**UNIVERSIDAD MAYOR DE SAN ANDRÉS**

**FACULTAD DE CIENCIAS PURAS Y NATURALES**

**CARRERA DE INFORMÁTICA**



**TESIS DE GRADO**

**“DISEÑO E IMPLEMENTACIÓN DE UNA ESTRUCTURA DE DATOS EFICIENTE PARA LA PREDICCIÓN DE TEXTO CON ACTUALIZACIONES”**

PARA OPTAR AL TÍTULO DE LICENCIATURA EN INFORMÁTICA

MENCIÓN: INGENIERÍA DE SISTEMAS INFORMÁTICOS

**POSTULANTE:** BRANIMIR FERNANDO ESPINOZA ARGOLLO

**TUTOR METODOLÓGICO:** Lic. ELIZABETH PATRICIA POMMIER GALLO

**ASESOR:** M.Sc. JORGE HUMBERTO TERÁN POMIER

**LA PAZ – BOLIVIA**

**2019**

ÍNDICE

[CAPÍTULO I 1](#_Toc26917000)

[MARCO REFERENCIAL 1](#_Toc26917001)

[1.1 INTRODUCCIÓN 1](#_Toc26917002)

[1.2. ANTECEDENTES 2](#_Toc26917003)

[1.3. PLANTEAMIENTO DEL PROBLEMA 5](#_Toc26917004)

[1.3.1. PROBLEMA CENTRAL 5](#_Toc26917005)

[1.3.2. PROBLEMAS SECUNDARIOS 6](#_Toc26917006)

[1.4. DEFINICIÓN DE OBJETIVOS 6](#_Toc26917007)

[1.4.1. OBJETIVO GENERAL 6](#_Toc26917008)

[1.5. HIPÓTESIS 6](#_Toc26917009)

[1.5.1. OPERACIONALIZACIÓN DE VARIABLES 7](#_Toc26917010)

[1.6. JUSTIFICACIÓN 7](#_Toc26917011)

[1.6.1. ECONÓMICA 8](#_Toc26917012)

[1.6.2. SOCIAL 8](#_Toc26917013)

[1.6.3. CIENTÍFICA 8](#_Toc26917014)

[1.7. ALCANCES Y LÍMITES 8](#_Toc26917015)

[1.7.1. ALCANCES 8](#_Toc26917016)

[1.7.2. LÍMITES 9](#_Toc26917017)

[1.8. APORTES 9](#_Toc26917018)

[1.8.1. APORTE PRÁCTICO 10](#_Toc26917019)

[1.8.2. APORTE TEÓRICO 10](#_Toc26917020)

[1.9. METODOLOGÍA 10](#_Toc26917021)

[CAPÍTULO II 11](#_Toc26917022)

[MARCO TEÓRICO 11](#_Toc26917023)

[2.1. CONCEPTOS BÁSICOS 11](#_Toc26917024)

[2.1.1 ALFABETO 11](#_Toc26917025)

[2.1.2 SÍMBOLO 11](#_Toc26917026)

[2.1.3 CADENA 11](#_Toc26917027)

[2.1.4 OPERACIONES CON CADENAS 12](#_Toc26917028)

[2.1.4.1 LONGITUD 12](#_Toc26917029)

[2.1.4.2 IGUALDAD DE CADENAS 12](#_Toc26917030)

[2.1.4.3 CONCATENACIÓN DE CADENAS 12](#_Toc26917031)

[2.1.4.4 PREFIJO 13](#_Toc26917032)

[2.2. ÁRBOL DE PREFIJOS TRIE 13](#_Toc26917033)

[2.2.1. CONCEPTOS BÁSICOS 13](#_Toc26917034)

[2.2.2. BÚSQUEDA EN UN TRIE 15](#_Toc26917035)

[2.2.3 INSERCIÓN EN UN TRIE 15](#_Toc26917036)

[2.2.4 IMPLEMENTACIÓN DE UN TRIE 16](#_Toc26917037)

[2.2.4.1 LA CLASE TRIE Y REPRESENTACIÓN DE UN NODO 16](#_Toc26917038)

[2.2.4.2 BÚSQUEDA EN UN TRIE 16](#_Toc26917039)

[2.2.4.3 INSERCIÓN EN UN TRIE 17](#_Toc26917040)

