Algoritmos y estructura de datos

Asignatura anual, código 082021

MODULO 4 Vectores y matrices

Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional FRBA

Dr. **Oscar Ricardo Bruno**,
Ing. **Pablo Augusto Sznajdleder**.
Ing. **Jose Maria Sola**
Ing. **Yamila Zakhen**
Ing. **Natalia Perez**

Tabla de contenido

Arreglos y registros

1

Tipos de Datos

Como ya hemos visto, el objeto de estudio de Algoritmos y Estructura de datos esta centrado básicamente en eso, en los algoritmos, que refiere al "…*conjunto de finito de reglas, ordenadas de forma lógica y precisa para la solución de un problema, con utilización o no de un computador*…" y la estructura de datos. Los tipos de datos Identifican o determinan un dominio de valores y un 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.
- Se define como posiciones contiguas de memoria de tipos no homogéneos. Es, entonces, 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, su declaración es:

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

Las estructuras pueden ser anidadas. 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;    //       define un identificador con la estructura declarada.
```

En el caso de la definición precedente, Alumno es un registro (struct para C) con tres miembros (campos) uno de los cuales es un registro (struct) 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, en principio, 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. 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 lista o vector.
- El arreglo con 2 o más índices o dimensiones es una tabla o matriz. Un grupo de elementos homogéneo con un orden interno en el que se necesitan 2 o más 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.
TipoAlumno MatrizAlumno[10][5] //declara matriz o tabla de 10 filas y y columnas
```

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

Vector de 5 componentes
int V[5]

V[0]
V[1]
V[2]
V[3]
V[4]

Matriz de 5 filas y tres columnas
int M[5] [3]

M[0][0]	M[0][1]	M[0][2]
M[1][0]	M[1][1]	M[1][2]
M[2][0]	M[2][1]	M[2][2]
M[3][0]	M[3][1]	M[3][2]
M[4][0]	M[4][1]	M[4][2]

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

  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
```

Analisis comparativo de estructuras

Los vectores permiten manejar conjunto de datos del mismo tipo. En el modulo anterior hemos visto otra estructura (el archivo) que también permite manejar conjunto de datos del mismo tipo. Ambas estructuras tienen sus propiasn particularidades y, para la resolución de los problemas, debemos decidir cual de ellas es la más adecuada a los efector de resolver los problemas planteados. Por ejemplo, si el dato debe persistir más alla de la aplicación no hay duda que el almacenamiento debe ser físico, en un archivo. Pero si se trata de buscar, reordenar o priorizar la velocidad de procesamiento, allí la elección debe darse hacia estructuras de almacenamiento electrónico, en este caso los vectores o matrices. El cuadro que sigue busca caracterizar estas estructuras para poder tomar la mejor decisión en el momento de la selección.

Caracteristica	Archivo	Vector
Almacenamiento	Fisico	Electronico
Procesamirnto	Lento	Rapido
Persistencia al fin de la aplicacion	SI	NO
Tamaño en tiempo ejecucion	Variable	Fijo
Busqueda	Directa Binaria Secuencial (Ineficiente)	Directa Binaria Secuencial
Carga	Directa Agregando al final	Directa Agregando al final Agregando en orden
Ordenamiento	Con Pos. Unica Predecible	Con Pos. Unica Predecible Moetodos de ordenamiento
Recorridos	Completo Con corte de control Apareando	Completo Con corte de control Apareando
Carga sin repetir la clave	NO (salvo el caso de PUP)	SI
Busqueda de los N Mejores	NO	SI
Utilizacion como est. auxiliar	NO (salvo el caso de PUP)	Es su uso mas frecuente

Desde el punto de vista de la algoritmia se puede sostener si es necesario buscar o modificar el orden entre los datos de entrada y salida lo mas adecuado es tratar con estructuras auxiliares con almacenamiento electrónico. Esto es priorizar ARRAY sodre ARCHIVO. Desde la eficiencia priorizar para la carga o la busquda DIRECTA, BINARIA, SECUENCIAL (esta ultima solo en array).

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;

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

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

FIN. // Búsqueda secuencial En Vector

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;

FIN. // Búsqueda minimo distinto de cero En Vector
```

