UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica Final: Cifras y Letras

Práctica 6: Estructura de árbol para el diccionario

(Práctica opcional - puntuable)

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

1.1 Cifras y letras

En esta práctica, y las prácticas sucesivas, nos centraremos en el juego conocido como cifras y letras. Este juego se ha popularizado a través de concursos de televisión en distintos países.

Prueba de las letras

En esta práctica, retomaremos el problema de las letras, tratando de dar una solución más eficiente al mismo. Recordamos en qué consiste una partida del juego de las letras:

Esta parte del juego consiste en formar la mejor palabra posible (dependiendo de uno de los dos criterios que explicamos a continuación) a partir de un conjunto de letras extraídas al azar de una bolsa. Por ejemplo, dadas las siguientes letras:

O D Y R M E T

una buena solución posible sería METRO. El número de letras que se juegan en cada partida se decide de antemano, y las letras disponibles pueden repetirse. Existen dos modalidades de juego:

- Juego a longitud: En este modo de juego, se tiene en cuenta sólo la longitud de las palabras, y gana la palabra más larga encontrada
- Juego a puntos: En este modo de juego, a cada letra se le asigna una puntuación, y la puntuación de la palabra será igual a la suma de las puntuaciones de las letras que la componen

En esta práctica modificaremos la solución que hemos dado en las dos prácticas anteriores al problema de las letras. Dicha modificación nos permitirá almacenar nuestro diccionario de forma más eficiente, lo que producirá dos efectos en nuestro programa:

1. Necesitaremos almacenar menos información para representar el diccionario completo (lo podremos comprobar contando el número de letras almacenadas en total)
2. Podremos recuperar más rápidamente las palabras que se pueden formar dado un vector de letras, ya que la estructura de árbol en la que almacenaremos el diccionario nos permitirá explorar las posibles soluciones de una forma bastante eficiente.

1.2 Información necesaria para una partida de las letras: Archivos de entrada

Teniendo en cuenta la descripción del juego de las letras que hemos hecho en el apartado anterior, parece claro que vamos a necesitar tres almacenes principales de información para poder jugar una partida de las letras

Información sobre las letras

Como hemos dicho anteriormente, uno de los modos de juego a los que podemos jugar asigna una puntuación a cada letra, y la puntuación de la palabra será la suma de las puntuaciones de cada una de sus letras. Por tanto, necesitamos recoger la información de la puntuación para cada letra de algún sitio.

Además, hemos dicho que podríamos tener cada letra repetida un número de veces. No obstante, esto puede llevar a problemas en algunas partidas. Si todas las letras pudieran repetirse un número indeterminado de veces, podría ocurrir que en alguna partida sólo tuviésemos la letra Z muchas

veces, lo que dificultaría en gran medida formar una palabra. Además, parece lógico pensar que si las letras que más se repiten son letras que aparecen mucho en el diccionario, las palabras que se podrán formar serán más largas, haciendo el juego más interesante para los participantes. Por esto, la forma que tendremos de seleccionar las letras de cada partida será considerando un número de repeticiones de cada letra, formando una "bolsa" con todas ellas, y sacando al azar con probabilidad uniforme elementos de esa bolsa. Así, las letras con más repeticiones tendrán mayor probabilidad de salir, mientras que las que tengan pocas repeticiones saldrán menos a menudo (y no se repetirán en la misma partida si no hay más de una copia, ya que haremos extracciones sin reemplazamiento).

Por estos dos motivos, tendremos que guardar la siguiente información para cada letra del abecedario:

- Su puntuación
- El número de repeticiones disponible

Esta información la leeremos de un fichero como el que se muestra a continuación:

```
Letra  Cantidad Puntos
A      12       1
B      2        3
C      5        3
D      5        2
E      12       1
F      1        4
G      2        2
H      2        4
I      6        1
J      1        8
L      1        1
M      2        3
N      5        1
O      9        1
P      2        3
Q      1        5
R      6        1
S      6        1
T      4        1
U      5        1
V      1        4
X      1        8
Y      1        4
Z      1        10
```

Fichero 1: letras.txt

Crearemos una estructura de datos adecuada para almacenar esta información durante una partida, y poder aprovecharla para calcular las puntuaciones de nuestras palabras. Esta estructura de datos se denominará Conjunto de Letras (TDA LettersSet)

Además, como hemos dicho que necesitaremos extraer letras aleatorias para una partida, utilizando para ello las repeticiones del archivo anterior, construiremos también un contenedor específico para ello. Dicho contenedor será capaz de leer un Conjunto de Letras con las repeticiones de cada una

de las letras del juego, y crear la bolsa de la que haremos las extracciones aleatorias. Este TDA se denominará Bolsa de Letras (TDA LettersBag).

