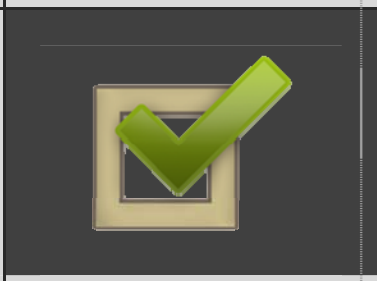
ALGORITMOS Y ESTRUCTURA DE DATOS



Biblioteca de cadenas de caracteres

Funciones

Conversión entre números y cadenas

- `string intToString(int n)`

Retorna una cadena compuesta por todos los dígitos el parámetro *n*.

Ejemplo: `intToString(123)` **retorna: "123".**

- `int stringToInt(string s)`

Retorna un valor entero que surge de interpretar a la cadena `s` **como un valor numérico.**

Ejemplo: `stringToInt("123")` **retorna: 123.**

- `string doubleToString(double d)`

Retorna una cadena compuesta por todos los dígitos del parámetro *d*, incluyendo al separador decimal.

Ejemplo: `doubleToString(123.45)` **retorna: "123.45".**

- `double stringToDouble(double d)`

Retorna un valor flotante que surge de interpretar a la cadena `s` **como un valor numérico real, con decimales.**

Ejemplo: `stringToDouble("123.45")` **retorna: 123.45.**

- `string charToString(char c)`

Retorna un `sring` **de longitud 1 conteniendo únicamente al carácter** `c`**.**

Ejemplo: `charToString('A')` **retorna: "A".**

- `char stringToChar(string s)`

Retorna un `char` **que coincide con el primer carácter de la cadena** `s`**.**

Ejemplo: `stringToChar("A")` **retorna: 'A'.**

Tratamiento de tokens

Sea `s` **una cadena y** `c` **un carácter, entonces llamaremos token a toda subcadena de** `s` **que se encuentre encerrada entre dos caracteres** `c` **o entre el inicio de** `s` **y la primer ocurrencia de** `c` **o entre la última ocurrencia de** `c` **y el final de la cadena** `s`**.**

Por ejemplo, si `c` **es el carácter '|' y** `s` **es la cadena: "John|Paul|George|Ringo", los *tokens* que podremos extraer serán: "John", "Paul", "George" y "Ringo", en ese orden. Pero si** `c` **fuera el carácter 'o', de la misma**

cadena `s` podríamos extraer los *tokens*; "J", "hn|Paul|Ge", "rge|Ring" y ""; en este caso, el último *token* consiste en una cadena vacía.

- `bool tokenizer(string s,char sep,string& token,int& aux)`

La función retornará `true` mientras queden *tokens* por extraer y en cada invocación asignará al parámetro `token` el *token* que corresponda.

- `int tokenCount(string s, char sep)`

Retorna la cantidad de *tokens* que el separador `sep` genera en la cadena `s`.

`void addToken(string& s, string t, char sep)`

Agrega el *token* `t` al final de la cadena `s`.

- `string getTokenAt(string s, int i,char sep)`

Dada la cadena `s` y el separador `sep`, retorna el *token* que se ubica en la *i-ésima* posición.

- `void setTokenAt(string& s, string t, int i, char sep)`

Asigna el *token* `t` en la posición `i` de la cadena `s` *tokenizada* por el carácter `sep`.

- `int findToken(string s, string t, char sep)`

Dada la cadena `s` *tokenizada* por el separador `sep`, retorna la posición que el *token* `t` ocupa dentro de la cadena o -1 si la misma no contiene a dicho *token*.

Colecciones de datos | TAD `Coll`

Estructura	Descripción
`struct Coll { string s; };`	Este TAD representa una colección de elementos de cualquier tipo de dato: `string`, `int`, `double`, `char` , incluso, tipos definidos por el programador. La implementación debe hacerse mediante *tokens*.

Funciones generales

- `Coll collCreate()` | Retorna una colección vacía.
- `int collSize(Coll coll)` | Retorna la cantidad de elementos de la colección.
- `void collRemoveAll(Coll& coll)` | Remueve todos los elementos de la colección dejándola vacía.
- `void collRemoveAt(Coll coll&,int p)` | Remueve el elemento ubicado en la posición `p`.

Funciones para valores enteros (`int`)

- `void collAddInt(Coll& coll,int i)` | Agrega el valor entero `i` al final de la colección.
- `void collSetIntAt(Coll& coll,int i,int p)` | Reemplaza al elemento de la posición `p` por `i`.
- `int collGetIntAt(Coll coll, int p)` | Retorna el valor entero ubicado en la posición `p`.
- `int collFindInt(Coll coll, int i)` | Retorna la posición de la primer ocurrencia de `i` o -1.