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

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

FIN_SI

PARA j [2, U] HACER
        SI (v[j] > Maximo)
```

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 posición y segundo el de la primera.

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

```
        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
```

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)
        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);
```

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

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

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 identificador que controla el tamaño logico

```
                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
```

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) HA
            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
            FIN_SI
        FIN_MIENTRAS;
```

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 salir

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 ultimo el anterior al de la mitad actual

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

```
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
```

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

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.

```
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;
```

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 N – 1 siendo n la cantidad de elementos

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

```
                FIN_PARA;
        FIN_PARA;
FIN
```

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 ;
                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
```

Implementaciones C C++

Operaciones sobre arrays

Antes de comenzar

Se presentan las principales operaciones que son generalmente utilizadas para el manejo de arrays. Además se incorpora la utilización de tipos de datos genéricos, implementados con templates, y también la importancia de poder desacoplar las porciones de código que son propias de un problema, la generalidad de la algoritmia a través de 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. Se pueden crear apuntadores a funciones, en lugar de direccionar datos los punteros a funciones apuntan a código ejecutable. Un puntero a una función es un apuntador cuyo valor es el nombre de la función.

Sintaxis
TipoRetorno (*PunteroFuncion)(ListaParametros)
int f(int); //declara la funcion f
int (*pf)(int); // define pf a funcion int con argumento int
pf = f; // asigna la direccon de f a pf

Los apuntadores permiten pasar una función como argumento a otra función. 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 desarrollada mas abajo. 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.
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`.

```
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;
}
```

ANEXO Contenedores en C++ Vector

3

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	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

Ejercicios con vectores y matrices

1. Ingresar un valor N (< 25). Generar un arreglo de N componentes en el cual las mismas contengan los primeros números naturales pares e imprimirlo.
2. Ingresar un valor entero N (< 30) y a continuación un conjunto de N elementos. Si el último elemento del conjunto tiene un valor menor que 10 imprimir los negativos y en caso contrario los demás.
3. Ingresar un valor entero N (< 20). A continuación ingresar un conjunto VEC de N componentes. A partir de este conjunto generar otro FACT en el que cada elemento sea el factorial del elemento homólogo de VEC. Finalmente imprimir ambos vectores a razón de un valor de cada uno por renglón
4. Ingresar un valor entero N (< 25). A continuación ingresar un conjunto VEC de N componentes. Si la suma de las componentes resulta mayor que cero imprimir las de índice impar, sino los otros elementos.
5. Ingresar un valor entero N (< 30). A continuación ingresar un conjunto UNO y luego otro conjunto DOS, ambos de N componentes. Generar e imprimir otro conjunto TRES intercalando los valores de posición impar de DOS y los valores de posición par de UNO.
6. Ingresar un valor entero N (< 40). A continuación ingresar un conjunto VALOR de N elementos. Determinar e imprimir el valor máximo y la posición del mismo dentro del conjunto. Si el máximo no es único, imprimir todas las posiciones en que se encuentra.
7. Ingresar un valor entero N (< 15). A continuación ingresar un conjunto DATO de N elementos. Generar otro conjunto de dos componentes MEJORDATO donde el primer elemento sea el mayor valor de DATO y el segundo el siguiente mayor (puede ser el mismo si está repetido). Imprimir el conjunto MEJORDATO con identificación.
8. Ingresar un valor entero N (< 25). A continuación ingresar un conjunto GG de N elementos. Imprimir el arreglo en orden inverso generando tres estrategias para imprimir los elementos a razón de: a) Uno por línea, b) Diez por línea, c) Cinco por línea con identificación
9. Ingresar un valor entero N (< 40). A continuación ingresar un conjunto A y luego otro conjunto B ambos de N elementos. Generar un arreglo C donde cada elemento se forme de la siguiente forma: C[1] = A[1]+B[N] C[2] = A[2]+B[N-1]
10. Ingresar dos valores enteros M (< 10) y N (< 15). A continuación ingresar un conjunto A de M elementos y luego otro B de N elementos. Generar e imprimir:
    a) Un conjunto C resultante de la anexión de A y B.
    b) Un conjunto D resultante de la anexión de los elementos distintos de cero de A y B.
11. Ingresar dos valores enteros M (< 25) y N (< 10) A continuación ingresar un conjunto A de M elementos y luego otro B de N elementos, ambos ordenados en forma creciente por magnitud. Generar e imprimir el conjunto TOTAL resultante del apareo por magnitud de los conjuntos A y B.
12. Ingresar un valor entero N (< 40). Luego ingresar un conjunto REFER de N elementos reales (ingresan ordenados por magnitud creciente). Finalmente ingresar un valor pesquisa X. Desarrollar el programa que determine e imprima:
    a) Con cual elemento (posición) del conjunto coincide, o
    b) Entre cuales dos elementos (posiciones) se encuentra, o
    c) Si es menor que el primero o mayor que el último.
13. Ingresar un valor entero CANT (< 50) y a continuación un conjunto SINOR de CANT elementos. Desarrollar un programa que determine e imprima:
    a) El conjunto SINOR en el que cada elemento original se intercambie por su simétrico: A[1] con A[CANT], A[2] con A[N-1], ... etc.
    b) El conjunto SINOR ordenado de menor a mayor sobre si mismo indicando la posición que ocupaba cada elemento en el conjunto original.

