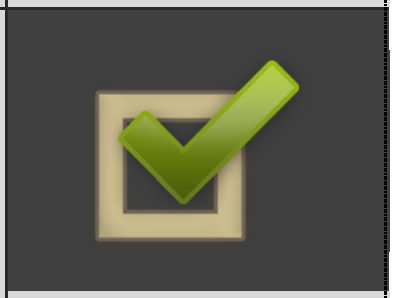
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

ALGORITMOS Y ESTRUCTURA DE DATOS
Machete Oficial 2014



Biblioteca de templates

Operaciones sobre arrays

Función: `agregar`

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

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

Función: `buscar`

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

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

Función: `eliminar`

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

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

Función: `insertar`

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

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

Función: `insertarOrdenado`

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

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

Función: **buscaEInsertaOrdenado**

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

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

Función: **ordenar**

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

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

Función: **busquedaBinaria**

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

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

Operaciones sobre estructuras dinámicas

El Nodo

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

Operaciones s/Listas

Función: **agregar**

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

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

Función: **liberar**

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

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

Función: **buscar**

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

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

Función: **eliminar**

Elimina el primer nodo de la lista direccionada por `p` cuyo valor coincida con `v`. El valor Buscado puede no existir. Retorna (ok? True: False)

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

Función: **eliminarDoble**

Elimina el primer nodo de la lista doblemente enlazada direccionada por `p` en el inicio `y q` en el final cuyo valor coincida con `v`. El valor Buscado puede no existir. Retorna (ok? True: False)

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

Función: **eliminarPrimerNodo**

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

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

Función: **insertarOrdenado**

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

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

Función: **insertarDoble**

Idem para una lista doblemente enlazada direccionada por p y q.

```
template <typename T>
Nodo<T>* insertarDoble(Nodo<T>*& p, Nodo<T>*& q, Tv, int (*criterio)(T,T));
```

Función: **ordenar**

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

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

Función: **buscaEInsertaOrdenado**

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

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

Operaciones sobre pilas

Función: **push** (poner)

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

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

Función: **pop** (sacar)

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

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

Opeaciones sobre colas (implementación: lista enlazada con dos punteros)

Función **encolar**

Encola el valor `v` en la cola direccionada por `p` y `q` implementada sobre enlazada.

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

Función **desencolar**

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

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

Operaciones sobre archivos

Función **read**

Lee un registro de tipo `T` desde el archivo `f`. Para determinar si llegó o no el *eof* se debe utilizar la función `feof`.

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

Función **write**

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

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

Función **seek**

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

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

Función **fileSize**

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

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

Función **filePos**

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

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

Función **busquedaBinaria**

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

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

```
int busquedaBinaria(FILE* f, K v, int (*criterio)(T,K));
```

CONCEPTOS DE TEMPLATES Y EJEMPLOS DE USO

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

Generalización de tipo de dato.

La función para un tipo de dato simple presenta la misma lógica algorítmica, si se modifica el tipo de dato es solo eso lo que debería modificarse. Las plantillas o template nos permiten resolver esa situación a través de la definición de un tipo de dato genérico que toma el tipo del dato con el que se hace la invocación, en este caso el tipo de dato genérico es representado por T

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

Ejemplo de invocacion

```
int main()
{
  string aStr[10]; // vector de cadenas
  int lens =10;
  ..................
  mostrar<string>(aStr,lens); // T es string

  int aInt[10]; // vector enteros
  int leni =10;
  ..................
  mostrar<int>(aInt,leni); // T es int
  return 0;
}
```

PUNTEROS A FUNCIONES

Las funciones pueden ser pasadas cómo parámetros a otras funciones para que éstas las invoquen. Utilizaremos esta característica de los lenguajes de programación para parametrizar el criterio de precedencia de los ordenamientos.

La función `criterio`, que debemos desarrollar por separado, debe comparar dos elementos e1 y e2, ambos de tipo `T`, y retornar un valor: negativo, positivo o cero según se sea: e1<e2, e1>e2 o e1=e2 respectivamente.

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

Ordenar con diferentes tipos de datos y 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;
}
```

Ejemplo

```
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

Declaración de la estructura:

```
struct Alumno
{
  int legajo;
  string nombre;
  int nota;
};
```

Mostrar arrays de estructuras

Debe recibir una función que será la encargada de mostrar cada registro 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];
  ……
  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

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;
}
```

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

```
int main()
{
  Alumno arr[6];
  …….
  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);
```