (ok ? ≠0 : 0).

`int isdigit (int);`

Determina si el carácter dado es un dígito decimal (entre '0' y '9'). Retorna (ok ? ≠0 : 0).

`int islower (int);`

Determina si el carácter dado es una letra minúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isprint (int);`

Determina si el carácter dado es imprimible, incluye al espacio. Retorna (ok ? ≠0 : 0)

`int isspace (int);`

Determina si el carácter dado es alguno de estos: espacio (`' '`), `'\n'`, `'\t'`, `'\r'`, `'\f'`, `'\v'`. Retorna (ok ? ≠0 : 0)

`int isupper (int);`

Determina si el carácter dado es una letra mayúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isxdigit (int);`

Determina si el carácter dado es un dígito hexadecimal (`'0'`..`'9'`, `'a'`..`'f'` o `'A'`..`'F'`). Retorna (ok ? ≠0 : 0)

`int tolower (int c);`

Si `c` es una letra mayúscula (entre las 26 del alfabeto inglés), la convierte a minúscula.

Retorna (mayúscula ? minúscula : `c`)

`int toupper (int c);`

Si `c` es una letra minúscula (entre las 26 del alfabeto inglés), la convierte a mayúscula.

Retorna (minúscula ? mayúscula : `c`)

1.3. Manejo de Cadenas <string.h>

Define el tipo **size_t** y la macro **NULL**, ver *Definiciones Comunes*.

```
unsigned strlen (const char*);
```

Cuenta los caracteres que forman la cadena dada hasta el 1er carácter **'\0'**, excluido. Retorna (longitud de la cadena).

1.3.1. Concatenación

```
char* strcat (char* s, const char* t);
```

Concatena la cadena **t** a la cadena **s** sobre **s**. Retorna (**s**).

```
char* strncat (char* s, const char* t, size_t n);
```

Concatena hasta **n** caracteres de **t**, previos al carácter nulo, a la cadena **s**; agrega siempre un **'\0'**. Retorna (**s**).

1.3.2. Copia

```
char* strncpy (char* s, const char* t, size_t n);
```

Copia hasta **n** caracteres de **t** en **s**; si la longitud de la cadena **t** es < **n**, agrega caracteres nulos en **s** hasta completar **n** caracteres en total; atención: no agrega automáticamente el carácter nulo. Retorna (**s**).

```
char* strcpy (char* s, const char* t);
```

Copia la cadena **t** en **s** (es la asignación entre cadenas). Retorna (**s**).

1.3.3. Búsqueda y Comparación

```
char* strchr (const char* s, int c);
```

Ubica la 1ra. aparición de **c** (convertido a **char**) en la cadena **s**; el **'\0'** es considerado como parte de la cadena. Retorna (ok ? puntero al carácter localizado : **NULL**)

```
char* strstr (const char* s, const char* t);
```

Ubica la 1ra. ocurrencia de la cadena t (excluyendo al '\0') en la cadena s. Retorna (ok ? puntero a la subcadena localizada : **NULL**).

```
int strcmp (const char*, const char*);
```

Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)

```
int strncmp (const char* s, const char* t, size_t n);
```

Compara hasta **n** caracteres de **s** y de **t**. Retorna (como **strcmp**).

```
char* strtok (char*, const char*);
```

Separa en "tokens" a la cadena dada como 1er. argumento; altera la cadena original; el 1er. argumento es la cadena que contiene a los "tokens"; el 2do. argumento es una cadena con caracteres separadores de "tokens". Retorna (ok ? puntero al 1er. carácter del "token" detectado : **NULL**).