Para estas dos estructuras, mantendremos la implementación que habíamos propuesto en las prácticas anteriores (en particular, la práctica 4). Por tanto, mantendremos sin modificar los TDAs:

- LettersSet (implementado como un `map<char, LetterInfo>`)
- Bag (implementado como una clase template que nos permite extraer elementos aleatorios)
- LettersBag (implementado como una `Bag<char>`)

Información sobre las palabras - Diccionario

Finalmente, dado que vamos a jugar a un juego que trabaja sobre palabras, es importante establecer qué palabras estarán permitidas en una partida. Una “palabra” formada con las letras del ejemplo anterior puede ser METROYD, pero claramente no podríamos aceptar esta palabra en una partida, ya que no pertenece a nuestro idioma.

Por tanto, necesitaremos trabajar con un listado de palabras permitidas en nuestro juego, que funcionará a modo de diccionario. Sólamente aceptaremos como válidas aquellas palabras que pertenezcan a nuestro diccionario. La información de las palabras la recopilaremos de un archivo como el siguiente:

```
a
aaronita
aaronico
aba
ababa
ababillarse
ababol
abacal
abacalero
...
```

Fichero 2: diccionario.txt

La principal modificación que haremos a la práctica respecto a la solución obtenida en las prácticas anteriores estará centrada en la forma de guardar la información en el diccionario. En la siguiente sección comentaremos en detalle esta modificación, y cómo la misma nos ayuda a resolver el problema de forma más eficiente.

2.- Una nueva forma de guardar el diccionario - estructura de árbol

Lo primero en lo que tenemos que reparar para hacer esta modificación es en la cantidad de información redundante que estamos almacenando cuando utilizamos un conjunto para almacenar nuestro diccionario. Aunque no tenemos palabras repetidas en nuestro set, no estamos aprovechando una propiedad que cumplen las palabras que estamos almacenando: muchas de ellas comparten un prefijo común, más o menos largo, que en algunos casos llega a ser la palabra completa. Tomando como referencia las palabras que se muestran en el recuadro anterior, por ejemplo, tenemos que las palabras aaronita y aaronico comparten un prefijo de 6 letras, y sólo se diferencian en las últimas 2. Igualmente, aba está completamente contenida en ababa, y ésta a su vez está incluida casi por completo en ababillarse. Si encontramos una estructura en la que podamos almacenar las palabras de forma que compartan esos prefijos, podremos ahorrar mucho espacio eliminando toda la información redundante. Para ello, vamos a hacer uso de una estructura de árbol.

2.1 Módulo `tree` – implementación de un árbol general en formato LCRS

El módulo tree, que se proporciona completamente implementado, representa un árbol general, en el que cada nodo puede tener un número de hijos arbitrario. Para ello, cada nodo almacena la siguiente información:

- La etiqueta del nodo, es decir, la información que dicho nodo almacena, y que puede ser de cualquier tipo (para ello utilizamos una template de C++)
- Un puntero al nodo hijo más a la izquierda
- Un puntero al nodo hermano de la derecha.
- Un puntero al nodo padre

Esta forma de guardar un árbol es lo que se conoce como representación LCRS (Left Child Right Sibling). Permite que un nodo padre itere por todos sus hijos visitando a su hijo más a la izquierda, e iterando a partir de ahí por los nodos hermanos de la derecha hasta encontrar un nodo nulo. Así, el árbol queda completamente determinado (podemos recorrerlo por completo) simplemente almacenando un puntero al nodo raíz.

En nuestro caso, como queremos almacenar un conjunto de palabras, almacenaremos en cada nivel del árbol un carácter de la palabra, y reconstruiremos las palabras cogiendo todos los caracteres desde el nodo raíz (que contendrá un carácter especial y no utilizaremos para formar palabras) hasta un nodo determinado del árbol, haciendo un recorrido en profundidad.

Para poder determinar qué nodos nos permiten formar una palabra de nuestro diccionario, tendremos que poder indicar si en cada nodo termina o no una palabra. Por tanto, la etiqueta de cada nodo contendrá dos valores, un carácter, que representará una determinada letra de las palabras que se formen por esa rama, y un valor booleano, que nos indicará si en ese punto termina una palabra.

Se puede ver un ejemplo de árbol almacenando un conjunto de palabras en la figura 1. En concreto, en dicho ejemplo se almacenan las palabras:

- a
- ama
- amar
- amo
- amor
- cal
- col
- coz

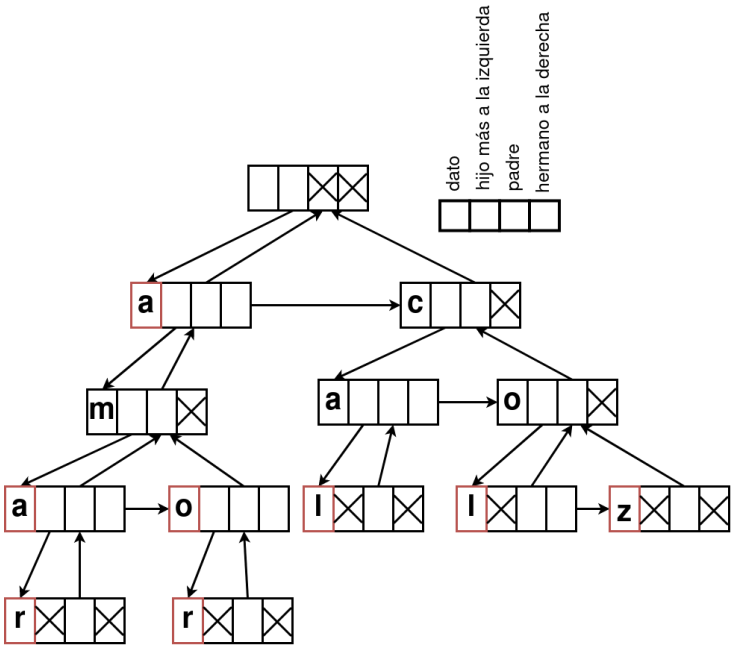


Figura 1: *Ejemplo de árbol representando un conjunto de palabras. Los nodos de color rojo indican el final de una palabra. Las palabras pueden reconstruirse viajando de la raíz a un nodo en rojo*

Al almacenar la información en este tipo de estructura, obtendremos dos ventajas:

- La cantidad de información almacenada se reduce, ya que para las palabras que compartan prefijo sólo almacenaremos una copia de cada elemento que compartan. En el ejemplo anterior, tenemos que almacenar 24 caracteres entre todas las palabras de nuestra lista, mientras que en el árbol sólo hay 12 nodos (13 si contamos la raíz, que no almacena ningún carácter).
- Buscar en el diccionario si hay una determinada palabra es rápido, ya que sólo tendremos que buscar, cuando nos encontremos en un determinado nodo, si entre sus hijos se encuentra la siguiente letra de nuestra palabra. Si en un determinado nivel no está la letra que nos toca buscar, la palabra no podrá estar en el diccionario, y si conseguimos encontrar todas las letras en orden, la palabra pertenecerá al diccionario si el último carácter está marcado como fin de palabra (es decir, si su bool es true)

La implementación completa del árbol se puede encontrar en los archivos `tree.h`, `tree.tpp` y `node.tpp`, que se encuentran dentro de la carpeta include. En principio, las clases plantilla deben declararse e implementarse en el mismo archivo. El uso de archivos `.tpp` permite separar en cierto modo la implementación de la declaración, pero todos los archivos que implementen una clase template deben estar en una carpeta de cabeceras. Podríamos almacenar estos archivos también en la carpeta src, pero tendríamos que indicar al compilador que busque archivos de cabecera también en dicha carpeta (con el parámetro -I del compilador). Para evitar tener que utilizar la carpeta de código también como carpeta de cabeceras, suele ser costumbre mantenerlos en la carpeta include.

Iterador para la clase árbol

La estructura de árbol que hemos implementado es muy útil para almacenar la información, pero puede ser un poco difícil de recorrer. Aquí es donde se ve realmente la utilidad de implementar iteradores para una clase. En concreto, querremos implementar un iterador en preorden para el árbol. El recorrido en preorden de un árbol funciona de la siguiente manera. Dado un nodo de un árbol en formato LCRS:

- Visitamos el nodo actual
- Visitamos en preorden su hijo más a la izquierda
- Visitamos en preorden su hermano a la derecha

Por tanto, nuestro iterador deberá soportar la funcionalidad para recorrer el árbol en preorden. Para implementar el iterador, en la parte privada del mismo, tendremos dos elementos, un nodo del árbol, que será el nodo que estamos visitando en ese momento, y un contador que nos indicará el nivel en el que nos encontramos en ese momento. En particular deberemos implementar las siguientes funciones para dicho iterador:

- Constructor por defecto: Creará un iterador que apunte a un nodo nulo y ponga el nivel a 0
- Constructor con parámetros, que reciba un nodo: Creará un iterador cuyo nodo actual sea el argumento que hemos pasado y ponga el nivel a 0
- Operadores de igualdad y distinto: Estos operadores compararán si el iterador está apuntando, o no, a un mismo nodo, es decir, si sus nodos internos son iguales
- Operador *: Devolverá el contenido del nodo al que estamos apuntando en este momento
- Operador ++: Nos permitirá avanzar al siguiente nodo. El siguiente nodo se define como:
    - Nuestro hijo a la izquierda si no es nulo. Si vamos al hijo, aumentaremos el contador de nivel en 1
    - Si nuestro hijo es nulo, nuestro hermano a la derecha si no es nulo. Aquí no aumentamos el contador, ya que nuestro hermano está en el mismo nivel que nosotros