[2.2.4.4 UTILIZACIÓN 17](#_Toc26917041)

[2.2.5 COMPLEJIDAD 19](#_Toc26917042)

[2.4. ÁRBOL CARTESIANO(TREAP) 19](#_Toc26917043)

[2.4.1 OPERACIONES DE UN TREAP 19](#_Toc26917044)

[2.4.1.1 OPERACION SPLIT 20](#_Toc26917045)

[2.4.1.2 OPERACION MERGE 20](#_Toc26917046)

[2.4.1.3 OPERACION DE INSERSION 20](#_Toc26917047)

[2.4.1.4 OPERACION DE ELIMINACIÓN 20](#_Toc26917048)

[2.4.1.5 BUSQUEDA 21](#_Toc26917049)

[2.4.2 IMPLEMENTACIÓN DE UN TREAP 21](#_Toc26917050)

[2.5 TREAPS IMPLICITOS 21](#_Toc26917051)

[2.5.1 IDEA GENERAL 22](#_Toc26917052)

[2.5.2 IMPLEMENTACION DE UN TREAP IMPLICITO 22](#_Toc26917053)

[2.6 EL ALGORITMO DE MATANI D. 22](#_Toc26917054)

[2.6.1 CONSTRUCCIÓN DE LA ESTRUCTURA 22](#_Toc26917055)

[2.6.2 CONSULTAS 23](#_Toc26917056)

[CAPITULO III 24](#_Toc26917057)

[MARCO APLICATIVO 24](#_Toc26917058)

[3.1 ESPECIFICACION DE LA ESTRUCTURA DE DATOS 24](#_Toc26917059)

[3.2 DISEÑO DE LA ESTRUCTURA 25](#_Toc26917060)

[3.2.1 UN TRIE QUE RESPONDE POSICIONES 25](#_Toc26917061)

[3.2.1.1 POSICION DE UNA CADENA EN EL TRIE 26](#_Toc26917062)

[3.2.1.2 RANGO DE POSICIONES DADO UN PREFIJO 29](#_Toc26917063)

[3.2.2 CADENA EN EL TRIE DADA UNA POSICIÓN 30](#_Toc26917064)

[3.2.2.1 COMPLEJIDAD COMPUTACIONAL 34](#_Toc26917065)

[3.2.3 TREAP IMPLICITO CON RESPUESTA EN RANGOS 34](#_Toc26917066)

[3.2.3.1 MAXIMO EN UN RANGO EN UN TREAP IMPLICITO 34](#_Toc26917067)

[3.2.4 ALTERNATIVA A CONSTRUCCIÓN EN TIEMPO LINEAL DE UN TREAP IMPLICITO 37](#_Toc26917068)

[3.2.4.1 CONSTRUCCION DEL ARBOL BINARIO 37](#_Toc26917069)

[3.2.4.2 ASIGNACIÓN DE PRIORIDADES 39](#_Toc26917070)

[3.2.4.3 COMPLEJIDAD COMPUTACIONAL 41](#_Toc26917071)

[3.2.5 COMBINANDO LAS ESTRUCTURAS - LA ESTRUCTURA SUGGEST 42](#_Toc26917072)

[3.2.5.1 ESTRUCTURA BASE 42](#_Toc26917073)

[3.2.5.2 CARGA DE DATOS Y CONSTRUCCIÓN 43](#_Toc26917074)

[3.2.5.3 CONSULTAS 45](#_Toc26917075)

[3.2.5.4 ADICIONES Y ACTUALIZACIONES 46](#_Toc26917076)

[3.5 CODIGO FUENTE 48](#_Toc26917077)

[BIBLIOGRAFÍ A 50](#_Toc26917078)

[ANEXOS 52](#_Toc26917079)

[ANEXO C: CODIGO DE LA ESTRUCTURA DE DATOS PROPUESTA 55](#_Toc26917080)

[Archivo trie.hpp 55](#_Toc26917081)

[Archivo treap.hpp 58](#_Toc26917082)

[Archivo principal 62](#_Toc26917083)

[ANEXO D: CODIGO DE LA VERSION ORIGINAL(REFACTORIZADO) 64](#_Toc26917084)

# CAPÍTULO I

# MARCO REFERENCIAL

# 1.1 INTRODUCCIÓN

Autocompletar es una función que sugiere palabras o frases completas después de que un usuario ha escrito unas pocas letras. La función aumenta la velocidad de entrada de texto, especialmente en dispositivos móviles, ya que no es necesario escribir todas las letras de una palabra. La funcionalidad de autocompletar se encuentra comúnmente en los motores de búsqueda y aplicaciones de mensajería.