1.3.4. Manejo de Memoria

```
void* memchr(const void* s, int c, size_t n);
```

Localiza la primer ocurrencia de **c** (convertido a un **unsigned char**) en los **n** iniciales caracteres (cada uno interpretado como **unsigned char**) del objeto apuntado por **s**. Retorna (ok ? puntero al carácter localizado : **NULL**).

```
int memcmp (const void* p, const void* q, unsigned n);
```

Compara los primeros **n** bytes del objeto apuntado por **p** con los del objeto apuntado por **q**. Retorna (0 si son iguales; < 0 si el 1ero. es "menor" que el 2do.; > 0 si el 1ero. es "mayor" que el 2do.)

```
void* memcpy (void* p, const void* q, unsigned n);
```

Copia **n** bytes del objeto apuntado por **q** en el objeto apuntado por **p**; si la copia tiene lugar entre objetos que se superponen, el resultado es indefinido. Retorna (**p**).

```
void* memmove (void* p, const void* q, unsigned n);
```

Igual que **memcpy**, pero actúa correctamente si los objetos se superponen. Retorna (**p**).

```
void* memset (void* p, int c, unsigned n);
```

Inicializa los primeros **n** bytes del objeto apuntado por **p** con el valor de **c** (convertido a **unsigned char**). Retorna (**p**).

1.4. Utilidades Generales <stdlib.h>

1.4.1. Tips y Macros

`size_t`
`NULL`
Ver *Definiciones Comunes*.
`EXIT_FAILURE`
`EXIT_SUCCESS`
Macros que se expanden a expresiones constantes enteras que pueden ser utilizadas como argumentos de `exit` ó valores de retorno de `main` para retornar al entorno de ejecución un estado de terminación no exitosa o exitosa, respectivamente.
`RAND_MAX`
Macro que se expande a una expresión constante entera que es el máximo valor retornado por la función `rand`, como mínimo su valor debe ser 32767.

1.4.2. Conversión

```
double atof (const char*);
```
Convierte una cadena que representa un real `double` a número `double`. Retorna (número obtenido, no necesariamente correcto).
```
int atoi (const char*);
```
Convierte una cadena que representa un entero `int` a número `int`. Retorna (número obtenido, no necesariamente correcto) .
```
long atol (const char*);
```
Convierte una cadena que representa un entero `long` a número `long`. Retorna (número obtenido, no necesariamente correcto).
```
double strtod (const char* p, char** end);
```
Convierte como `atof` y, si el 2do. argumento no es `NULL`, un puntero al primer carácter no convertible es colocado en el objeto apuntado por `end`. Retorna como `atof`.
```
long strtol (const char* p, char** end, int base);
```
Similar a `atol` pero para cualquier base entre 2 y 36; si la base es 0, admite la representación decimal, hexadecimal u octal; ver `strtod` por el parámetro `end`. Retorna como `atol`.
```
unsigned long strtoul (const char* p, char** end, int base);
```
Igual que `strtol` pero convierte a `unsigned long`. Retorna como `atol`, pero `unsigned long`.

1.4.3. Administración de Memoria

```
void* malloc (size_t t);
```
Reserva espacio en memoria para almacenar un objeto de tamaño `t`. Retorna (ok ? puntero al espacio reservado : `NULL`)
```
void* calloc (size_t n, size_t t);
```
Reserva espacio en memoria para almacenar un objeto de `n` elementos, cada uno de tamaño `t`. El espacio es inicializado con todos sus bits en cero. Retorna (ok ? puntero al espacio reservado : `NULL`)
```
void free (void* p);
```
Libera el espacio de memoria apuntado por `p`. No retorna valor.
```
void* realloc (void* p, size_t t);
```
Reubica el objeto apuntado por `p` en un nuevo espacio de memoria de tamaño `t` bytes. Retorna (ok ? puntero al posible nuevo espacio : `NULL`).

1.4.4. Números Pseudo-Aleatorios

```
int rand (void);
```
Determina un entero pseudo-aleatorio entre 0 y `RAND_MAX`. Retorna (entero pseudo-aleatorio).
```
void srand (unsigned x);
```

Inicia una secuencia de números pseudo-aleatorios, utilizando a **x** como semilla. No retorna valor.