Funciones para valores flotantes (`double`)

- `void collAddDouble(Coll& coll,double d)`
- `void collSetDoubleAt(Coll& coll,double d, int p)`
- `double collGetDoubleAt(Coll coll, int p)`
- `int collFindDouble (Coll coll, double d)`

Funciones para valores de cadena (`string`)

- `void collAddString(Coll& coll,string s)`
- `void collSetStringAt(Coll& coll,string s,int p)`
- `string collGetStringAt(Coll coll,int p)`
- `int collFindString(Coll coll,string s)`

Funciones para valores carácter (`char`)

- `void collAddChar(Coll& coll,char c)`
- `void collSetCharAt(Coll& coll,char c,int p)`
- `char collGetCharAt(Coll coll,int p)`
- `int collFindChar(Coll coll,char c)`

Funciones para valores genéricos (`T`)

- ```
  template <typename T>
  void collAdd<T>(Coll& coll,T t,string toString(T))
  ```

  Agrega un valor de tipo `T` al final de la colección. La función `toString` que se recibe cómo parámetro recibe, a su vez, un parámetro de tipo `T` y debe retornar una cadena de caracteres que permita representarlo.

- ```
  template <typename T>
  void collSetAt<T>(Coll& coll,T t, int p, string toString(T))
  ```

  Reemplaza por `t` al elemento que se ubica en la posición `p` de la colección. La función `toString` debe retornar una cadena que represente al valor de tipo `T` que recibe cómo parámetro.

- ```
  template <typename T>
  T collGetAt(Coll coll, int p, T toObject(string))
  ```

  Retorna el elemento de tipo `T` que se ubica en la posición `p` de la colección. La función `toObject` que se recibe cómo parámetro debe retornar un valor de tipo `T` a partir de la cadena `s` que recibe.

- ```
  template <typename T, typename K>
  int collFind(Coll coll, K k, int compare(T,K), T toObject(string) )
  ```

Retorna la posición de la primer ocurrencia de `k`, de tipo `K`, dentro de alguno de los elementos de la colección de valores de tipo `T`. La función `toObject`, que recibe cómo parámetro, ya fue explicada más arriba. La función `compare` debe comparar dos valores de tipo `T` y `K` respectivamente; y retornar un valor entero que será: menor, igual o mayor que 0 (cero) según se considere que `t` sea: menor, igual o mayor que `k`.

Ejemplos

Colección de enteros

```
int main()
{
   Coll c = collCreate();
   collAddInt(c,1);
   collAddInt(c,2);
   collAddInt(c,3);
   for(int i=0; i<collSize(c); i++)
   {
      int v = collGetIntAt(c,i);
      cout << v << endl;
   }
   return 0;
}
```

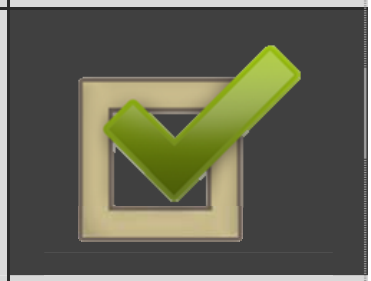
Colección de cadenas

```
int main()
{
   Coll c = collCreate();
   collAddString(c,"uno");
   collAddString(c,"dos");
   collAddString(c,"tres");
   for(int i=0; i<collSize(c); i++)
   {
      string v = collGetStringAt(c,i);
      cout << v << endl;
   }
   return 0;
}
```

