Universidad de Concepción
Facultad de Ingeniería
Dpto. de Ing. Informática y Cs. de la Computación

PROYECTO 1:

Texto Predictivo

Estructuras de Datos (2020-1)

Profesor: Diego Seco Naveiras

Carrera: Ingeniería Civil Informática

Ayudantes:

Catalina Pezo Vergara

Alexis Espinoza Rebolledo

Integrantes:

Lucas Kraemer Ananías

Cristián Pérez Vásquez

Fecha de Entrega: 08 de Junio de 2020

Descripción de la tarea

En el presente proyecto, se busca implementar un sistema de predicción de textos para el idioma español o inglés, en el lenguaje de programación C++. Este predictor debe incorporar una base de datos de palabras, que sería el diccionario, donde cada palabra tendrá asignada un número entero positivo, el cual representa la frecuencia con la que dicha palabra se ha utilizado. El predictor recibirá palabras para luego retornar las respectivas predicciones en un archivo de texto.

En la implementación, el predictor debe recibir el diccionario, las palabras a ser predichas para ser procesadas, el archivo donde se escribirán los resultados de la predicción, y un número entero *k*.

El predictor usará este número para buscar las *k* palabras que comiencen con cada palabra del archivo que se ingresó, que contiene cada una de las que van a ser predichas; éstas se llamarán prefijos. En caso de que no existan *k* palabras que comiencen con el prefijo respectivo, se debe repetir el proceso de búsqueda de predicción, pero quitando la última letra del prefijo. Este proceso se repetirá hasta completar las *k* predicciones. En caso de que haya empates en la frecuencia, el criterio de desempate será de orden alfabético.

Finalmente, las *k* palabras que resulten de la predicción serán escritas en el archivo de resultados, y cada una de éstas aumentará en 1 la frecuencia que le corresponde en la base de datos del diccionario.

Descripción de la solución propuesta

Para abordar la tarea, fue necesario crear clases que cumplieran distintos objetivos, para así mantener el orden y no mezclar las funciones de cada una de éstas innecesariamente. Estas clases son:

- **Controladora**
- **Predictor**
- **Trie**

Controladora

La clase **Controladora** se encarga de ingresar y procesar los archivos que recibe el sistema al momento de llamarlo por línea de comandos. Además contiene una instancia de la clase **Predictor**, por lo que será el puente directo entre el sistema de predicción y el "trabajo interno", es decir, los métodos de **Predictor** serán llamados por medio de **Controladora**.

Además, **Controladora** se encargará de ingresar el diccionario al sistema de predicción, de llamar al método *Predictor::busqueda()* para buscar cada palabra contenida en el archivo de los prefijos, y finalmente de imprimir los resultados de la predicción en el archivo *resultados* que también fue ingresado a **Controladora**.

Predictor

La clase **Predictor** es quien realiza todo el trabajo de recibir los prefijos y buscar sus similares en el **Trie**. Contiene una instancia de la clase **Trie**, donde podrá acceder a los métodos de esta última para conocer las palabras que en un principio fueron insertadas como diccionario.

Esta clase actúa como mediador entre **Controladora** y **Trie**, ya que es quien adecúa el formato de cada palabra del diccionario para ser ingresada en este último.

Por otro lado, **Predictor** también se encarga de priorizar las palabras dependiendo de la frecuencia que está almacenada en cada una de ellas en **Trie**, es decir, retornará aquellas palabras que tengan mayor frecuencia respecto a otras. Y finalmente, en caso de que exista empates en la frecuencia, ordena las palabras por orden alfabético, y retorna aquellas que estén primero.

Trie

La clase Trie es una estructura de datos que almacena el diccionario que recibió **Controladora**. Esta estructura de datos no conoce los métodos de **Predictor** ni **Controladora**, pero **Predictor** trabaja en conjunto con **Trie** para obtener las palabras mediante métodos *getters*, es decir, métodos que acceden al **Trie** para devolver palabras almacenadas en la base de datos, pero sin modificar las propias palabras. Todos los recorridos desde el nodo raíz hasta los nodos hojas corresponden a una palabra almacenada.

Para buscar una palabra en **Trie** se empieza en la raíz del árbol, buscando un subárbol hijo cuyo arco que los conecta contenga el primer caracter de la palabra. Si éste existe, se prosigue de forma recursiva, esta vez con la raíz del subárbol y el segundo caracter de la palabra (y así sucesivamente), hasta encontrar el primer caracter que no exista. Una vez ahí, se crean los nodos y arcos pertinentes hasta llegar al último caracter de la palabra, que en este caso es el signo '$', que indica el fin de la misma.