Un buen autocompletador debe ser rápido y actualizar la lista de sugerencias inmediatamente después de que el usuario escriba la siguiente letra (Ajanki A., 2016).

Un caso especial de esta función de autocompletar es cuando se tiene demasiadas opciones de sugerencias para poder completar el prefijo actual y se debe seleccionar un pequeño subconjunto que contenga las mejores K respuestas de entre todas las posibles, para ello asociaremos un valor numérico a cada posible respuesta, al cual llamaremos peso, mientras más grande sea el peso, será considerada como una mejor respuesta, el peso dependerá del contexto en el que se maneje la función de autocompletar, por ejemplo si tenemos como cadenas los títulos de las entradas de un blog el peso de cada título será el número de visitas que tuvo dicha entrada del blog. La presente tesis abordara este caso especial.

Motores de búsqueda como los de Google, Bing, Yahoo, DuckDuckGo implementan predicción de texto basado en autocompletado de prefijos en sus buscadores, la cantidad de información que manejan estos motores de búsqueda tranquilamente supera la escala de los millones, también como la información en internet está en constante crecimiento y cambio, la información que almacenan estos motores de búsqueda debe ser actualizada constantemente. Es por eso que se utilizan algoritmos y estructuras de datos muy eficientes en tiempo y espacio para realizar dicha tarea.

Lamentablemente la mayoría de los algoritmos de dichos motores de búsqueda no están disponibles para el público en general al ser código privativo. Una excepción es la librería cpp-libface del motor de búsqueda de DuckDuckGo la cual puede ser encontrada en sus repositorios de github.

Dhruv Matani el creador de cpp-libface realiza una publicación científica en 2011, en el que describe el algoritmo y estructuras de datos utilizados para desarrollar cpp-libface (Matani, 2011). Cpp-libface carga desde un archivo un conjunto de palabras/frases como posibles respuestas de autocompletado, después de un pre-proceso está listo para responder a consultas, lamentablemente cpp-libface no permite adicionar nuevas palabras/frases una vez construida la estructura interna.

Existen alternativas para la predicción de texto basado autocompletado en la actualidad, que a través de servicios web ofrecen a los clientes la posibilidad de cargar datos para el autocompletado en su servidor y realizar consultas a este, tales son los casos de Solr o ElasticSearch, sin embargo, la gran complejidad y tamaño de dichas soluciones hacen casi imposible su aplicación en proyectos pequeños o medianos como: aplicaciones para teléfonos móviles, editores de texto o aplicaciones sin acceso a una red.

En la presente tesis se presenta un algoritmo que servirá como una alternativa de solución para el problema de predicción de texto basado en autocompletado de prefijos, éste permitirá insertar nuevas palabras/frases en tiempo de ejecución. El algoritmo estará basado en la idea que Dhruv Matani utilizó para desarrollar cpp-libface.

Posteriormente se realizarán experimentos para determinar su rendimiento en comparación con otros algoritmos similares disponibles en la literatura.

# 1.2. ANTECEDENTES

Muchos motores de búsqueda como Google, Bing y Yahoo! muestran sugerencias de búsqueda cuando los usuarios ingresan frases de búsqueda en sus interfaces. Esas sugerencias ayudan al usuario a buscar la información que sea más relevante. También ayuda al usuario si este no está seguro de lo que está buscando, pero tiene una vaga idea de que es lo que quiere (Matani, 2011).

En (Matani D., 2011) se describen algunos enfoques para solucionar el problema planteado en la presente tesis junto a las desventajas de cada enfoque, a continuación, se nombran algunos de dichos enfoques:

* El **Enfoque Ingenuo,** en este enfoque pre-procesamos las n cadenas de entrada ordenándolas lexicográficamente, para responder a una consulta podemos utilizar búsqueda binaria para encontrar el primer elemento en la lista ordenada que tiene como prefijo la cadena de consulta, y de forma similar para el último elemento con dicho prefijo, llamemos A y B respectivamente a las posiciones en la lista ordenada de los elementos mencionados, a continuación ordenamos los elementos en el rango [A, B] según su peso y tomamos los K mayores. La principal desventaja de este enfoque es la velocidad de respuesta, el problema radica en que el rango [A, B] podría ser en todos los casos toda la lista.
* **Enfoque intensivo de espacio**, en este enfoque por cada cadena del diccionario se guarda todos los prefijos de las cadenas originales junto al peso de la cadena original, a continuación, se ordenan todas las cadenas obtenidas en orden lexicográfico ordenando por el peso de mayor a menor en caso de tener cadenas iguales. Para responder una consulta procedemos igual que en el enfoque ingenuo buscando el primer elemento con el prefijo buscado utilizando búsqueda binaria, a continuación, tomamos los K primeros elementos a partir de esa posición. La desventaja de este enfoque es el espacio necesario para guardar todos los prefijos de las cadenas originales.
* **Árbol de búsqueda ternaria**, podemos optimizar el anterior enfoque guardando todos los prefijos generados en un árbol de búsqueda ternaria, esto significa un gran ahorro de espacio en memoria. El problema radica en que el peor caso para el tiempo de consulta en un árbol de búsqueda ternaria es lineal en la cantidad de palabras insertadas, alcanzar este peor caso depende en el orden en el que se insertaron las cadenas en el árbol.

(Matani D., 2011) propone un algoritmo que supera a los ya mencionados antes, en primera instancia el algoritmo ordena lexicográficamente todas las cadenas como en el primer enfoque, a continuación, construye un Árbol de Segmentos con los pesos de las cadenas. Para responder a las consultas utiliza una cola de prioridad, en la cual va guardando las posibles respuestas para la consulta, se guardan datos en formato de cuaternas (mayor en rango, índice, a, b), se utiliza el Segment Tree para hallar el elemento ‘mayor en rango’ y su correspondiente ‘indice’ en el rango [a, b), en primera instancia se adiciona a la cola el rango original [A, B). Posteriormente en cada iteración se procede a obtener el elemento con mejor respuesta de la cola de prioridad y si es posible se crean 2 nuevos rangos (mayor en rango A, índice A, a, índice) y (mayor en rango B, índice B, índice + 1, b) los cuales son adicionados a la cola de prioridad. De esta manera después de K iteraciones se tienen las K mejores repuestas.

Este algoritmo tiene una mejora substancial en el tiempo de respuesta, pero tiene la desventaja (al igual que el enfoque ingenuo) de que no acepta actualizaciones como ser ingresar una nueva cadena al diccionario de respuestas, ya que esto implicaría la reconstrucción total del árbol de segmentos.

En (Diethelm N. 2014) se describe una estructura llamada Suggest Trees muy parecida al tercer enfoque planteado en (Matani D. 2011), en esta alternativa se debe conocer de antemano el valor de K, en cada nodo del árbol de búsqueda ternaria se guarda una lista ordenada de las K mejores soluciones en el subárbol que tiene como raíz a ese nodo.

La principal desventaja de esta estructura es que se debe conocer de antemano el valor de K.

En (Vaijapurkar A. Patani R. Kashyap Jha V., 2016) se propone un algoritmo para el problema de autocompletar usando probabilidad condicional, el algoritmo implementa un *trie* y calcula un costo a cada arco mediante la siguiente fórmula:

Para responder a una consulta realiza un recorrido en el trie siguiendo los caracteres de la cadena de consulta, una vez situado en el último nodo del recorrido se realiza un recorrido siempre seleccionando el arco con mayor costo, así hasta llegar a un nodo hoja, la cadena que termina en ese nodo hoja es la respuesta a la consulta.

Una desventaja del algoritmo es que K=1, porque solo responde una posible sugerencia. Otra desventaja es que implementa un *trie* sin ningún tipo de optimización y lo hace ineficiente en memoria para una gran cantidad de datos.

Pero en cambio una ventaja de dicho algoritmo es que cada vez que se realiza una consulta, la cadena de la consulta es adicionada al *trie*, haciendo así que los costos de los arcos cambien para futuras consultas, mejorando así la experiencia de usuario.