- Si nuestro hermano también es nulo, el siguiente nodo a visitar es el hermano a la derecha de nuestro padre (a nuestro padre lo visitamos antes de bajar a sus hijos, por tanto no tenemos que volver a él). Si subimos al hermano de nuestro padre, decrementaremos en 1 el contador de nivel.
- Seguiremos buscando en el hermano derecho de nuestros ancestros hasta que encontremos un nodo no nulo, decrementando el contador de nivel por cada nivel que subamos.
- Habremos acabado de recorrer el árbol cuando lleguemos a la raíz, es decir, cuando encontremos un nodo cuyo padre sea nulo. En ese caso, pondremos el nodo actual al nodo vacío para marcar que hemos terminado de recorrer el árbol

Finalmente, tendremos que implementar las funciones begin_preorder y end_preorder, que apuntarán al nodo raíz y a un nodo nulo, respectivamente. **La primera tarea que tendréis que realizar es la implementación del iterador para el árbol, que será necesario para iterar posteriormente por las palabras de nuestro diccionaro.**

2.2 Clase diccionario

Una vez tenemos nuestra estructura de árbol definida e implementada, vamos a utilizarla para gestionar nuestro diccionario. Como ya hemos dicho, lo que almacenamos en cada nodo del árbol es una letra y un bool, que nos indica si en dicho nodo finaliza una palabra. Así, podremos encontrar las palabras de nuestro diccionario recorriendo el árbol en profundidad.

Este diccionario tendrá que tener la siguiente funcionalidad, al igual que teníamos en la práctica anterior:

- Constructor por defecto, que nos inicializa el árbol vacío
- Constructor de copia, que copia un diccionario en otro
- Destructor, que nos destruye el diccionario, liberando la memoria que hayamos reservado
- Una función para borrar todos los nodos de nuestro diccionario, `clear()`
- Una función para comprobar si una palabra existe en el diccionario, `exists(string word)`, que va recorriendo el árbol en profundidad buscando si la siguiente letra de la palabra que estamos buscando está entre los nodos hijos del nodo actual. Si conseguimos llegar a la última letra encontrándolas todas, y el bool del último nodo que visitemos es `true`, sabemos que nuestra palabra se encuentra en el diccionario.
- Una función para insertar una nueva palabra en nuestro diccionario, `insert(string word)`, que irá iterando en profundidad nuestro árbol e insertando cada una de las letras de la palabra (si la letra ya existe, no hacemos nada, simplemente seguimos insertando a partir de ese nodo las letras siguientes), y pondrá a `true` el booleano del último nodo que visitemos.
- Una función para eliminar palabras, `erase(string word)`, que buscará si esa palabra existe en el diccionario, y si existe, pondrá a `false` el booleano de su último nodo, para que dejemos de considerar esa palabra como parte del diccionario.
- El operador de asignación, que nos permita copiar el contenido de un árbol en otro
- La sobrecarga de los flujos de entrada y salida, que nos permitirán cargar nuestro diccionario desde un flujo de entrada e imprimir el diccionario en un flujo de salida
- Una función que nos permita obtener todas las palabras de una determinada longitud, que nos era útil para el solver que implementamos en la práctica 5

Además de estas funciones públicas, que nos dan la interfaz completa necesaria para trabajar con el Diccionario desde fuera de la clase, hay varias operaciones que son interesantes, porque se realizan muchas veces en el resto del código, pero que no tiene sentido darlas como funciones públicas, ya que son para la gestión interna de nuestra estructura de árbol. Las implementaremos, por tanto, como funciones privadas. Estas funciones son las siguientes:

- Una función que, dado un nodo del árbol, nos busque la posición de inserción de un hijo suyo de forma ordenada. Dado que estamos trabajando con una estructura de diccionario, en la que nos interesa que todas las palabras estén ordenadas en orden alfabético, nos interesará introducir las letras hijas de cada nodo de forma ordenada (si se observa el gráfico del apartado anterior, los hijos de cada nodo estaban ordenados por el orden alfabético. Un ejemplo claro son los hijos de la m de la izquierda, primero aparece la a, y después aparece la o. Si quisiéramos insertar ahora la palabra americano, la e se debería insertar entre estos dos elementos). Así que, dado un nodo y una letra a insertar como hija, nos interesará devolver el útlimo hijo de dicho nodo cuyo carácter sea menor o igual que el carácter que queremos insertar. Así, podremos insertar la nueva letra como un hermano a la derecha de dicho nodo. La función que devuelve dicha posición es la función `findLowerBoundChildNode(char character, node current)`
- Tras esta operación, como vamos a insertar muchos caracteres, también nos interesará separar esta funcionalidad en una función nueva (no es la misma operación buscar un nodo que insertarlo). Dicha función, `insertCharacter(char character, node current)`, intentará insertar el carácter como un nuevo nodo hijo del nodo current, y nos devolverá el nodo que se ha creado. Si entre los hijos de current ya existía un nodo con la letra que estamos intentando insertar, simplemente nos devolverá ese nodo, para que podamos seguir insertando por ahí, pero no tendría que insertar nada porque ya tenemos el hijo que queríamos dentro de nuestro árbol.
- Finalmente, como queremos buscar todas las palabras con una longitud dada, vamos a tener una función que nos permita reconstruir esa información. Esa función es `wordsByDepth(int remaining_levels, node current_node, string current_word)`. Irá descendiendo el árbol por niveles, añadiendo los caracteres que se encuentre al string actual, hasta llegar al nivel 0. Cuando hayamos llegado al nivel 0, meteremos las palabras que terminen en ese punto en una lista, y la devolveremos. Así podemos buscar de forma recursiva todas las palabras de una determinada longitud.