Colección de objetos (tipos genéricos)

```
struct Persona
{
   int dni;
   string nombre;
}
Persona personaCreate(int dni, string nom)
{
   Persona p;
   p.dni = dni;
   p.nombre = nom;
   return p;
}
string personaToString(Persona p)
{
   return intToString(p.dni)+","+p.nombre;
}
Persona stringToPersona(string s)
{
   Persona p;
   p.dni = stringToInt(getTokenAt(s,0,','));
   p.nombre = getTokenAt(s,1,','));
   return p;
}

int personaCompareDNI(Persona p, int dni)
{
   return p.dni-dni;
}
int main()
{
   Coll c = collCreate();
   collAdd<Persona>(c,personaCreate(10,"Pedro"),personaToString);
   collAdd<Persona>(c,personaCreate(20,"Pablo"),personaToString);
   collAdd<Persona>(c,personaCreate(30,"Juan"),personaToString);
   for(int i=0; i<collSize(c); i++)
   {
      Persona v = collGetAt<Persona>(c,i,stringToPersona);
      cout << v.dni << "," << v.nombre << endl;
   }

   int dni;
   cout << "Ingrese un dni: ";
   cin >> dni;

   // busco una persona x DNI... la encuentro en la posicion: pos
   int pos = collFind<Persona,int>(c,dni,personaCompareDNI,stringToPersona);

   // obtengo la persona ubicada en la posicion pos de la coleccion
   Persona pers = collGetAt<Persona>(c,pos,stringToPersona);

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

ALGORITMOS Y ESTRUCTURA DE DATOS



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.

Autor: Ing. Pablo Augusto Sznajdleder.

Revisores: Ing. Analía Mora, Martín Montenegro.

Introducción

Las funciones que analizaremos en este documento son:

- `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","w+b");

   char c = 'A';
   fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'A' contenido en c

   c = 'B';
   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

   // cierro el archivo
   fclose(arch);

   return 0;
}
```

La función `fwrite` recibe los siguientes parámetros:

- Un puntero al *buffer* (variable) que contiene el datos que se van a grabar en el archivo.
- El tamaño (en bytes) del tipo de dato de dicho *buffer*; lo obtenemos con la función: `sizeof`.
- La cantidad de unidades del tamaño indicado más arriba que queremos escribir; en nuestro caso: 1.
- Finalmente, el archivo en donde grabará el contenido almacenado en el *buffer*.

La función `fopen` recibe los siguientes parámetros:

- Una cadena indicando el nombre físico del archivo que se quiere abrir.
- Una cadena indicando la modalidad de apertura; `"w+b"` indica que queremos crear el archivo si es que aún no existe o bien, si existe, que queremos dejarlo vacío.

La función `fclose` cierra el archivo que recibe cómo parámetro.

Leer un archivo de caracteres

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

using namespace std;

int main()
{
   // abro el archivo para lectura
   FILE* arch = fopen("DEMO.DAT","r+b");
```

```
   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;
}
```

La función `fread` recibe exactamente los mismos parámetros que `fwrite`. Respecto de la función `fopen`, en este caso la modalidad de apertura que utilizamos es: `"r+b"` para indicarle que el archivo ya existe y que no queremos vaciar su contenido; solo queremos leerlo.

Respecto de la función `feof` retorna `true` o `false` según si se llegó al final del archivo o no.

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","r+b");
   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 al inicio
   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 comparable a `fileSize` de Pascal 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);

      // leo el registro apuntado por el indicador de posicion
      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
   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: binarySearch

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 binarySearch(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","r+b");
   // 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);
```

ALGORITMOS Y ESTRUCTURA DE DATOS

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).

Autor: Ing. Pablo Augusto Sznajdleder.

Revisores: Ing. Analía Mora, Martín Montenegro.

Agregar un elemento al final de un array

La siguiente función agrega el valor `v` al final del *array* `arr`, incrementa su longitud `len` y retorna su posición.

```
int add(int arr[], int& len, int v)
{
   arr[len]=v;
   len++;
   return len-1; // retorna la posicion del elemento que agrego
}
```

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 find(int arr[], int len, int v)
{
   int i=0;
   while( i<len && arr[i]!=v )
   {
      i++;
   }
   return i<len?i:-1; // retorna la posicion de v o -1 si v no existe en arr
}
```

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 remove(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 insert(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 orderedInsert(int arr[], int& len, int v)
{
   int i=0;
```

```
   // mientras no me pase de largo y mientras no encuentre lo que busco...
   while( i<len && arr[i]<=v )
   {
      i++;
   }
   // inserto el elemento en la i-esima posicion del array
   insert(arr,len,v,i); // invoco a la funcion insert
   return i; // retorna la posicion en que se inserto al elemento
}
```