# 1.3. PLANTEAMIENTO DEL PROBLEMA

La autocompletación es una técnica prominente utilizada ampliamente por casi todos los principales motores de búsqueda, editores que trabajan en la entrada del usuario. Después de ingresar un nuevo carácter en el editor, se filtran las sugerencias que coinciden con la palabra actualizada y sugiere las palabras mejor calificadas. (Vaijapurkar A. Patani R. Kashyap Jha V., 2016)

La autocompletación solo es efectiva si aparece "instantáneamente" en una escala de tiempo aceptable para el ser humano, que tiene un límite superior de 100 ms. (Vaijapurkar A. Patani R. Kashyap Jha V., 2016)

Es por eso que cuando se trata de diseño e implementación de algoritmos, la eficiencia es un factor muy importante a considerar, refiriéndose con eficiencia al tiempo de respuesta y a la cantidad de memoria utilizada.

También, ofrecer respuestas inapropiadas es igual de malo que no ofrecer respuestas. Esto distrae al usuario y lo pone ansioso. (Vaijapurkar A. Patani R. Kashyap Jha V., 2016)

De las estructuras de datos y algoritmos mencionados el problema es que la mayoría de ellos son estructuras estáticas, es decir que no aceptan actualizaciones o cambios, como por ejemplo ingresar nuevas cadenas al diccionario de respuestas, esta es la principal tarea que se pretende abordar en la presente tesis.

## 1.3.1. PROBLEMA CENTRAL

¿Cómo la predicción de texto que soporte actualizaciones mantendrá la eficiencia?

## 1.3.2. PROBLEMAS SECUNDARIOS

* Los algoritmos y estructuras de datos eficientes existentes para la predicción de escritura no cuentan con la capacidad de soportar actualizaciones sin que sea necesario reconstruir o reprocesar toda la estructura.
* El código fuente de algoritmos usados por empresas grandes como ser Google, Bing, Yahoo para la predicción de escritura no están disponibles para el público en general al tratarse de códigos privativos.
* Existe poca información es español de las estructuras sobre las estructuras que se utilizaran.

# 1.4. DEFINICIÓN DE OBJETIVOS

## 1.4.1. OBJETIVO GENERAL

- Diseñar e implementar una estructura de datos eficiente para la predicción de texto que soporte actualizaciones.

- Modificar el algoritmo de Matani D. para implementar una estructura de datos eficiente para la predicción de texto que soporte actualizaciones.

- Combinar distintos algoritmos y estructuras de datos en una estructura de datos eficiente para la predicción de texto que soporte actualizaciones.

- Utilizar distintos algoritmos y estructuras de datos en una estructura de datos eficiente para la predicción de texto que soporte actualizaciones.

# 1.5. HIPÓTESIS

La combinación de distintos algoritmos y estructuras de datos para la predicción de texto que soporte actualizaciones mantendrá la eficiencia.

## 1.5.1. OPERACIONALIZACIÓN DE VARIABLES

Una vez elaborada la hipótesis se procedió a la identificación de variables independientes y dependientes:

* Variable independiente: Combinación de algoritmos y estructuras de datos en una única estructura de datos para la predicción de texto que soporte actualizaciones.
* Variable dependiente: Eficiencia de la estructura de datos

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable** | **Tipo de variable** | **Operacionalización** | **Categorización** | **Definición** |
| Eficiencia de la estructura de datos | Dependiente | Cantidad de recursos computacionales utilizados para la ejecución de la estructura de datos. | 1 Complejidad Temporal  2 Complejidad Espacial | 1 Indica el tiempo tomado por la computadora para ejecutar un determinado proceso  2 Indica la cantidad de memoria utilizada por un algoritmo o estructura de datos para ejecutar un proceso |
| **Indicador** | **Nivel de medición** | **Unidad de medida** | **Índice** | **Valor** |
| Complejidad computacional | Ordinal | Notación O grande. Una función que depende de n, donde n representa el tamaño de la entrada. O(f(n)) | Un algoritmo es más eficiente que otro si este cuenta con una función f(n) de menor orden en su complejidad computacional en notación O-grande | Se tomara la complejidad computacional como única medida de eficiencia |

**Figura 1.1:** Variable independiente

**Fuente:** [Elaboración propia]

# 1.6. JUSTIFICACIÓN

## 1.6.1. ECONÓMICA

Motores de búsqueda como ser Google, Yahoo! y Bing implementan complejos algoritmos de predicción de texto, dichos algoritmos no se encuentran disponible al público en general al tratarse de código privativo. Existen alternativas como servicios en internet que ofrecen servicios similares para la implementación de predicción de texto que cobran por ello.