Todas estas funciones se os entregan ya implementadas. Con esta funcionalidad, ya podemos sustituir el diccionario implementado en la práctica 4 por este nuevo, sin necesidad de cambiar ninguna de las funciones que habíamos implementado en el resto de las clases de las prácticas 4 y 5.

Iterador del diccionario

La estructura que hemos definido es muy eficiente a la hora de almacenar la información, pero es poco útil para recuperar las palabras que hay almacenadas en nuestro diccionario. Por tanto, para facilitar la iteración sobre dichas palabras, vamos a implementar un iterador que nos permita visitarlas por orden alfabético. Para ello, haremos uso del iterador del árbol que hemos implementado en el ejercicio anterior. En concreto, el iterador del diccionario irá recorriendo cada uno de los nodos de nuestro diccionario, y parándose en cada uno de los nodos que esté marcado como final de palabra válida. Este iterador almacenará en su interior un iterador del árbol y una string que representa la palabra actual que estamos visitando, y tendrá los siguientes métodos implementados:

- Constructor por defecto, que inicializará la palabra actual a la cadena vacía y el iterador interno al iterador por defecto del árbol
- Constructor por parámetros, que recibirá otro iterador del diccionario y copiará tanto la string como el iterador del árbol al nuevo iterador del diccionario
- Operador *, que devolverá la palabra actual
- Operadores != y ==, que comprobarán si el iterador actual es igual o no al que se pasa como argumento (es decir, si los iteradores internos apuntan al mismo nodo)
- Operador ++, que avanzará el iterador interno hasta que se encuentre el siguiente nodo en el que termina una palabra válida. Se podrá utilizar el nivel del iterador para saber si hay que

añadir o eliminar letras de la palabra que llevamos hasta el momento (si el nivel es mayor, significa que hemos descendido un nivel, y por tanto, añadimos la nueva letra a la palabra que llevamos dentro del iterador. Si el nivel es menor, significa que estamos subiendo hacia los padres y tenemos que borrar letras de la palabra).

Finalmente, tenemos que implementar las funciones begin y end, que apunten a la primera palabra del diccionario, y a la palabra vacía con el iterador del árbol apuntando al final del árbol. **Tendréis que implementar el iterador del diccionario como parte de la práctica.**

Programa de prueba - `diccionario.cpp`

Para comprobar el funcionamiento correcto de los iteradores implementados, se proporciona implementado el archivo diccionario.cpp. El funcionamiento de este archivo es simple:

- Se carga el contenido del archivo que se pasa como argumento en un diccionario con estructura de árbol
- Se recorre el diccionario creado con un iterador, y se imprimen por pantalla todas sus palabras

Si los iteradores anteriores están correctamente implementados, el programa deberá funcionar con normalidad.

En los siguientes apartados, pasamos a comentar las mejoras de eficiencia que esta implementación nos produce en el árbol.

Mejora de la eficiencia en el juego de las letras

Con las funciones anteriores, ya podríamos trabajar con nuestra estructura de árbol en conjunto con el resto de las clases de las prácticas 4 y 5. No obstante, podemos implementar otra serie de funciones en nuestro árbol, para aprovechar su estructura para diversas operaciones. En particular, una funcionalidad en la que ganaremos mucho en eficiencia a la hora de formar palabras en el solver es delegando la búsqueda de palabras posibles dada una serie de letras a nuestro diccionario.

Hasta ahora, la mejor forma que teníamos de buscar soluciones era iterar sobre el Diccionario, extrayendo las palabras de una determinada longitud, y filtrando en dichas palabras las que no podíamos formar con las letras de la partida desde el solver. De esta forma, al final, el solver se quedaba sólamente con aquellas palabras que se podían formar, y de ahí seleccionamos las de mayor puntuación como solución final.