1.4.5. Comunicación con el Entorno

```
void exit (int estado);
```

Produce una terminación normal del programa. Todos los flujos con *buffers* con datos no escritos son escritos, y todos los flujos asociados a archivos son cerrados. Si el valor de **estado** es **EXIT_SUCCESS** se informa al ambiente de ejecución que el programa terminó exitosamente, si es **EXIT_FAILURE** se informa lo contrario. Equivalente a la sentencia **return estado**; desde la llamada inicial de **main**. Esta función *no retorna a su función llamante*.

```
void abort (void);
```

Produce una terminación anormal del programa. Se informa al ambiente de ejecución que se produjo una terminación no exitosa. Esta función *no retorna a su función llamante*.

```
int system (const char* lineadecomando);
```

Si **lineadecomando** es **NULL**, informa si el sistema posee un procesador de comandos. Si **lineadecomando** no es **NULL**, se lo pasa al procesador de comandos para que lo ejecute.
Retorna (**lineacomando** ? valor definido por la implementación, generalmente el nivel de error del programa ejecutado : (sistema posee procesador de comandos ? ≠0 : 0)).

1.4.6. Búsqueda y Ordenamiento

```
void* bsearch (
     const void* k,
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Realiza una búsqueda binaria del objeto ***k** en un arreglo apuntado por **b**, de **n** elementos, cada uno de tamaño **t** bytes, ordenado ascendentemente. La función de comparación **fc** debe retornar un entero < 0, 0 o > 0 según la ubicación de ***k** con respecto al elemento del arreglo con el cual se compara. Retorna (encontrado ? puntero al objeto : **NULL**).

```
void qsort (
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Ordena ascendentemente un arreglo apuntado por **b**, de **n** elementos de tamaño **t** cada uno; la función de comparación **fc** debe retornar un entero < 0, 0 o > 0 según su 1er. argumento sea, respectivamente, menor, igual o mayor que el 2do. No retorna valor.

1.5. Entrada / Salida <stdio.h>

1.5.1. Tipos

```
size_t
```

Ver *Definiciones Comunes*.

```
FILE
```

Registra toda la información necesitada para controlar un *flujo*, incluyendo su *indicador de posición en el archivo*, puntero asociado a un *buffer* (si se utiliza), un *indicador de error* que registra sin un error de lectura/escritura ha ocurrido, y un *indicador de fin de archivo* que registra si el fin del archivo ha sido alcanzado.

```
fpos_t
```

Posibilita registrar la información que especifica unívocamente cada posición dentro de un archivo.

1.5.2. Macros

NULL
Ver *Definiciones Comunes*.
EOF
Expresión constante entera con tipo **int** y valor negativo que es retornada por varias funciones para indicar *fin de archivo*; es decir, no hay mas datos entrantes que puedan ser leídos desde un *flujo*, esta situación puede ser porque se llegó al fin del archivo o porque ocurrió algún error. Contrastar con **feof** y **ferror**.
SEEK_CUR
SEEK_END
SEEK_SET
Argumentos para la función **fseek**.
stderr
stdin
stdout
Expresiones del tipo **FILE*** que apuntan a objetos asociados con los flujos estándar de error, entrada y salida respectivamente.

1.5.3. Operaciones sobre Archivos

```
int remove(const char* nombrearchivo);
```
Elimina al archivo cuyo nombre es el apuntado por **nombrearchivo**. Retorna (ok ? 0 : ≠0)
```
int rename(const char* viejo, const char* nuevo);
```
Renombra al archivo cuyo nombre es la cadena apuntada por **viejo** con el nombre dado por la cadena apuntada por **nuevo**. Retorna (ok ? 0 : ≠0).

