# Tokenizador

## Introducción

En el procesamiento de texto, la segmentación de strings en palabras es una tarea fundamental y esencial para una amplia gama de aplicaciones. El tokenizador es un componente clave en el procesamiento del lenguaje natural y en el análisis de datos de texto. Sin embargo, la creación de un tokenizador eficiente y preciso puede ser un desafío. Por lo tanto, en este trabajo se propone la implementación de una clase Tokenizador en C++ para segmentar strings en palabras, con la opción de detectar una serie de casos especiales de palabras.

La hipótesis de partida es que, al implementar un algoritmo basado en reglas específicas, se puede lograr un tokenizador eficiente y preciso. Además, al agregar la capacidad de detectar casos especiales de palabras, como palabras compuestas y números, el tokenizador será capaz de manejar una amplia variedad de textos y aumentar su precisión en la segmentación de strings. Este trabajo se centrará en el diseño y la implementación de la clase Tokenizador, y se evaluará su rendimiento mediante pruebas y comparaciones con otros tokenizadores.

Partimos de la siguiente versión de un tokenizador, vista en clase:

```
void
Tokenizador::Tokenizar (const string& str, list<string>& tokens) {
   string::size_type lastPos = str.find_first_not_of(delimiters,0);
   string::size_type pos = str.find_first_of(delimiters,lastPos);

   while(string::npos != pos || string::npos != lastPos)
   {
       tokens.push_back(str.substr(lastPos, pos - lastPos));
       lastPos = str.find_first_not_of(delimiters, pos);
       pos = str.find_first_of(delimiters, lastPos);
   }
}
```

Esta función recorre una cadena detectando secuencias de caracteres delimitadas por caracteres que se hayan definido como delimitadores, las extrae y las añade a la lista de tokens.

Además, en el enunciado de la práctica se proponen varias funciones para tokenizar ficheros y realizar accesos a directorios.

En cuanto a los casos especiales se detectarán URL, números decimales, e-mail, acrónimos y palabras compuestas (separadas por guiones). Además se van a poder quitar acentos y mayúsculas.

## Solución implementada

### Estructuras de datos

### Algoritmo

## Justificación de la solución

Como primera mejora de la versión inicial del tokenizador se han implementado contenedores asociativos para guardar los caracteres delimitadores. Para esto se ha usado el *unordered_set*.

```
delimiters = delimitadoresPalabra;
delimitersSet = unordered_set<char>(delimiters.begin(), delimiters.end());
```

Como se ha visto en teoría, la complejidad temporal del tokenizador inicial es de $O(n * d)$ siendo *n* la cantidad de caracteres en el string a tokenizar y *d* la cantidad de caracteres en el string de delimitadores. Al usar un contenedor asociativo cuya búsqueda tiene una complejidad promedio constante, se puede sustituir el *find_first_of()* y el *find_first_not_of()* por una versión más eficiente.

```
size_t Tokenizador::efficient_find_first_of(const string &str, const size_t &pos) const {
    for (size_t i = pos; i < str.size(); i++) {
        if (this->delimitersSet.count(str[i])>0) {
            return i;
        }
    }
    return string::npos;
}

size_t Tokenizador::efficient_find_first_not_of(const string &str, const size_t &pos) const {
    for (size_t i = pos; i < str.size(); i++) {
        if (this->delimitersSet.count(str[i])==0) {
            return i;
        }
    }
    return string::npos;
}
```

Con esta mejora la complejidad temporal promedio sería de $O(n)$.

Por otro lado, se ha añadido una complejidad espacial adicional. Además, se podría mejorar todavía más la complejidad temporal ya que el cálculo de los valores hash de los *unordered_set* consumen la mayor parte del tiempo de computación:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 11.78      0.24     0.24                             std::__detail::_Mod_range_hashing::operator()(unsigned long, unsigned long) const
  8.65      0.42     0.18                             Tokenizador::pasar_a_minusculas_sin_acentos(std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
  5.29      0.54     0.11                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
  4.33      0.62     0.09                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
  3.85      0.70     0.08                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
  3.37      0.78     0.07                             Tokenizador::TokenizarEspecialesEstados2(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
  2.88      0.83     0.06                             std::__detail::_Node_iterator_base<char, false>::_Node_iterator_base(std::__detail::_Hash_node<char, false>*)
  2.64      0.89     0.06                             char const& std::__detail::_Identity::operator()<char const&>(char const&) const
  2.40      0.94     0.05                             std::__detail::_Hash_code_base<char, char, std::__detail::_Identity, std::hash<char>, std::__detail::_Mod_range_
  2.40      0.99     0.05                             _init
  2.16      1.03     0.04                             std::__detail::_Hash_code_base<char, char, std::__detail::_Identity, std::hash<char>, std::__detail::_Mod_range_
  2.16      1.08     0.04                             std::__detail::_Hash_node_value_base<char>::_M_v() const
  1.92      1.12     0.04                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
```

## Análisis de las mejoras

## Análisis de la complejidad

### Teórica

### Práctica