Ejercicios que integran estructuras de datos array y archivos

14. Dado un archivo PRECIOS (desordenado) con los precios de cada articulo, donde cada registro contiene:
    a) Nro. de artículo (5 dígitos)
    b) Descripción del artículo (19 caracteres)
    c) Precio por unidad (real)
    d) Cantidad en stock (5 dígitos) e) Nro. de proveedor (4 dígitos)

    Desarrollar el programa que imprima el contenido del archivo ordenado por:
    1) Nro. de articulo creciente
    2) Descripción del articulo (alfabético creciente)
    3) Nro. de proveedor creciente y dentro del mismo por Nro. de articulo creciente
    4) Nro. de proveedor creciente y dentro del mismo por Nro. de articulo decreciente
15. Una empresa de aviación realiza 500 vuelos semanales a distintos puntos del país y requiere un programa para el otorgamiento de pasajes. Para ello dispone de un archivo de registros, en el que cada registro contiene información de los vuelos que realiza y la cantidad de pasajes disponibles en cada uno de ellos según se indica:
    a) Código del vuelo (6 dígitos)
    b) Cantidad de pasajes disponibles (3 dígitos)

    Se dispone además de otro archivo con los datos de los potenciales compradores, en el que cada registro tiene:
    c) Código de vuelo solicitado
    d) Cantidad de pasajes solicitados (3 dígitos)
    e) DNI del solicitante (8 dígitos)
    f) Apellido y nombres del solicitante (30 caracteres)

    Desarrollar estrategia, algoritmo y codificación del programa que determine e imprima:
    1) Para los solicitantes a los cuales se les venden pasajes,

DNI	APELLIDO Y NOMBRES	CANTIDAD DE PASAJES	CÓDIGO VUELO
99999999	XXXXXXXXXXXXXXXXXXX	999	999

    Al final del proceso el siguiente listado:

CÓDIGO DE VUELO	PASAJES LIBRES	PASAJES NO VENDIDOS
999999	999	999

    Se le vende al solicitante si la cantidad de pasajes que solicita está disponible, en caso contrario se computa como pasajes no vendidos.
16. Se desarrolla una carrera automovilística de regularidad constituida por 50 trayectos numerados de 1 a 50. Por cada trayecto se generó un registro con el número de trayecto y el tiempo asignado en segundos y se encuentran en el archivo ASIGNADO (sin ningún orden) con: a) Nro. del trayecto b) Tiempo asignado en segundos

    Para llevar el control de los corredores, de posición y de abandonos se dispone de un archivo TIEMPO donde cada registro contiene: a) Nro. del corredor (3 dígitos) b) Nro. del trayecto, c) Tiempo en segundos.