1.5.4. Acceso

```
FILE* fopen (
     const char* nombrearchivo,
     const char* modo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** asociando un flujo con este según el **modo** de apertura. Retorna (ok ? puntero al objeto que controla el flujo : **NULL**).
```
FILE* freopen(
     const char* nombrearchivo,
     const char* modo,
     FILE* flujo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** y lo asocia con el flujo apuntado por **flujo**. La cadena apuntada por **modo** cumple la misma función que en **fopen**. Uso más común es para el redireccionamiento de **stderr**, **stdin** y **stdout** ya que estos son del tipo **FILE*** pero no necesariamente *lvalues* utilizables junto con **fopen**. Retorna (ok ? flujo : **NULL**).
```
int fflush (FILE* flujo);
```
Escribe todos los datos que aún se encuentran en el buffer del flujo apuntado por **flujo**. Su uso es imprescindible si se mezcla **scanf** con **gets** o **scanf** con **getchar**, si se usan varios **fgets**, etc. Retorna (ok ? 0 : **EOF**).
```
int fclose (FILE* flujo);
```
Vacía el *buffer* del flujo apuntado por **flujo** y cierra el archivo asociado. Retorna (ok ? 0 : **EOF**)

1.5.5. Entrada / Salida Formateada

Flujos en General

```
int fprintf (FILE* f, const char* s, ...);
```
Escritura formateada en un archivo ASCII. Retorna (ok ? cantidad de caracteres escritos : < 0).

```
int fscanf (FILE* f, const char*, ...);
```
Lectura formateada desde un archivo ASCII. Retorna (cantidad de campos almacenados) o retorna (**EOF** si detecta fin de archivo).

Flujos stdin y stdout

```
int scanf (const char*, ...);
```
Lectura formateada desde **stdin**. Retorna (ok ? cantidad de ítems almacenados : **EOF**).
```
int printf (const char*, ...);
```
Escritura formateada sobre **stdout**. Retorna (ok ? cantidad de caracteres transmitidos : < 0).

Cadenas

```
int sprintf (char* s, const char*, ...);
```
Escritura formateada en memoria, construyendo la cadena **s**. Retorna (cantidad de caracteres escritos).
```
int sscanf (const char* s, const char*, ...);
```
Lectura formateada desde una cadena **s**. Retorna (ok ? cantidad de datos almacenados : **EOF**).

1.5.6. Entrada / Salida de a Caracteres

```
int fgetc (FILE*); ó
int getc (FILE*);
```
Lee un carácter (de un archivo ASCII) o un byte (de un archivo binario). Retorna (ok ? carácter/byte leído : **EOF**).
```
int getchar (void);
```
Lectura por carácter desde **stdin**. Retorna (ok ? próximo carácter del buffer : **EOF**).
```
int fputc (int c, FILE* f); ó
int putc (int c, FILE* f);
```
Escribe un carácter (en un archivo ASCII) o un byte (en un archivo binario). Retorna (ok ? **c** : **EOF**).
```
int putchar (int);
```
Eescritura por carácter sobre stdout. Retorna (ok ? carácter transmitido : **EOF**).
```
int ungetc (int c, FILE* f);
```
"Devuelve" el carácter o byte **c** para una próxima lectura. Retorna (ok ? **c** : **EOF**).

1.5.7. Entrada / Salida de a Cadenas

```
char* fgets (char* s, int n, FILE* f);
```
Lee, desde el flujo apuntado **f**, una secuencia de a lo sumo **n**-1 caracteres y la almacena en el objeto apuntado por **s**. No se leen más caracteres luego del carácter nueva línea o del fin del archivo. Un carácter nulo es escrito inmediatamente después del último carácter almacenado; de esta forma, s queda apuntando a una cadena. Importante su uso con **stdin**. Si leyó correctamente, **s** apunta a los caracteres leídos y retorna **s**. Sí leyó sólo el fin del archivo, el objeto apuntado por **s** no es modificado y retorna **NULL**. Si hubo un error, contenido del objeto es indeterminado y retorna **NULL**. Retorna (ok ? **s** : **NULL**).
```
char* gets (char* s);
```
Lectura por cadena desde **stdin**; es mejor usar **fgets**() con **stdin** . Retorna (ok ? **s** : **NULL**).
```
int fputs (const char* s, FILE* f);
```
Escribe la cadena apuntada por **s** en el flujo **f**. Retorna (ok ? último carácter escrito : **EOF**).
```
int puts (const char* s);
```
Escribe la cadena apuntada por **s** en **stdout**. Retorna (ok ? ≥ 0 : **EOF**).