Más adelante veremos como independizar el criterio de precedencia para lograr que la misma función sea capáz 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 searchAndInsert(int arr[], int& len, int v, bool& enc)
{
   // busco el valor
   int pos = find(arr,len,v);
   // determino si lo encontre o no
   enc = pos>=0;
   // si no lo encontre entonces lo inserto ordenado
   if( !enc )
   {
      pos = orderedInsert(arr,len,v);
   }
   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 add y mostrar

```
template <typename T> int add(T arr[], int& len, T v)
{
   arr[len]=v;
   len++;
   return len-1;
}
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()
{
   // declaro un array de cadenas y su correspondiente longitud
   string aStr[10];
   int lens=0;

   // trabajo con el array de cadenas
   add<string>(aStr,lens,"uno");
   add<string>(aStr,lens,"dos");
   add<string>(aStr,lens,"tres");

   // muestro el contenido del array
   mostrar<string>(aStr,lens);

   // declaro un array de enteros y su correspondiente longitud
   int aInt[10];
   int leni =0;

   // trabajo con el array de enteros
   add<int>(aInt,leni,1);
   add<int>(aInt,leni,2);
   add<int>(aInt,leni,3);

   // muestro el contenido del array
   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 sort(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 como 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 `sort` aplique al momento de comparar cada par de elementos del *array* `arr`.

Observemos con atención el tercer parámetro que recibe la función `sort`. 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 sort(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 ver 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;
         }
      }
   }
   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 = 6;

   // un array con 6 cadenas
   string x[] = {"Pablo", "Pedro", "Andres", "Juan", "Zamuel", "Oronio"};

   // ordeno ascendentemente pasando como parametro la funcion criterioAZ
   sort<string>(x,len,criterioAZ);
   mostrar<string>(x,len);
```

```
   // ordeno descendentemente pasando como parametro la funcion criterioZA
   sort<string>(x,len,criterioZA);
   mostrar<string>(x,len);

   // un array con 6 enteros
   int y[] = {4, 1, 7, 2, 8, 3};

   // ordeno ascendentemente pasando como parametro la funcion criterio09
   sort<int>(y,len,criterio09);
   mostrar<int>(y,len);

   // ordeno ascendentemente pasando como parametro la funcion criterio90
   sort<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;
}
```

Usando arrays de estructuras

```
int main()
{
   Alumno arr[6];
   int len=0;
   add<Alumno>(arr,len,crearAlumno(30,"Juan",5));
   add<Alumno>(arr,len,crearAlumno(10,"Pedro",8));
   add<Alumno>(arr,len,crearAlumno(20,"Carlos",7));
   add<Alumno>(arr,len,crearAlumno(60,"Pedro",10));
   add<Alumno>(arr,len,crearAlumno(40,"Alberto",2));
   add<Alumno>(arr,len,crearAlumno(50,"Carlos",4));
   for(int i=0; i<len; i++)
   {
      cout<<arr[i].legajo<<", "<<arr[i].nombre<<", "<<arr[i].nota<<endl;
   }
   return 0;
}
```

Pregunta: ¿Por qué no utilizamos la función `mostrar` para mostrar el contenido del *array* de alumnos?

Ordenar arrays de estructuras, por diferentes criterios

Recordemos la función `sort`:

```
template <typename T> void sort(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 `sort`.

```
int main()
{
   Alumno arr[6];
   int len=0;

   add<Alumno>(arr,len,crearAlumno(30,"Juan",5));
   add<Alumno>(arr,len,crearAlumno(10,"Pedro",8));
   add<Alumno>(arr,len,crearAlumno(20,"Carlos",7));
   add<Alumno>(arr,len,crearAlumno(60,"Pedro",10));
   add<Alumno>(arr,len,crearAlumno(40,"Alberto",2));
   add<Alumno>(arr,len,crearAlumno(50,"Carlos",4));

   // ordeno por legajo
   sort<Alumno>(arr,len,criterioAlumnoLegajo);
   // recorrer y mostrar el contenido del array...

   // ordeno por nombre
   sort<Alumno>(arr,len,criterioAlumnoNombre);
   // recorrer y mostrar el contenido del array...

   // ordeno por nombre+legajo
   sort<Alumno>(arr,len,criterioAlumnoNomYLeg);
   // recorrer y mostrar el contenido del array...

   return 0;
}
```

Resumen de plantillas

Función `add`.

Descripción: Agrega el valor `v` al final del *array* `arr`, incrementa su longitud y retorna la posición

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

Función `find`.

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 find(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 `remove`.

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

```
template <typename T>
void remove(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 `insert`.

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

```
template <typename T>
void insert(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 `orderedInsert`.

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

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

   insert<T>(arr,len,v,i);

   return i;
}
```

Función `searchAndInsert`.

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 searchAndInsert(T arr[], int& len, T v, bool& enc, int criterio(T,T))
{
   // busco el valor
   int pos = find<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 = orderedInsert<T>(arr,len,v,criterio);
   }

   return pos;
}
```

Función `sort`.

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

```
template <typename T>
void sort(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;
}
```

Búsqueda binaria

Función `binarySearch`.

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 binarySearch(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;
}
```

ALGORITMOS Y ESTRUCTURA DE DATOS

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.