Gráficamente, la clase **Trie** se muestra de la forma:

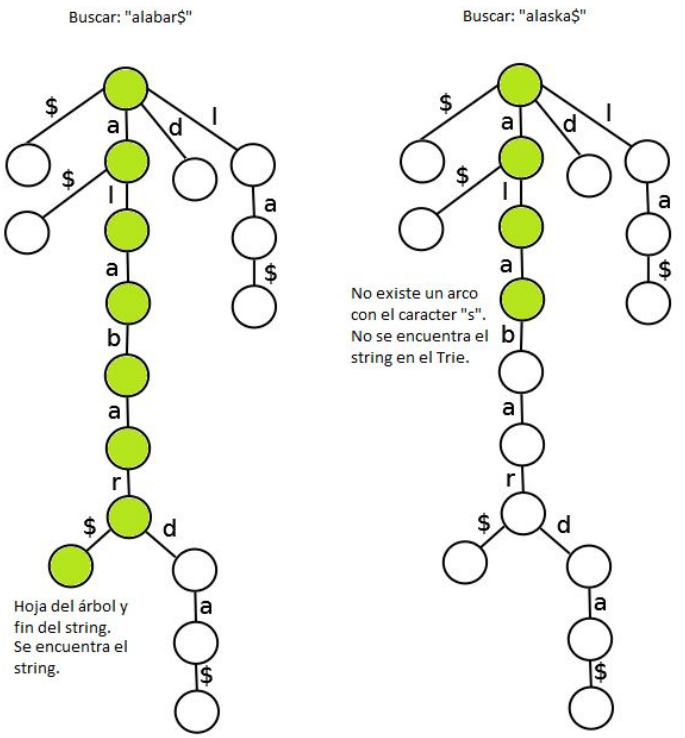
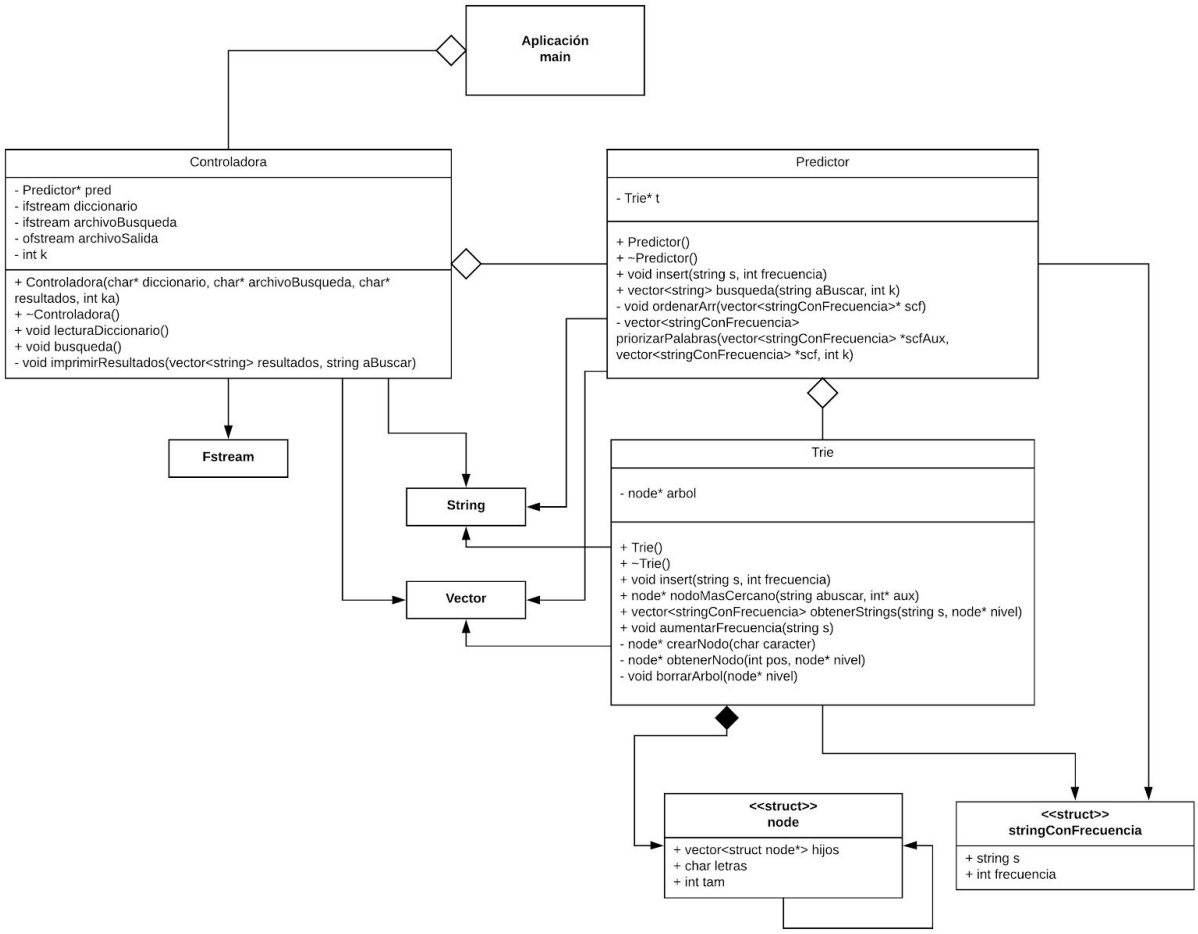


