EXPLOTACIÓN DE LA INFORMACIÓN

Memoria Tokenizador

Rafael Bustos Moreno 49627353-T

INTRODUCCIÓN

En esta práctica se realizará una implementación de un 'tokenizador'. Dicha implementación la utilizaremos más adelante para la práctica de crear un 'indexador'.

Los principales problemas que se presentan en esta práctica son dos: capturar casos especiales en la query y pasar la query a minúsculas y quitarle los acentos mediante el uso de la codificación ISO8859-1.

Para resolver el problema de los casos especiales, la hipótesis de partida era iterar cada caracter de la query hasta encontrar un delimitadores, una vez encontrado este, comprobar las heurísticas de cada caso especial y verificar de que caso especial se trata. Una vez verificado, guardar ese token como caso especial.

Por otra parte, para el problema de pasar a minúsculas y quitar acentos he planteado una solución iterativa en la que itero cada caracter de la query y quitando los acentos y pasando a minúsculas cada caracter.

ANÁLISIS DE LA SOLUCIÓN ELEGIDA

Empezaré exponiendo parte de mi código implementado. Primero pondré el fragmento del método Tokenizar al que se llama cuando se quiere tokenizar una query:

```
// Funcion tokenizar que devuelve una lista de tokens
void Tokenizador::Tokenizar(string str, list<string>& tokens) const
{
    tokens.clear();

    // Hacemos una copia de la variable privada delimiters y le metemos los delimitadors espacio y salto de línea
    string delimiters = eliminaDuplicados(this->delimiters + " \n");

    // Comprobamos si la variable booleana para pasar a minuscula y sin acentos esta activa o no
    if(pasarAminuscSinAcentos) { str = convertirSinMayusSinAcen(str); }

    // Comprueba si queremos tokenizar con los casos especiales o no
    casosEspeciales ? TokenizarCasosEspeciales(str, tokens, delimiters) : TokenizarSinCasosEspeciales(str, tokens);
}
```

Como se puede observar este método su función más importante es llamar a otros métodos como quitar acentos o tokenizar con casos especiales. Dicho esto, pasaré a exponer el fragmento en donde paso a minúsculas y quito acentos.

En este método que se llama desde tokenizar lo que se hace es recorrer la query caracter a carácter, llamando al método normalizarCaracter pasándole por parámetro cada caracter de la query.

```cpp
// Funcion auxiliar que pasa un string pasado a otro sin minusculas y sin acentos
string Tokenizador::convertirSinMayusSinAcen(string str) const
{
    string minusculas;
    string::iterator it;

    for(it = str.begin(); it != str.end(); it++)
    {
        switch (*it)
        {
            case '\300'...'\305': case '\340'...'\345':
                minusculas += 'a';
                break;
            case '\310'...'\313': case '\350'...'\353':
                minusculas += 'e';
                break;
            case '\314'...'\317': case '\354'...'\357':
                minusculas += 'i';
                break;
            case '\321':
                minusculas += '\361';
            case '\322'...'\326': case '\362'...'\366':
                minusculas += 'o';
                break;
            case '\331'...'\334': case '\371'...'\374':
                minusculas += 'u';
                break;
            default:
                minusculas += tolower(*it);
                break;
        }
    }

    return minusculas;
}
```

En este método lo que hago es recorrer cada caracter de la query. En cada caracter compruebo mediante la codificación ISO8859-1 si pertenece a un caracter con tilde o no y almaceno el caracter al que pertenece sin tilde y minúscula. En caso de que no pertenezca a un caracter con tilde, paso a minúscula el caracter y lo almaceno.