1.5.8. Entrada / Salida de a Bloques

```
unsigned fread (void* p, unsigned t, unsigned n, FILE* f);
```
Lee hasta **n** bloques contiguos de **t** bytes cada uno desde el flujo **f** y los almacena en el objeto apuntado por **p**. Retorna (ok ? **n** : < **n**).
```
unsigned fwrite (void* p, unsigned t, unsigned n, FILE* f);
```

Escribe n bloques de t bytes cada uno, siendo el primero el apuntado por p y los siguientes, sus contiguos, en el flujo apuntado por f. Retorna (ok ? **n** : < **n**).

1.5.9. Posicionamiento

```
int fseek (
     FILE* flujo,
     long desplazamiento,
     int desde
);
```

Ubica el *indicador de posición de archivo* del flujo binario apuntado por **flujo**, **desplazamiento** caracteres a partir de **desde**. **desde** puede ser **SEEK_SET**, **SEEK_CUR** ó **SEEK_END**, comienzo, posición actual y final del archivo respectivamente. Para flujos de texto, **desplazamiento** deber ser cero o un valor retornado por **ftell** y **desde** debe ser **SEEK_SET**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fsetpos (FILE* flujo, const fpos_t* posicion);
```

Ubica el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** según el valor del objeto apuntado por **posicion**, el cual debe ser un valor obtenido por una llamada exitosa a **fgetpos**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fgetpos (FILE* flujo, fpos_t* posicion);
```

Almacena el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** en el objeto apuntado por **posicion**, cuyo valor tiene significado sólo para la función **fsetpos** para el restablecimiento del *indicador de posición de archivo* al momento de la llamada a **fgetpos**. Retorna (ok ? 0 : ≠ 0).

```
long ftell (FILE* flujo);
```

Obtiene el valor actual del *indicador de posición de archivo* para el flujo apuntado por **flujo**. Para flujos binarios es el número de caracteres (bytes ó posición) desde el comienzo del archivo. Para flujos de texto la valor retornado es sólo útil como argumento de **fseek** para reubicar el indicador al momento del llamado a **ftell**. Retorna (ok ? indicador de posición de archivo : −1L).

```
void rewind(FILE *stream);
```

Establece el indicador de posición de archivo del flujo apuntado por **flujo** al principio del archivo. Semánticamente equivalente a **(void)fseek(stream, 0L, SEEK_SET)**, salvo que el indicador de error del flujo es desactivado. No retorna valor.

1.5.10. Manejo de Errores

```
int feof (FILE* flujo);
```

Chequea el *indicador de fin de archivo* del flujo apuntado por **flujo**. Contrastar con la macro **EOF** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de fin de archivo* activado ? ≠ 0 : 0).

```
int ferror (FILE* flujo);
```

Chequea el *indicador de error* del flujo apuntado por **flujo** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de error* activado ? ≠ 0 : 0).

```
void clearerr(FILE* flujo);
```

Desactiva los indicadores de fin de archivo y error del flujo apuntado por flujo. No retorna valor.

```
void perror(const char* s);
```

Escribe en el flujo estándar de error (**stderr**) la cadena apuntada por **s**, seguida de dos puntos (**:**), un espacio, un mensaje de error apropiado y por último un carácter nueva línea (**\n**). El mensaje de error está en función a la expresión **errno**. No retorna valor.