Ahora, en lugar de hacer esa búsqueda, que nos obligaría a recorrer el árbol en profundidad cada vez buscando las palabras de una determinada longitud, lo que vamos a hacer es pedirle al diccionario directamente la lista de palabras que podemos formar. Para ello, nos vamos a apoyar en dos funciones, una privada, recursiva, que tendrá la lógica completa que tenemos que implementar, y una pública, que simplemente se encargará de llamar a la función privada en la primera iteración, es decir, sobre el nodo raíz y con todas las letras que tenemos disponibles. Describimos a continuación, con detalle, el funcionamiento de esas dos funciones. Empezamos con la función privada:

Búsqueda de soluciones en profundidad de forma recursiva

Para hacer la búsqueda en profundidad, haremos uso de la función:

```
list<string> getCoincidentWords(multiset<char> &characters,
                                 node current_node, string current_word);
```

Esta función hará lo siguiente:

- Crea una lista de palabras resultado, vacía: `result`
- Si el nodo en el que estamos es un nodo nulo, hemos terminado, devolvemos la lista de resultados (en el nodo en el que estamos no puede haber solución ni soluciones por debajo, ya que un nodo nulo no puede tener hijos, hemos terminado de explorar esta rama)
- Si el nodo en el que estamos tiene una palabra válida, es decir, si su bool está a true, metemos la palabra completa en la lista de resultados. Para ir almacenando esa palabra completa, iremos introduciendo los caracteres en el string `current_word`
- Visitamos el nodo hijo más a la izquierda del nodo en el que estamos, `current_node.left_child()`
- Mientras sigamos teniendo hijos:
    - Buscamos si el carácter del hijo está entre los caracteres disponibles
    - Si está, añadimos ese carácter a la palabra actual, `current_word`, eliminamos el carácter de los caracteres restantes, y llamamos a la función `getCoincidentWords` con el nodo hijo, la lista de caracteres sin el carácter que acabamos de borrar, y con la string `current_word` con el nuevo carácter al final
    - Juntamos la lista de palabras posibles que nos ha devuelto nuestro hijo con la nuestra
    - Visitamos el siguiente hijo, que es el hermano a la derecha del nodo en el que estamos, `current_node.right_sibling()`
    - Devolvemos el carácter que habíamos sacado al visitar a nuestro hijo al conjunto de caracteres posible, para que esté disponible para sus hermanos
- Finalmente, devolvemos la lista de palabras que hemos formado

Con esta estructura recursiva, la función pública que tenemos que crear es la siguiente:

```
vector <string> getPossibleWords(vector<char> available_characters)
```

Que simplemente transformará el vector de caracteres en un multiset (esto se hace por temas de eficiencia, ya que vamos a tener que buscar en este vector muchas veces, y buscar en un multiset es O(log(n)) mientras que buscar en un vector es O(n)), llamará a la función privada anterior utilizando como `current_node` la raíz del árbol (es el punto por el que empezamos nuestra búsqueda) y como `current_word` la palabra vacía, y finalmente convertirá la lista de palabras en un vector (de nuevo, se hace esta conversión por temas de eficiencia, ya que unir dos listas es O(1), mientras que concatenar dos vectores es O(n), así que por dentro del algoritmo es mejor llevar listas) y lo devolverá al usuario.

Como podréis imaginar, esta forma de buscar palabras en el diccionario, aunque pueda ser un poco más enrevesada que la que hacíamos con el set, es mucho más eficiente, porque no tendremos que buscar por todas las palabras, si no que en cada caso sólamente exploraremos los nodos que nos interesan, es decir, aquellos caminos que podemos seguir formando en función a las letras que nos queden disponibles.

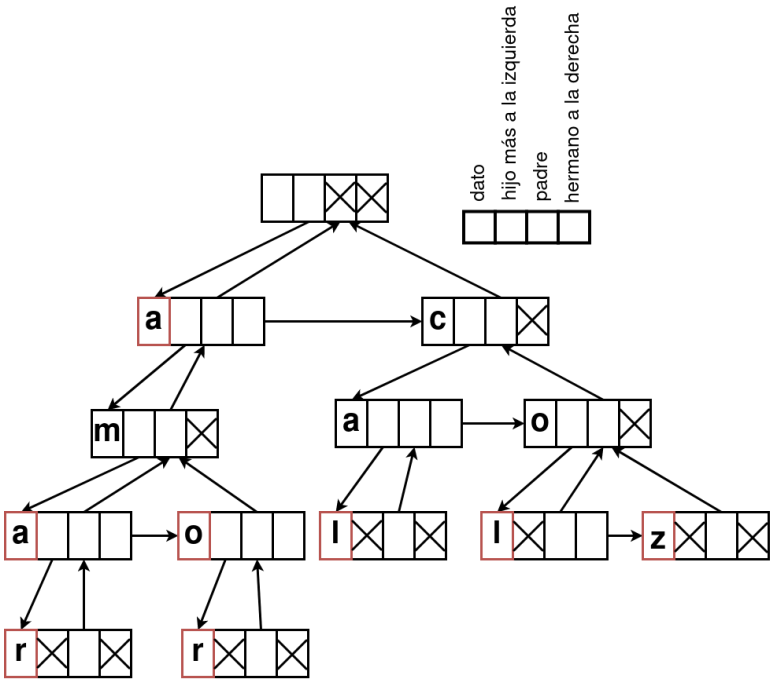
Las funciones que realizan la búsqueda se os darán implementadas, dado que puede ser compleja su implementación. No obstante, es importante que entendáis bien cómo funcionan, ya que tendréis que analizar su eficiencia en el desarrollo de la práctica.