Habiendo visto este método, pasaré a mostrar y explicar el método TokenizarCasosEspeciales:

```cpp
// Funcion tokenizar con casos especiales
void Tokenizador::TokenizarCasosEspeciales(const string &str, list<string> &tokens, const string &delimiters) const
{
    string::size_type lastPos = str.find_first_not_of(delimiters,0), pos = str.find_first_of(delimiters,lastPos);

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

En este método, como observamos al principio se extrae la posición del primer delimitador que se encuentra y seguidamente la posición del primer elemento que no sea delimitador que se encuentra. Este método sigue mucho la dinámica del tokenizador que se nos daba en el enunciado de la práctica.

Una vez se saca las posiciones anteriores, se procede a verificar si pertenece a alguno de los 5 casos especiales que existen. Para esto se irán llamando a los métodos que se ven dentro de los ifs. Empezaré enseñando el método casoUrl:

```
// Funcion para sacar url
bool Tokenizador::casoUrl(list<string> &tokens, const string &str, string::size_type &pos, string::size_type &lastPos, const string &delimitadoresUrl,
const string &delimiters) const
{
    if (str.find("http:", lastPos) == lastPos || str.find("https:", lastPos) == lastPos || str.find("ftp:", lastPos) == lastPos)
    {
        char siguienteAHtpp = str[str.find_first_of(":", lastPos) + 1];
        bool sigueCaracter = (delimitadoresUrl.find(siguienteAHtpp) && siguienteAHtpp != '\0');

        // Si después de los dos puntos no le sigue ningun caracter, no se considera URL
        if(!sigueCaracter) return false;

        pos = str.find_first_of(delimitadoresUrl, lastPos);
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        lastPos = str.find_first_not_of(delimiters, pos);
        pos = str.find_first_of(delimiters, lastPos);

        return true;
    }

    return false;
}
```

Como se observa, primero se verifica que cumple con las heurísticas explicadas en la práctica. Si alguna de ellas no se cumple el método devolverá false. Haciendo que no se reconozca como url y se compruebe si pertenece a alguno de los siguientes casos especiales.

El resto de métodos de los casos son todos iguales, a excepción de los acrónimos y multipalabra:

```
// Funcion para verificar si es acronimo o multipalabra o no y guardar el token correspondiente
bool Tokenizador::casoAcronimoYMulti(const char &car, list<string> &tokens, const string &str, string::size_type &pos,
string::size_type &lastPos, const string &delimitadoresAcronimOMulti, const string &delimiters) const
{
    string::size_type posAnterior = delimiters.find(str[pos-1]), posPosterior = delimiters.find(str[pos+1]),
    posAux= str.find_first_of(delimitadoresAcronimOMulti, lastPos);
    string tokenAcumulador;

    while((str[pos] == car) && (posPosterior == string::npos) && (posAnterior == string::npos) && (str[pos+1] != '\0') && (str[pos-1] != '\0'))
    {
        tokenAcumulador += str.substr(lastPos, pos-lastPos) + str[pos];

        lastPos = str.find_first_not_of(delimiters, pos);
        pos = str.find_first_of(delimiters, lastPos);

        posAnterior = delimiters.find(str[pos-1]);
        posPosterior = delimiters.find(str[pos+1]);

        if (pos == posAux || posPosterior != string::npos || posAnterior != string::npos || str[pos+1] == '\0')
        {
            tokenAcumulador += str.substr(lastPos, pos-lastPos);
            tokens.push_back(tokenAcumulador);
            lastPos = str.find_first_not_of(delimiters, pos);
            pos = str.find_first_of(delimiters, lastPos);

            return true;
        }
    }

    return false;
}
```

Como se observa, utilizo este método para realizar tanto los acrónimos como los multipalabras. Lo único que cambia es que paso por parámetro el caracter característico de cada caso especial. En el caso de multipalabra se enviará por parámetro el caracter ‘–’ y en el caso acrónimos el caracter ‘.’.

Descartando este métodos, los demás son bastante similares a estos métodos y me parecía excesivo meter todos los métodos de casos especiales en la memoria.

JUSTIFICACIÓN DE LA SOLUCIÓN ELEGIDA

En este apartado justificaré el porque realice la práctica de la manera que expongo en el apartado anterior. Realmente no he realizado muchos cambios de implementación a como lo tenía en mente, ya que fui desde un primer momento a realizarlo de la manera en la que está hecho.

El único caso que cambié fue el hecho de hacer quitar los delimitadores especiales de los delimitadores iniciales. Este método al principio lo llamaba dentro de los métodos que realizaban la comprobación de si era un caso especial o no, como se observa debajo:

```cpp
// Funcion para sacar url
bool Tokenizador::casoUrl(list<string> &tokens, const string &str, string::size_type &pos, string::size_type &lastPos, const string &delimiters) const
{
    string delimitadoresUrl = quitarEspeciales("_:/.?&-=#@", delimiters);

    if (str.find("http:", lastPos) == lastPos || str.find("https:", lastPos) == lastPos || str.find("ftp:", lastPos) == lastPos)
    {
        char siguienteAHtpp = str[str.find_first_of(":", lastPos) + 1];
        bool sigueCaracter = (delimitadoresUrl.find(siguienteAHtpp) && siguienteAHtpp != '\0');

        // Si despu?s de los dos puntos no le sigue ningun caracter, no se considera URL
        if(!sigueCaracter) return false;

        pos = str.find_first_of(delimitadoresUrl, lastPos);
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        lastPos = str.find_first_not_of(delimiters, pos);
        pos = str.find_first_of(delimiters, lastPos);

        return true;
    }

    return false;
}
```

```cpp
// Funcion para verificar si es acronimo o multipalabra o no y guardar el token correspondiente
bool Tokenizador::casoAcronimoYMulti(const string &car, list<string> &tokens, const string &str, string::size_type &pos,
string::size_type &lastPos, const string &delimiters) const
{
    string tokenAcumulador, delimitadoresAcronimOMulti = quitarEspeciales(car, delimiters);
    string::size_type posAnterior = delimiters.find(str[pos-1]), posPosterior = delimiters.find(str[pos+1]),
    posAux= str.find_first_of(delimitadoresAcronimOMulti, lastPos);

    while((str[pos] == car[0]) && (posPosterior == string::npos) && (posAnterior == string::npos) && (str[pos+1] != '\0') && (str[pos-1] != '\0'))
    {
        tokenAcumulador += str.substr(lastPos, pos-lastPos) + str[pos];

        lastPos = str.find_first_not_of(delimiters, pos);
        pos = str.find_first_of(delimiters, lastPos);

        posAnterior = delimiters.find(str[pos-1]);
        posPosterior = delimiters.find(str[pos+1]);

        if (pos == posAux || posPosterior != string::npos || posAnterior != string::npos || str[pos+1] == '\0')
        {
            tokenAcumulador += str.substr(lastPos, pos-lastPos);
            tokens.push_back(tokenAcumulador);
            lastPos = str.find_first_not_of(delimiters, pos);
            pos = str.find_first_of(delimiters, lastPos);

            return true;
        }
    }

    return false;
}
```

Lo que hice en vez de calcularlo dentro del método fue pasarle esta variable por parámetro. De esta manera el algoritmo no lo calcularía por cada token haciendo que el tiempo y el espacio en memoria decrementase.

Análisis de las mejoras realizadas en la práctica

Viendo en el anterior apartado que el único cambio realizado fue el hecho de calcular los delimitadores de cada caso especial fuera de los métodos de verificación de caso, expondré solo un análisis sobre esto.

En primer lugar, insertaré el fragmento de código del método quitarEspeciales:

```
// Funcion que devuelve los delimitadores habiendole quitados los delimitadores pasador por argumento
string Tokenizador::quitarEspeciales(const string &especiales, const string &delimiters) const
{
    string aux = delimiters;
    string::size_type pos = aux.find_first_of(especiales);

    while(pos != string::npos)
    {
        aux.erase(aux.begin() + pos);
        pos = aux.find_first_of(especiales);
    }

    return aux;
}
```

Como se observa, se recorre el string de delimiters con el método find_first_of. Entonces siendo n el tamaño de delimiters, la complejidad de este algoritmo será n*m, ya que se recorre m veces, siendo m el número de delimitadores especiales que existen en delimiters y en especiales. Si nos vamos al método tokenizarEspeciales vemos que tenemos un while que recorre la query 1 vez. Por tanto, siendo d el número de tokens de la query, en caso de que tengamos que llamar al método quitarEspeciales por cada token, tendría una complejidad de O(n*m*d).

Sin embargo, de la manera que he escogido al final, se llama a los métodos quitarEspeciales solo una vez al principio del método. Por lo que, siendo O(n) la complejidad del método quitarEspeciales y al llamarse una sola vez, su complejidad seguirá siendo O(n).

Análisis de complejidad del tokenizador

Para el análisis de complejidad nos fijaremos unicamente en el método tokenizarCasosEspeciales:

```
// Funcion tokenizar con casos especiales
void Tokenizador::TokenizarCasosEspeciales(const string &str, list<string> &tokens, const string &delimiters) const
{
    string::size_type lastPos = str.find_first_not_of(delimiters,0);
    string::size_type pos = str.find_first_of(delimiters,lastPos);
    string delimitadoresUrl = quitarEspeciales("_:/.?&-=#@", delimiters), delimitadoresDecimal = quitarEspeciales(".,%$", delimiters),
    delimitadoresEmail = quitarEspeciales(".-_", delimiters), delimitadoresAcronim = quitarEspeciales(".", delimiters),
    delimitadoresMulti = quitarEspeciales("-", delimiters);

    while(string::npos != pos || string::npos != lastPos)
    {
        if(!casoUrl(tokens, str, pos, lastPos, delimitadoresUrl, delimiters))
        {
            if(!casoDecimal(tokens, str, pos, lastPos, delimitadoresDecimal, delimiters))
            {
                if(!casoEmail(tokens, str, pos, lastPos, delimitadoresEmail, delimiters))
                {
                    if(!casoAcronimoYMulti('.', tokens, str, pos, lastPos, delimitadoresAcronim, delimiters))
                    {
                        if(!casoAcronimoYMulti('-', tokens, str, pos, lastPos, delimitadoresMulti, delimiters))
                        {
                            tokens.push_back(str.substr(lastPos, pos - lastPos));
                            lastPos = str.find_first_not_of(delimiters, pos);
                            pos = str.find_first_of(delimiters, lastPos);
                        }
                    }
                }
            }
        }
    }
}
```

Mejor Caso

Primero calcularé el mejor caso. Por tanto, la complejidad mejor para quitarEspeciales sería n, que supondría recorrer el string de delimiters 1 vez, es decir, no encontrar ningún delimitador común entre los especiales del caso y los ya introducidos. Por otro lado, el mejor caso para el while siguiente sería que todos los tokens fueran URL, ya que es el único caso en el que recorro la query una vez.

```
// Funcion para sacar url
bool Tokenizador::casoUrl(list<string> &tokens, const string &str, string::size_type &pos, string::size_type &lastPos, const string &delimitadoresUrl,
const string &delimiters) const
{
    if (str.find("http:", lastPos) == lastPos || str.find("https:", lastPos) == lastPos || str.find("ftp:", lastPos) == lastPos)
    {
        char siguienteAHtpp = str[str.find_first_of(":", lastPos) + 1];
        bool sigueCaracter = (delimitadoresUrl.find(siguienteAHtpp) && siguienteAHtpp != '\0');

        // Si después de los dos puntos no le sigue ningun caracter, no se considera URL
        if(!sigueCaracter) return false;

        pos = str.find_first_of(delimitadoresUrl, lastPos);
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        lastPos = str.find_first_not_of(delimiters, pos);
        pos = str.find_first_of(delimiters, lastPos);

        return true;
    }

    return false;
}
```

Como podemos calcular de aquí la complejidad en el mejor caso sería siendo m el tamaño de la query, la complejidad en el mejor caso sería $\Omega(m)$.

Teniendo estas dos cotas, tendríamos que la complejidad en el mejor caso sería $\Omega(n+m)$.

Peor Caso

En el peor de los casos sabemos que quitarEspeciales tiene como peor caso O(n*m), siendo n el tamaño de delimiters y m los caracteres en común entre delimiters y especiales.

En el método tokenizarCasosEspeciales, el peor de los casos sería que todos los los tokens fuesen números decimales. Realmente, recorro igual que en los URL la cadena una vez, por lo que la complejidad sería O(m) siendo m el tamaño de la query, pero a efectos prácticos es el método en el que más operaciones realizo.

Por tanto la complejidad en el peor de los casos sería O(n*d+m).