    Los registros de este archivo están ordenados por trayecto pero no por corredor. A partir del abandono de un corredor en un trayecto no habrá más registros para el en el archivo.

    Desarrollar estrategia, algoritmo y codificación del programa que determine e imprima:
    1) Por cada etapa, su número y el del corredor ganador de la misma.
    2) Por cada etapa, su número y los de los corredores que abandonan en la misma.

17. Una empresa que distribuye mercadería hacia distintas localidades del interior dispone de dos archivos de registros: Uno denominado DESTINOS con información de la distancia a cada uno de los destinos: a) Nro. de destino (3 dígitos) b) Distancia en kilómetros (NNN.NNN). Otro denominado VIAJES con los viajes realizados por cada camión (< 200), donde cada registro contiene: a) Patente del camión (6 caracteres) b) Nro. de destino c) Nro. de chofer (1 a 150). Desarrollar estrategia, algoritmo y codificación del programa que determine e imprima:
    1) Cantidad de viajes realizados a cada destino (solo si > 0).
    2) Nro. de chofer con menor cantidad de Km (entre los que viajaron).
    3) Patente de los camiones que viajaron al destino 116 sin repeticiones de las mismas.

18. Ejercicio Nro. 54:

19. Ingresar dos valores, M (< 30) y N (< 25) y a continuación por filas todos los componentes de una matriz MATRIZA de M filas y N columnas. Desarrollar un programa que:
    a) Imprima la matriz MATRIZA por columnas.
    b) Calcule e imprima el valor promedio de los componentes de la matriz.
    c) Genere e imprima un vector VECSUMCOL donde cada componente sea la suma de la columna homóloga.
    d) Genere e imprima un vector VECMAXFIL donde cada componente sea el valor máximo de cada fila.

20. Ejercicio Nro. 55:

21. Ingresar un valor N (< 25) y luego por filas una matriz cuadrada CUADRA de N filas y columnas. Desarrollar un programa que determine e imprima:
    a) Todos los elementos de la diagonal principal o secundaria según de cual resulte mayor la sumatoria de elementos.
    b) Los elementos del cuarto (N/2 filas y N/2 columnas) cuya sumatoria resulte mayor (considerando que N fuera par).
    c) Los elementos de la triangular superior o inferior dependiendo de cual tenga mayor sumatoria de elementos.
22. Ingresar dos valores, M (< 20) y N (< 25) y a continuación por columnas todos los componentes de una matriz DESORDE de M filas y N columnas. Desarrollar un programa que:
    a) Ordene (creciente) cada columna de la matriz sobre si misma y la imprima a razón de una columna por renglón.
    b) Ordene (creciente) la matriz sobre si misma por fila desde el elemento 1,1 al M,N y la imprima a razón de una fila por renglón.
23. Ingresar por plano, fila y columna todos los elementos de una matriz MATRIDIM de M planos, filas y columnas. Desarrollar un programa que:
    a) Imprima la matriz MATRIDIM por columnas, fila, plano.
    b) Calcule e imprima el valor promedio de la matriz.
    c) Determine e imprima el mayor valor y en que lugar del la matriz se encuentra.
    d) Genere e imprima una matriz MATCSUMCOL donde cada elemento sea la suma de la columna homóloga.
    e) Genere e imprima una matriz MATMAXFIL donde cada elemento sea el valor máximo de cada fila.