Finalmente, hay otro punto en el que ganamos eficiencia, y que hemos comentado pero no explorado. En particular, hablamos de la eficiencia en espacio. En el siguiente apartado se hace un análisis de esta mejora, y se explica qué funciones tendréis que implementar sobre el diccionario para mostrar la diferencia entre el número de veces que guardamos una letra en el diccionario, y el número de veces que la letra se utiliza.

Mejora de la eficiencia en espacio para el diccionario

Como hemos visto en el ejemplo del principio de la práctica, necesitamos guardar muchos menos caracteres con esta representación que con la representación anterior, pero ¿hasta qué punto hemos reducido el número de caracteres que almacenamos?. Para dar respuesta a esa pregunta, vamos a implementar cierta funcionalidad sobre nuestro árbol para contar el número de veces que guardamos cada letra en nuestra estructura de árbol, y el número de veces que se utiliza cada una de estas letras para formar palabras. **Esta será una de las partes de implementación que tendréis que realizar para esta práctica.**

Como hemos dicho, vamos a diferenciar entre dos conteos distintos para cada letra en nuestro diccionario. Nos referiremos a dichos conteos como número de apariciones y número de usos, cuya diferencia se describe a continuación. Recuperamos el ejemplo del principio para ver las diferencias entre un concepto y otro:



Número de apariciones: Definimos el número de apariciones de una letra como el número de veces que aparece esa letra en nuestro diccionario, es decir, el número de nodos que tienen ese carácter almacenado. En el ejemplo anterior, el número de apariciones de la letra “a” es 3. Para contar dichas apariciones, simplemente tenemos que recorrer todos los nodos del árbol, llevando un contador con el número de apariciones de la letra, que aumentaremos en uno cada vez que visitemos un nodo que tenga ese carácter almacenado.

Número de usos: El número de usos de una letra se define como el número de veces que esa letra aparece entre las palabras de nuestro diccionario. Si recordáis la práctica 4, el número de veces que se usaba una letra en el diccionario coincidía con el número de veces que aparecía en el mismo, porque nunca se usaba una misma aparición de la letra para formar dos palabras distintas.

Aquí, el número de apariciones será mucho menor que el número de usos. En particular, el número de usos de la letra a, teniendo en cuenta que en nuestro diccionario están las palabras

a, ama, amar, amo, amor, cal, col, coz

Es de 8.

Se tendrán que implementar las siguientes funciones para dotar a nuestro diccionario de la funcionalidad necesaria para hacer estos conteos:

Conteo de ocurrencias en el árbol

`int getOccurrences(char c):`

Esta función itera sobre todos los nodos del diccionario (para esto puede ser útil el iterador que habéis implementado para el diccionario), y lleva un contador del número de veces que se ha visitado un nodo que contenía el carácter c. Finalmente, devuelve ese carácter tras iterar por todos los nodos

Conteo de usos en el diccionario

La mejor forma de resolver este problema es de forma recursiva con dos funciones, una pública y una privada (que implementa toda la lógica compleja):

`private: pair<int, int> getTotalUsages(node curr_node, char c):`

Esta función nos va a devolver una pareja de valores, el número de ocurrencias del carácter en los nodos que cuelgan del nodo actual, y el número de palabras que terminan por debajo del nodo actual (son los dos valores del par que se devuelve). La función, dado el nodo actual y el carácter que estamos buscando, realiza los siguientes pasos:

- Inicialmente, el número de ocurrencias del carácter y el número de palabras que terminan por debajo del nodo actual valen 0
- Si su hijo a la izquierda no es nulo, llama a getTotalUsages con ese nodo y el carácter de interés, y guarda el resultado
- Si su hermano a la derecha no es nulo, llama a getTotalUsages con ese nodo y el carácter de interés, y guarda el resultado
- El número de usos es la suma del número de usos de nuestro hermano a la derecha y nuestro hijo a la izquierda, y el número de palabras completas es la suma del número de palabras completas en nuestro hermano a la derecha y nuestro hijo a la izquierda
- Si en el nodo actual el carácter que tenemos coincide con el carácter que estamos buscando, aumentamos el número de usos de la letra en tantas unidades como palabras terminen en nuestro hijo de la izquierda. Además, si en nosotros termina una palabra, tendremos que aumentar en 1 dicho valor
- Si en el nodo actual termina una palabra, tendremos que aumentar el número de palabras completas en 1
- Finalmente, devolvemos los valores del número de usos y el número de palabras completas que hemos calculado

`public: int getTotalUsages(char c):`

La función pública que utilizamos, simplemente llama a la función anterior sobre la raíz del árbol, y devuelve el primer valor del par (ya que el segundo nos sirve para hacer cálculos intermedios en la función privada, pero no nos da información de cara a los usuarios).