1.6. Otros

1.6.1. Hora y Fecha <time.h>

`NULL`
`size_t`
Ver *Definiciones Comunes*.
`time_t`
`clock_t`
Tipos aritméticos capaces de representar el tiempo. Generalmente `long`.
`CLOCKS_PER_SEC`
Macro que expande a una expresión constante de tipo `clock_t` que es el número por segundos del valor retornado por la función `clock`.
`clock_t clock(void);`
Determina el tiempo de procesador utilizado desde un punto relacionado con la invocación del programa. Para conocer el valor en segundos, dividir por `CLOCKS_PER_SEC`. Retorna (ok ? el tiempo transcurrido: `(clock_t)(-1)`.
`char* ctime (time_t* t);`
Convierte el tiempo de `*t` a fecha y hora en una cadena con formato fijo. Ejemplo: Mon Sep 17 04:31:52 1973\n\0. Retorna (cadena con fecha y hora).
`time_t time (time_t* t);`
Determina el tiempo transcurrido en segundos desde la hora 0 de una fecha base; por ejemplo: desde el 1/1/70. Retorna (tiempo transcurrido). Si `t` no es `NULL`, también es asignado a `*t`.

1.6.2. Matemática

`int abs(int i);`
`long int labs(long int i);`
`<stdlib.h>` Calcula el valore del entero `i`. Retorna (valor absoluto de `i`).
`double ceil (double x);`
`<math.h>` Calcula el entero más próximo, no menor que `x`. Retorna (entero calculado, expresado como `double`).
`double floor (double x);`
`<math.h>` Calcula el entero más próximo, no mayor que `x`. Retorna (entero calculado, expresado como `double`).
`double pow (double x, double z);`
`<math.h>` Calcula x^z; hay error de dominio si $x < 0$ y z no es un valor entero, o si x es 0 y $z \neq 0$. Retorna (ok ? x^z : error de dominio o de rango).
`double sqrt (double x);`
`<math.h>` Calcula la raíz cuadrada no negativa de `x`. Retorna ($x \geq 0.0$? raíz cuadrada : error de dominio).

ANEXO 2 Flujos de texto y binario C++

7

C++ ifstream, ofstream y fstream

C++ para el acceso a ficheros de texto ofrece las clases ifstream, ofstream y fstream. (**i** input, **f** file y **stream**). (**o** output). fstream es para i/o.

Abrir los ficheros

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
/* pasando parámetros en la declaración de la variable */
   ifstream f("fichero.txt", ifstream::in);

/* se abre después de declararlo, llamando a open() */
   ofstream f2;
   f2.open("fichero2.txt", ofstream::out);
```

El primer parámetro es el nombre del fichero, el segundo el modo de apertura.

- **app** (**app**end) Para añadir al final del fichero. Todas las escrituras se hacen al final independiente mente de la posición del puntero.
- **ate** (**at e**nd). Para añadir al final del fichero. En caso de mover el puntero, la escritura se hace donde esta el mismo.
- **binary** (**binary**) Se abre el fichero como fichero binario. Por defecto se abre como fichero de texto.
- **in** (**in**put) El fichero se abre para lectura.
- **out** (**out**put) El fichero se abre para escritura
- **trunc** (**trunc**ate) Si el fichero existe, se ignora su contenido y se empieza como si estuviera vacio. Posiblemente perdamos el contenido anterior si escribimos en él.

Se puede abrir con varias opciones con el operador OR o el caracter **|** .

```
f2.open("fichero2.txt", ofstream::out | ofstream::trunc);
```

Hay varias formas de comprobar si ha habido o no un error en la apertura del fichero. La más cómoda es usar el operator ! que tienen definidas estas clases. Sería de esta manera

```
if (f)
{
   cout << "fallo" << endl;
```

```
   return -1;
}
```

!f (no f) retorna true si ha habido algún problema de apertura del fichero.

Leer y escribir en el fichero

Existen metodos específicos para leer y escribir bytes o texto: get(), getline(), read(), put(), write(). Tambien los operadores **<<** y **>>**.