Al tratarse de un trabajo de diseño e implementación de un algoritmo no será necesario costos en cuanto a infraestructura se refiere.

## 1.6.2. SOCIAL

En la presente tesis se implementará una estructura de datos para la predicción de texto basada en prefijos en una librería disponible para su descarga en internet. Esta librería será de fácil acceso y uso en proyectos de software. Al mantener la eficiencia podrá ser una buena alternativa a servicios similares.

## 1.6.3. CIENTÍFICA

Se trabajará con estructura de datos poco conocidas como ser Treap y Trie. Se mostrará cómo realizar modificaciones en un Treap para realizar operaciones en intervalos con atributos compuestos. Además, se propondrá un nuevo enfoque para lo construcción de este, en tiempo lineal. Las ideas que se propondrán en este trabajo podrán ser de gran ayuda a trabajos similares que utilicen esta estructura de datos.

# 1.7. ALCANCES Y LÍMITES

## 1.7.1. ALCANCES

* Se diseñará una estructura de datos eficiente para la predicción de escritura la cual soportará inserción de nuevas cadenas al diccionario de respuestas.
* Se implementará una estructura de datos eficiente para la predicción de escritura la cual soportará inserción de nuevas cadenas al diccionario de respuestas, como una librería de código libre de fácil uso, disponible para descargar de manera online.
* Se propondrá alternativas a operaciones ya existentes de las estructuras de datos con las cuales se trabajará con el objetivo de mejorar la eficiencia.
* Se realizarán pruebas para medir la eficiencia temporal de la estructura de datos propuesta, frente a distintos tamaños de entrada, los resultados de estas pruebas se presentarán en tablas comparativas.

* Se comparará la eficiencia de la estructura de datos con el algoritmo original propuesto por Matani D.

## 1.7.2. LÍMITES

* La solución se basa en la idea existente de (Matani D, 2011) con la utilización de estructuras de datos distintas a la idea original.
* El proyecto se enfoca en la creación de una estructura de datos para resolver el problema planteado, y se limita a crear una librería con la implementación de la estructura y no así una herramienta con interfaz gráfica.
* El fin del proyecto no es crear una herramienta de usuario final, si no mostrar la estructura de datos propuesta y realizar comparaciones con otras estructuras para así determinar su usabilidad en aplicaciones reales.

# 1.8. APORTES

## 1.8.1. APORTE PRÁCTICO

Con el presente trabajo se pretende dar una herramienta para la predicción de texto eficiente aplicable al software, para ser utilizado como una librearía de fácil acceso y uso.

## 1.8.2. APORTE TEÓRICO

En el presente trabajo se utilizará las estructuras de datos Trie y Árbol Cartesiano(Treap) y se realizará una comparación de la eficiencia de dichas estructuras frente a otras estructuras de propósito similar, dicha comparación de eficiencia podrá ser utilizada por otros trabajos similares.

# 1.9. METODOLOGÍA

La metodología a aplicar en el presente trabajo será el método lógico inductivo, ya que se partirá de casos particulares que nos llevarán a un conocimiento más general.

En este trabajo se partirá de algoritmos y estructuras de datos ya existentes que resuelven problemas particulares, los cuales serán combinados en una sola estructura de datos para resolver el problema de predicción de texto que soporte actualizaciones.

# CAPÍTULO II

# MARCO TEÓRICO

# 2.1. CONCEPTOS BÁSICOS

## 2.1.1 ALFABETO

Un *alfabeto* es un conjunto finito, no vacío, de símbolos indivisibles u objetos atómicos. (Torrico L., 2008).

**Ejemplos:**

Alfabeto de dígitos decimales T= {0,1,2,3,4,5,6,7,8,9}

Alfabeto de las letras L = {a, b, …, z, A, …, Z}

2.1.2 SÍMBOLO

Un *símbolo* es un componente elemental del alfabeto. Por componente elemental entenderemos cualquier elemento u objeto atómico considerado como indivisible. En los ejemplos de arriba el **3** o la **z** son elementos y son los casos más usuales, sin embargo, también consideraremos como indivisible y como un único elemento, es decir, como un símbolo de un alfabeto a objetos tales como pares ordenados u otros. (Torrico L., 2008).