Autor: Ing. Pablo Augusto Sznajdleder.

Revisores: Ing. Analía Mora, Martín Montenegro.

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 Node
{
   int info;  // valor que contiene el nodo
   Node* 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:

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

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_n tales que: para todo $i>=1$ e $i<n$ se verifica que: n_i contiene la dirección de: n_{i+1} y n_n contiene la dirección nula `NULL`, entonces: el conjunto de nodos n_i constituye una lista enlazada. 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 `add` agrega un nuevo nodo con el valor `x` al final de la lista referenciada por `p`.

```
Node* add(Node*& p, int x)
{
   Node* nuevo = new Node();
   nuevo->info = x;
   nuevo->sig = NULL;
   if( p==NULL )
   {
      p = nuevo;
   }
   else
   {
      Node* aux = p;
      while(aux->sig!=NULL )
      {
         aux = aux->sig;
      }
      aux->sig = nuevo;
   }
   return 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(Node* p)
{
   Node* 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 `free` recorre la lista `p` liberando la memoria que ocupan cada uno de sus nodos.

```
void free(Node*& p)
{
   Node* sig;
   while( p!=NULL )
   {
      sig = p->sig;
      delete p;
      p = sig;
   }
}
```

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
   Node* p = NULL;
   // agrego valores
   add(p,1);
   add(p,2);
   add(p,3);
   add(p,4);
   mostrar(p);
   // libero la memoria utilizada
   free(p);
   return 0;
}
```

Determinar si una lista enlazada contiene o no un valor especificado

La función `find` 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.

```
Node* find(Node* p, int v)
{
   Node* 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 `remove` permite eliminar de la lista `p` al nodo que contiene el valor `v`.

```
void remove(Node*& p, int v)
{
   Node* aux = p;
   Node* 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 `removeFirst` elimina el primer nodo de la lista y retorna el valor que este contenía.

```
int removeFirst(Node*& p)
{
   int ret = p->info;
   Node* aux = p->sig;

   delete p;
   p = aux;

   return ret;
}
```

Insertar un nodo al principio de la lista

La función `insertFirst` crea un nuevo nodo cuyo valor será `v` y lo asigna cómo primer ele-mento de la lista direccionada por `p`. Luego de invocar a esta función `p` tendrá la dirección del nuevo nodo.

```
void insertFirst(Node*& p, int v)
{
   Node* nuevo = new Node();
   nuevo->info = v;
   nuevo->sig = p;
   p = nuevo;
}
```

Insertar un nodo manteniendo el orden de la lista

La función `orderedInsert` 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.

```
Node* orderedInsert(Node*& p, int v)
{
   Node* nuevo = new Node();
   nuevo->info = v;
   nuevo->sig = NULL;

   Node* ant = NULL;
   Node* 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 `sort` 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 sort(Node*& p)
{
   Node* q = NULL;
   while( p!=NULL ){
      int v = removeFirst(p);
      orderedInsert(q,v);
   }
   p = q;
}
```

Insertar en una lista enlazada un valor sin repetición

Busca el valor `v` en la lista `p`. Si no lo encuentra 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.

```
Node* findAndInsert(Node*& p, int v, bool& enc)
{
   Node* x = find(p,v);
   enc = x!=NULL;

   if( !enc )
   {
      x = orderedInsert(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.

Node

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

Función: add

```
template <typename T> Node<T>* add(Node<T>*& p, T x)
{
   // creo un nodo nuevo
   Node<T>* nuevo = new Node<T>();
   nuevo->info = x;
   nuevo->sig = NULL;

   if( p==NULL )
   {
      p = nuevo;
   }
   else
   {
      Node<T>* aux = p;
      while( aux->sig!=NULL )
      {
         aux = aux->sig;
      }