```
/* Declaramos un cadena para leer las líneas */
char cadena[100];
...
/* Leemos */
f >> cadena;
...
/* y escribimos */
f2 << cadena;
/*Copiar un archivo en otro */
/* Hacemos una primera lectura */
f >> cadena;/*Lectura anticipada controla si es fin de archivo*/
while (!f.feof()){
   /* Escribimos el resultado */
   f2 << cadena << endl;
   /* Leemos la siguiente línea */
   f >> cadena;
}
```

Cerrar los ficheros

```
f.close(); f2.close();
```

Ejemplos Archivos de texto

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   char cadena[128];
   // Crea un fichero de salida
   ofstream fs("nombre.txt");
   // Enviamos una cadena al fichero de salida:
   fs << "Hola, mundo" << endl;
   // Cerrar el fichero para luego poder abrirlo para lectura:
   fs.close();

   // Abre un fichero de entrada
   ifstream fe("nombre.txt");
   // Lectura mediante getline
   fe.getline(cadena, 128);
   // mostrar contenido por pantalla
   cout << cadena << endl;

   return 0;
}

int main() {
   char cadena[128];
   ifstream fe("streams.cpp");
   while(!fe.eof()) {
```

```
      fe >> cadena;
      cout << cadena << endl;
   }
   fe.close();
   return 0;}
```

Ejemplo Archivo binario

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
struct tipoReg {
   char nombre[32];
   int edad;
   float altura;
};

int main() {
   tipoReg r1;
   tipoReg r2;
   ofstream fsalida("prueba.dat",ios::out | ios::binary);
   strcpy(r1.nombre, "Juan");
   r1.edad = 32;
   r1..altura = 1.78;

   fsalida.write(reinterpret_cast<char *>(&r1),    sizeof (tipoReg));

fsalida.close();// lo cerramos para abrirlo para lectura

   ifstream fentrada("prueba.dat", ios::in | ios::binary);

   fentrada.read(reinterpret_cast<char *>(&r2), sizeof(tipoReg));
   cout << r2.nombre << endl;
   cout << r2.edad << endl;
   cout << r2.altura << endl;
   fentrada.close();

   return 0;
}
```

Acceso directo

```
#include <fstream>
using namespace std;
int main() {
   int i;
   char mes[][20] = {"Enero", "Febrero", "Marzo","Abril", "Mayo",
"Junio", "Julio", "Agosto", "Septiembre", "Octubre", "Noviembre",
      "Diciembre"};
   char cad[20];
```

```
   ofstream fsalida("meses.dat",ios::out | ios::binary);

   // Crear fichero con los nombres de los meses:
   cout << "Crear archivo de nombres de meses:" << endl;
   for(i = 0; i < 12; i++)
      fsalida.write(mes[i], 20);
   fsalida.close();

   ifstream fentrada("meses.dat", ios::in | ios::binary);

   // Acceso secuencial:
   cout << "\nAcceso secuencial:" << endl;
   fentrada.read(cad, 20);
   do {
      cout << cad << endl;
      fentrada.read(cad, 20);
   } while(!fentrada.eof());

   fentrada.clear();
   // Acceso aleatorio:
   cout << "\nAcceso aleatorio:" << endl;
   for(i = 11; i >= 0; i--) {
      fentrada.seekg(20*i, ios::beg);
      fentrada.read(cad, 20);
      cout << cad << endl;
   }