No tenéis que implementar esta función de forma idéntica a la nuestra, pero sí que pedimos una función recursiva para resolver este apartado. La descripción anterior es la solución que nosotros hemos encontrado para el conteo de los usos de una letra, si alguno de vosotros encuentra otra forma válida de hacerlo, estaremos encantados de discutir dicha solución con vosotros. no obstante, este es un ejercicio para practicar la implementación de funciones recursivas en árboles. **Cualquier implementación no recursiva, aunque devuelva un resultado correcto, no será considerada como válida.**

Programa de prueba - `cantidad_letras.cpp`

Para comprobar el funcionamiento correcto de las funciones de conteo de letras implementadas (tanto para el número de ocurrencias como para el uso), debéis implementar un pequeño programa que las utilice. Dicho programa se llamará `cantidad_letras.cpp`, y su funcionamiento será el siguiente: Recibirá como argumentos un fichero de diccionario y un fichero de letras (podéis usar la implementación de LettersSet de la práctica anterior, debería ser compatible con esta práctica), creará el LettersSet y el Dictionary con la información de dichos ficheros, y para cada letra del diccionario calculará el número de veces que se usa esa letra y el número de ocurrencias de la letra en la estructura de diccionario. En el diccionario de ejemplo con el que venimos trabajando en este guión, la salida debería ser la siguiente:

Letra	Usos	Ocurrencias
A	8	3
C	3	1
L	2	2
M	4	1
R	2	2
O	4	2
Z	1	1

3.- Práctica a Realizar

Atendiendo a la explicación de la práctica de los apartados anteriores, hay tres ejercicios para resolver en esta práctica

1.- Análisis de eficiencia de la búsqueda de soluciones:

Para este ejercicio, se deberá hacer un análisis exhaustivo de la eficiencia del algoritmo de búsqueda de palabras posibles a formar dada una serie de letras. Para este ejercicio se pide:

1. Un análisis teórico de la eficiencia de dicho algoritmo, en notación O, que dependa del número de letras de la partida y del tamaño del diccionario
2. Tras el análisis teórico, un análisis empírico **lo más completo posible** de la eficiencia del algoritmo. A modo de indicación, las variables que influyen en el tiempo de ejecución de este algoritmo son el número de letras con las que se juega una partida, y el tamaño del diccionario

Para este ejercicio **no hay que implementar código, más allá del necesario para hacer las ejecuciones para los tamaños de entrada y mediciones de tiempos.** El código necesario para resolver el problema de las letras con la estructura de árbol se os ofrece implementado por completo, por lo que no tendréis que preocuparos de dicha implementación. Para la exposición de resultados en el análisis de eficiencia, deberéis usar una página relacionada de la documentación de la práctica.

2.- Implementación de los iteradores del árbol y el diccionario

Para este ejercicio, se deben implementar el iterador del módulo tree y el iterador para las palabras del diccionario, y comprobar su correcto funcionamiento con el programa `diccionario.cpp`.

3.- Implementación de los contadores de usos y ocurrencias de las letras

Para este ejercicio, se deben implementar los contadores de uso y ocurrencias para las letras del diccionario, junto con el programa `cantidad_letras.cpp` que compruebe su correcto funcionamiento.

4.- Documentación y entrega

Toda la documentación de la práctica se incluirá en el propio Doxygen generado, para ello se utilizarán tanto las directivas Doxygen de los archivos .h y .cpp como los archivos .dox incluídos en la carpeta doc.

1. Código:
- Se dispone de la siguiente estructura de ficheros:
  - /
    - estudiante/ (Aquí irá todo lo que desarrolle el alumno)
      - doc/ (Imágenes y documentos extra para Doxygen)
      - include/ (Archivos cabecera)
      - src/ (Archivos fuente)
    - CMakeList.txt (Instrucciones CMake)
    - Doxyfile.in (Archivo de configuración de Doxygen)
    - juez.sh (Script del juez online [**HAY QUE CONFIGURARLO**])
- Es importantísimo leer detenidamente y entender qué hace CMakeList.txt.
- El archivo Doxyfile.in no tendríais por qué tocarlo para un uso básico, pero está a vuestra disposición por si queréis añadir alguna variable (no cambiéis las existentes)
- juez.sh crea un archivo submission.zip que incluye ya TODO lo necesario para subir a Prado a la hora de hacer la entrega. No tenéis que añadir nada más, especialmente binarios o documentación ya compilada.

4. Elaboración y puntuación
- La práctica se realizará POR PAREJAS. Los nombres completos se incluirán en la descripción de la entrega en Prado. Cualquiera de los integrantes de la pareja puede subir el archivo a Prado, pero SOLO UNO.
- Desglose:
  - **(0.3)** Ejercicio 1 - Análisis de eficiencia
  - **(0.1)** Ejercicio 2 - Implementación de iteradores
  - **(0.1)** Ejercicio 3 - Implementación de los contadores de letras
- La fecha límite de entrega aparecerá en la subida a Prado.