Diagrama de Clases UML

Todas las clases del proyecto se encuentran en el siguiente diagrama de clases, donde se muestra la conexión, las llamadas y las implementaciones entre cada una de ellas.



Los métodos principales de cada clase son:

- **Controladora**::Controladora(): Procesa los parámetros que recibe el programa, y quedan almacenados como variables privadas en la clase Controladora.
- **Controladora**::lecturaDiccionario(): Recibe cada palabra del diccionario, le asigna una frecuencia aleatoria entre 0 y 1000 y llama a Predictor::insert(), almacenando palabra por palabra en el Trie.

- **Controladora**::busqueda(): Recibe cada palabra del archivo de prefijos y llama a Predictor::busqueda(), que retornará la palabra más cercana al prefijo, considerando similitud, frecuencia y orden alfabético. Luego esta palabra retornada se ingresa como parámetro en Controladora::imprimirResultados().
- **Controladora**::imprimirResultados(): Cada palabra que recibe es escrita en el archivo de resultados
- **Predictor**::Predictor(): Crea una instancia de Trie y la almacena en una variable privada.
- **Predictor**::insert(): Le asigna el caracter '$' al final de la palabra recibida como parámetro y llama a Trie::insert()
- **Predictor**::busqueda(): Recibe el prefijo a buscar, llama a Trie::obtenerStrings() hasta obtener al menos las k cantidad de strings pedidas. Una vez obtenido esto, las ordena y selecciona aquellas que se imprimirán en el archivo de salida.
- **Trie**::Trie(): Crea una instancia de nodo denominada *arbol* y la almacena en una variable privada.
- **Trie**::insert(): Inserta las cadenas recibidas dentro del Trie. Para esto se acerca lo más posible a la palabra letra por letra y, si no se encuentra la letra de la palabra, crea nuevos nodos. Finalmente asigna la frecuencia de la palabra en el último nodo.
- **Trie**::obtenerStrings(): Busca en el Trie la palabra más cercana al prefijo entregado y llama a Trie::insertarStringsConFrecuencia() para obtener las strings más cercanas a este prefijo.

Detalles de Implementación

El Trie guarda los datos del diccionario en estructuras denominadas nodos. Estos nodos guardan tres variables: un caracter que almacena las letras de las palabras, un integer (entero) que almacena la cantidad de veces que se repite la palabra cuando se indique que se ha llegado al fin de la misma y, por último, un vector que es utilizado para guardar los siguientes nodos del Trie, con el fin de brindar las conexiones entre las letras de cada palabra.

En C++ la estructura se ve como:

```
struct node {
    std::vector<struct node*> hijos;
    char letras;
    int tam;
};
```

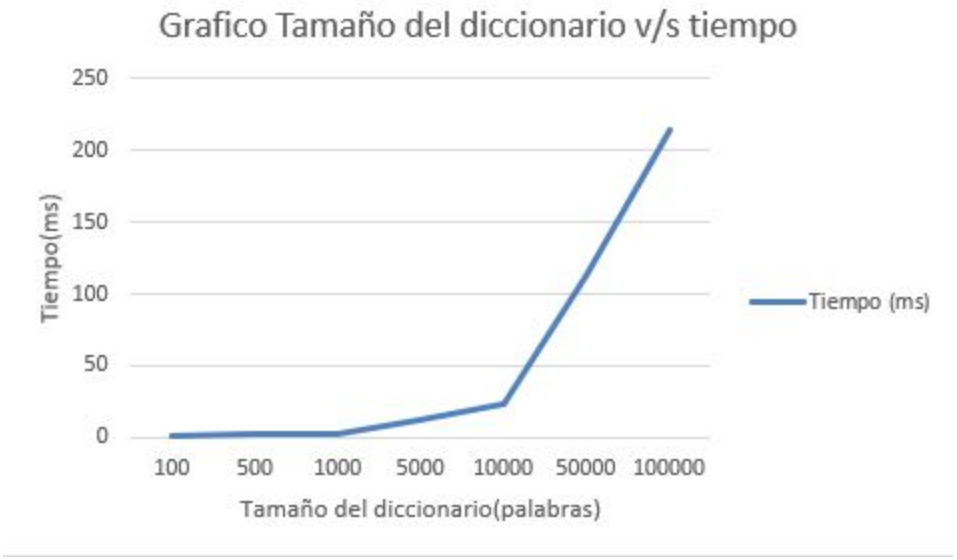
El pseudocódigo del algoritmo de búsqueda, utilizado por el Predictor es el siguiente:

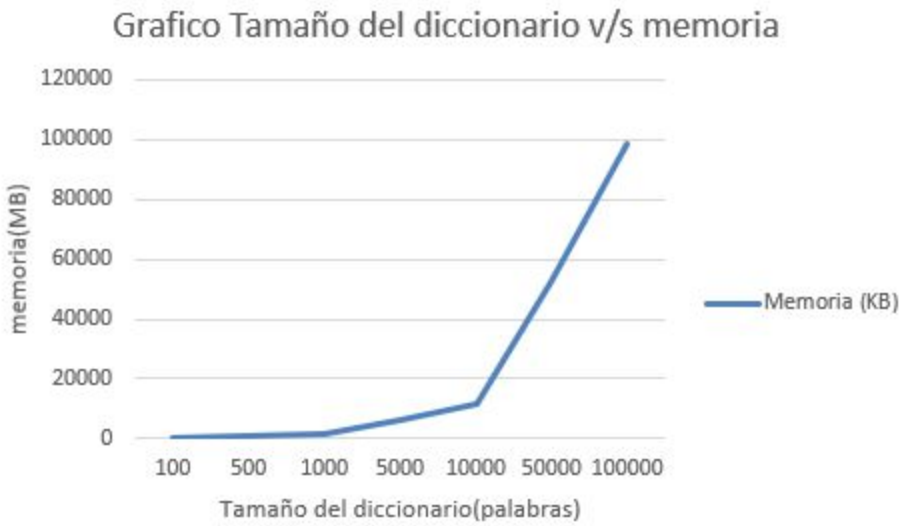
```
Algorithm:   busqueda(aBuscar, k):
Input:    A string aBuscar to search in the Trie and an integer k that contains the number
          of words that should be printed in the output file
Output:   A string vector resultados that contains the strings that are going to be printed
          in the output file.

    aux <- aBuscar.size()+1
    resultado <- an empty vector that contains stringConFrecuencia
    scf <- an empty vector that contains stringConFrecuencia
    scfAux <- an empty vector that contains stringConFrecuencia;
    resultados <- an empty vector that contains strings;
    do
        scfAux <- scf
        aux <- aux-1
        scf  <- t->obtenerStrings(aBuscar,&aux);
    while k > scf.size()
    ordenarArr(&scf);
    resultado <- priorizarPalabras(&scfAux,&scf,k);
    ordenarArr(&resultado);

    for i = 0 to k-1 do
        resultados.push_back(resultado[i].s);
        t->aumentarFrecuencia(resultado[i].s);
    return resultados;
```

Rendimiento y conclusiones

Luego de haber terminado el código y haberlo probado con diccionarios pequeños, se procedió a analizar el rendimiento respecto al tiempo de ejecución y el espacio en memoria utilizado. Para esto, se calculó el tiempo de ejecución en milisegundos (ms) de la construcción del Trie con diccionarios de 100, 500, 1000, 5000, 10000, 50000 y 100000 palabras. Por otro lado, usando la herramienta *Valgrind*, se calculó la memoria utilizada por el programa en kilobytes (KB), con los diccionarios mencionados anteriormente.

Los resultados son mostrados los siguientes gráficos:





Tamaño del diccionario (n)	Tiempo (ms)	Memoria (KB)
100	1	278
500	2	849
1000	3	1.444
5000	12	6.113
10000	23	11.704
50000	112	52.490
100000	214	98.857

Así, a modo de conclusión, se logró generar un predictor de textos utilizando la estructura de datos Trie, la que almacenará las palabras en forma de un árbol, que permite organizar y buscar los datos dentro de esta estructura de una manera rápida y sencilla, en comparación a guardar todas las palabras en un arreglo o vector.

Referencias

1. Ubuntu Wiki. (2009) “Valgrind_es - Ubuntu Wiki”, Recuperado de: https://wiki.ubuntu.com/Valgrind_es