   // Calcular el número de elementos
   // almacenados en un fichero:
   // ir al final del fichero
   fentrada.seekg(0, ios::end);
   // leer la posición actual
   pos = fentrada.tellg();
   // El número de registros es el tamaño en
   // bytes dividido entre el tamaño del registro:
   cout << "\nNúmero de registros: " << pos/20 << endl;
   fentrada.close();

   return 0;
}
```

Para el acceso aleatorio seekp y seekg, que permiten cambiar la posición del fichero en la que se hará la siguiente escritura o lectura. La 'p' es de put y la 'g' de get, es decir escritura y lectura, respectivamente. Otras funciones relacionadas con el acceso aleatorio son tellp y tellg, que indican la posición del puntero en el fichero.

La función seekg permite acceder a cualquier punto del fichero, no tiene por qué ser exactamente al principio de un registro, la resolución de la funciones seek es de un byte.

La función seekp nos permite sobrescribir o modificar registros en un fichero de acceso aleatorio de salida. La función tellp es análoga a tellg, pero para ficheros de salida.

ANEXO 3 Contenedores en C++ Vector

8

El contenedor “vector” en c++

```
#include <vector>
vector<int> valores (5);     // Declara un vector de 5 integers
vector<float> temps (31);    // Declara vector de 31 floats
```

Esta estructura combina la performance del acceso a un array al estilo C pero con los beneficios de la poseer una variable que maneja toda la lógica para un contenedor.

```
vector<int> notas(30);
for (vector<int>::size_type i = 0; i < 30; i++)
{
    cout << "Ingrese la nota para el estudiante n° " << i+1
         << ": ";
    cin >> notas[i];
}
```

Establecer su tamaño a través de Resize()

Al ser una estructura dinámica, cualquier variable de tipo vector puede aumentar o disminuir su capacidad bajo demanda.

```
vector<int> notas;
int qestudiantes;
cout << "Ingrese la cantidad de estudiantes: " << endl;
cin >> qestudiantes;
notas.resize (qestudiantes);
for (vector<int>::size_type i = 0; i < notas.size(); i++)
{
    cout << "Ingrese la nota para el estudiante nro " << i+1
         << ": ";
    cin >> notas[i];
}
```

El método “resize” redefine el tamaño del vector al tamaño que se le indica a través del valor pasado por parámetro.

Modificación a medida que se agregan valores con pushback()

El ejemplo anterior no funciona si se quisiera aumentar el tamaño del vector a medida que se agregan valores. El método pushback crea un espacio al final del vector e inserta el valor pasado por parámetro en él, similar al push de una cola.

```
vector<int> notas;
int nota;
char opcion;
do
{
    cout << "Ingrese la nota: " << endl;
    cin >> nota;

    notas.push_back (nota);

    cout << "Desea ingresar otra nota (s/n)? " << endl;
    cin >> opcion;
} while (opcion == 's' || opcion == 'S');
```

Inserción y eliminación de elementos – insert() y erase()

El prototipo del método insert() es el siguiente:

Identificador.insert(identificador.begin() + posición_a_insertar, valor_a_insertar)

```
notas.insert(notas.begin() + 2, 10); // Agrega un 10 en la posición 2
```

Para eliminar un elemento se utiliza el método erase, cuyo prototipo es:

Identificador.erase(identificador.begin() + posición_a_remover)

```
notas.erase(notas.begin() + 2);
```

Algoritmos STL

La ventaja de la utilización de las clases STL es la existencia de algoritmos que resuelven una gran mayoría de las funcionalidades requeridas para las estructuras de datos mayormente utilizadas, los vectores no son la excepción.

```
#include <algorithm>
```

Esta inclusión, permite utilizar las siguientes funciones entre otras:

Prototipo	Devuelve	Acción
sort (identificador.begin(), identificador.end());	-	Ordena el vector de menor a mayor
reverse (identificador.begin(), identificador.end());	-	Invierte el vector
count (identificador.begin(), identificador.end(), valor_buscado);	Entero	Cuenta la cantidad de ocurrencias de valor_buscado
max_element (idetificador.begin(), identificador.end());	Iterator al mayor valor	Busca el máximo valor
min_element (identificador.begin(), identificador.end());	Iterator al menor valor	Busca el mínimo valor