Por ejemplo, aunque no son usuales, no debe extrañarnos encontrar alfabetos así: T= {a1, a2, a3, a4, a5} de cinco símbolos, uno de ellos es **a3**. N = {(A,B), (A,A), (B,A)} de tres símbolos, uno de ellos es **(A,B)**. V = {cadenaquefungecomosímbolo1, cadenaquefungecomosímbolo2} de dos símbolos, uno de ellos es **cadenaquefungecomosímbolo2**. (Torrico L., 2008).

2.1.3 CADENA

Una *cadena* **u** es una secuencia ordenada de longitud finita de símbolos de un alfabeto T. Se dice que la cadena está formada sobre T y a menudo el alfabeto no se indica pues se sobreentiende. Se representa como **u**=a1a2…an donde a1,a2,...,an T.

El símbolo ai 1≤ i ≤ n, ocurre en la posición i de la cadena leída de izquierda a derecha.

La *cadena vacía*, la cual se denota por el símbolo (lambda), es una palabra sobre cualquier alfabeto y se define como una secuencia vacía de símbolos. (Torrico L., 2008).

Ejemplos:

Las siguientes son cadenas sobre el alfabeto T = {a, b}

u1 = abab

u2 = bbbb

u3 = abbbaaa

u4 =

## 2.1.4 OPERACIONES CON CADENAS

Sean tres cadenas sobre el alfabeto T **u**=c1c2…ck , **w**=a1a2…an y **z**=b1b2…bm. (Torrico L., 2008).

### 2.1.4.1 LONGITUD

La longitud de **w** se denota mediante el símbolo |**w**| y se define como el número de símbolos que tiene la cadena: |**w**|=n La cadena vacía es una secuencia de 0 símbolos. (Torrico L., 2008).

Ejemplos:

Si **w**=0101, |**w**|= |0101| = 4

Si **w**=, |**w|** = = 0

### 

### 2.1.4.2 IGUALDAD DE CADENAS

Se dice que **w** es igual a **z**, si tienen la misma longitud y los mismos símbolos en idénticas posiciones. Se denota mediante **w**=**z**. (Torrico L., 2008).

**w**=**z** si se cumple que |**w**| = |**z**| y

### 2.1.4.3 CONCATENACIÓN DE CADENAS

La concatenación de **w** con **z** es la cadena que se obtiene al añadir a la derecha de la cadena **w** la cadena **z**, y se denota por **wz** (o **w.z**, aunque a menudo el punto se omite). Es decir, consiste de todos los símbolos de **w** seguidos por los símbolos de **z**. **wz** =a1a2…anb1b2…bm. (Torrico L., 2008).

### 2.1.4.4 PREFIJO

Sean **u**, **w** y **z** tres cadenas sobre el alfabeto T. Se dice que la cadena **z** es prefijo de **w**, si para alguna cadena **u** se obtiene **w** = **zu**. La cadena vacía es prefijo de cualquier palabra. Toda palabra puede considerarse prefijo de sí misma.

El prefijo **z** de la cadena **w** se denomina prefijo propio si **z** ≠**w**. (Torrico L., 2008).

# 2.2. ÁRBOL DE PREFIJOS TRIE

Un árbol de prefijos o también llamado Trie es la herramienta básica para estructuras de datos con cadenas, similar al rol que cumplen los arboles balanceados de búsqueda binaria. Esta estructura fue inventada por Briandais(1959). (Brass P., 2008).

El nombre "trie" es un juego de palabras presentado por E. Fredkin en 1960 por que la estructura de datos es usada para "*retrieval*" (recuperar), pero se pronuncia como *"try"* para evitar confusiones con *"tree".* (Sedgewick R., Wayne K., 2011).

## 2.2.1. CONCEPTOS BÁSICOS

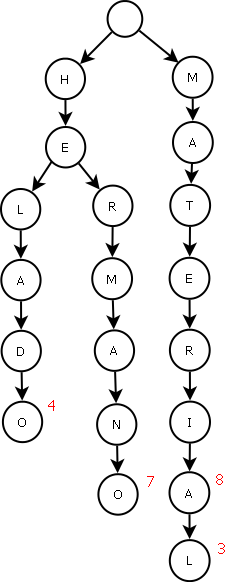
Un trie es un árbol con raíz que almacena o representa un conjunto de cadenas. Cada cadena en el conjunto es almacenada como una cadena de caracteres que inician en la raíz del árbol. Si dos cadenas tienen un prefijo en común, también tienen una cadena de caracteres en común en el árbol. (Laaksonen A., 2018)