      aux->sig = nuevo;
   }

   return nuevo;
}
```

Función: free

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

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
   Node<int>* p1 = NULL;
   add<int>(p1,1);
   add<int>(p1,2);
   add<int>(p1,3);
   // la recorro mostrando su contenido
   Node<int>* aux1 = p1;
   while( aux1!=NULL )
   {
      cout << aux1->info << endl;
      aux1 = aux1->sig;
   }
   // libero la memoria
   free<int>(p1);
   // creo la lista de cadenas
   Node<string>* p2 = NULL;
   add<string>(p2,"uno");
   add<string>(p2,"dos");
   add<string>(p2,"tres");
   // la recorro mostrando su contenido
   Node<string>* aux2 = p2;
   while( aux2!=NULL )
   {
      cout << aux2->info << endl;
      aux2 = aux2->sig;
   }
   // libero la memoria
   free<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()
{
   Node<Alumno>* p = NULL;
   add<Alumno>(p,crearAlumno(10,"Juan"));
   add<Alumno>(p,crearAlumno(20,"Pedro"));
   add<Alumno>(p,crearAlumno(30,"Pablo"));
```

```
   Node<Alumno>* aux = p;
   while( aux!=NULL )
   {
      cout << aux->info.leg << ", " << aux->info.nom << endl;
      aux = aux->sig;
   }

   free<Alumno>(p);

   return 0;
}
```

Función: `find`

```
template <typename T, typename K>
Node<T>* find(Node<T>* p, K v, int criterio(T,K) )
{
   Node<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()
{
   Node<Alumno>* p = NULL;
   add<Alumno>(p,crearAlumno(10,"Juan"));
   add<Alumno>(p,crearAlumno(20,"Pedro"));
   add<Alumno>(p,crearAlumno(30,"Pablo"));

   int leg;
   cout << "Ingrese el legajo de un alumno: ";
   cin >> leg;
   // busco por legajo
   Node<Alumno>* r = find<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 = find<Alumno,string>(p,nom,criterioAlumnoNom);
   if( r!=NULL )
   {
      cout << r->info.leg << ", " << r->info.nom << endl;
   }

   free<Alumno>(p);

   return 0;
}
```

Función: remove

```
template <typename T, typename K>
void remove(Node<T>*& p, K v, int criterio(T,K))
{
   Node<T>* aux = p;
   Node<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: removeFirst

```
template <typename T>
T removeFirst(Node<T>*& p)
{
   T ret = p->info;
   Node<T>* aux = p->sig;

   delete p;
   p = aux;

   return ret;
}
```

Función: insertFirst

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

Función: orderedInsert

```
template <typename T>
Node<T>* orderedInsert(Node<T>*& p, T v, int criterio(T,T) )
{
   Node<T>* nuevo = new Node<T>();
   nuevo->info = v;
   nuevo->sig = NULL;

   Node<T>* aux = p;
   Node<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: sort

```
template <typename T>
void sort(Node<T>*& p, int criterio(T,T))
{
   Node<T>* q = NULL;
   while( p!=NULL )
   {
      T v = removeFirst<T>(p);
      orderedInsert<T>(q,v,criterio);
   }

   p = q;
}
```

Función: searchAndInsert

```
template <typename T>
Node<T>* searchAndInsert(Node<T>*& p,T v,bool& enc,int criterio(T,T))
{
   Node<T>* x = find<T,T>(p,v,criterio);
   enc = x!=NULL;
   if( !enc )
   {
      x = orderedInsert<T>(p,v,criterio);
   }
   return x;
}
```

Pilas

Una pila es una estructura restrictiva que admite dos únicas operaciones: *apilar* y *desapilar* 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

```
template <typename T> void push(Node<T>*& p, T v)
{
   // se resuelve insertando un nodo al inicio de la lista
   insertFirst<T>(p,v);
}
```

Función: pop

```
template <typename T> T pop(Node<T>*& p)
{
   // se resuelve eliminando el primer nodo de la lista y retornando su valor
   return removeFirst(p);
}
```

Ejemplo de cómo usar una pila

```
int main()
{
   Node<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: *enqueue y dequeue*. 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*.

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: enqueue

```
template <typename T>
void enqueue(Node<T>*& p, Node<T>*& q, T v)
{
   add<T>(q,v);
   if( p==NULL )
   {
      p = q;
   }
   else
   {
      q = q->sig;
   }
}
```

Función: dequeue

```
template <typename T>
T dequeue(Node<T>*& p, Node<T>*& q)
{
   T v = removeFirst<T>(p);

   if( p==NULL )
   {
      q = NULL;
   }
   return v;
}
```

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

```
int main()
{
   Node<int>* p = NULL;
   Node<int>* q = NULL;
   enqueue<int>(p,q,1);
   enqueue<int>(p,q,2);
   enqueue<int>(p,q,3);
   cout << dequeue<int>(p,q) << endl;
   cout << dequeue<int>(p,q) << endl;
   enqueue<int>(p,q,4);
   enqueue<int>(p,q,5);
   while( p!=NULL )
   {
      cout << dequeue<int>(p,q) << endl;
   }
   return 0;
}
```