Al igual que con los árboles de búsqueda, un trie es una estructura de datos compuesta por nodos que contienen enlaces hacia una referencia nula u otros nodos. Cada nodo es señalado por un solo nodo, el cual es llamado padre, excepto por el nodo raíz el cual no tiene ningún nodo que lo apunte. Cada nodo tiene una cantidad de enlaces que es representada por el tamaño del alfabeto, a menudo los tries tienen un número sustancial de enlaces nulos por lo que al momento de dibujar un trie se omite dichos enlaces. Cada enlace corresponde a un carácter, ya que cada enlace apunta exactamente a un nodo etiquetamos cada nodo con el valor del carácter correspondiente al enlace que apunta, excepto la raíz, la cual no tiene ningún enlace que apunta a ella (Sedgewick R., Wayne K., 2011).

Denominaremos clave a cada cadena del conjunto de cadenas que representa el trie.

Cada nodo también tiene un valor correspondiente, el cual pude ser nulo o el valor asociado con una de las cadenas del conjunto. (Sedgewick R., Wayne K., 2011).

En la figura 2.1 se muestra un ejemplo de un trie que almacena el siguiente conjunto de pares clave, valor T = {(“HELADO”, 4), (“HERMANO”, 7), (“MATERIA”, 8), (“MATERIAL”, 3)}



**Figura 2.1:** Trie para las cadenas {“HELADO”, “HERMANO”, “MATERIA”, “MATERIAL”} y sus respectivos valores asociados.

**Fuente:** [Elaboración propia]

Como se puede apreciar en la figura 2.1 las cadenas “HELADO” y “HERMANO” comparten el prefijo “HE” y en el trie, los nodos correspondientes a este prefijo son creados una sola vez. Lo mismo sucede con las cadenas “MATERIA” y “MATERIAL” que comparten el prefijo “MATERIA”.

## 2.2.2. BÚSQUEDA EN UN TRIE

Para realizar una operación de búsqueda en un trie, iniciamos en la raíz del árbol y leemos la cadena consultada de izquierda a derecha, siguiendo por cada carácter de la cadena el puntero correspondiente a ese carácter al siguiente nodo. Después que terminamos de leer la cadena, se llega al nodo correspondiente. (Brass P., 2008).

Durante la realización de este procedimiento podrían ocurrir diferentes situaciones:

* **Se completa el recorrido** satisfactoriamente y el **valor asociado al nodo es diferente de nulo**, este es el único caso de **éxito**, se retorna el valor asociado.
* **Se completa el recorrido** satisfactoriamente y el **valor asociado al nodo es nulo**, en este caso la cadena consultada existe como prefijo de alguna cadena de conjunto de cadenas, pero la misma cadena no pertenece a dicho conjunto, se retorna nulo para representar que la cadena no se encuentra en el conjunto.
* **No se completa el recorrido**, esto ocurre porque en algún momento del recorrido el nodo correspondiente al carácter actual por el cual debe seguir el recorrido se encuentra con valor nulo, es decir el siguiente nodo no existe. En este caso se retorna nulo para representar que la cadena no se encuentra en el conjunto.

## 2.2.3 INSERCIÓN EN UN TRIE

Como con los arboles binarios, insertamos primero haciendo una búsqueda: en un trie eso significa usando los caracteres de la cadena para seguir hacia abajo en el trie hasta llegar al último carácter de la cadena o a un enlace nulo. En este punto una de las dos siguientes condiciones se cumple:

* Encontramos un enlace nulo entes de llegar al último carácter de la cadena(cadena). En este caso no existe un nodo correspondiente al último carácter de la cadena, así que necesitamos crear nodos por cada carácter en la cadena que aún no visitamos y asignar al valor del ultimo nodo el valor asociado con la cadena.
* Encontramos el ultimo carácter de la cadena antes de llegar a un enlace nulo. En este caso asignamos al valor del nodo el valor asociado con la cadena (este o no este valor en nulo). (Sedgewick R., Wayne K., 2011).

## 2.2.4 IMPLEMENTACIÓN DE UN TRIE

A continuación, se presenta una implementación en el lenguaje de programación JAVA de la estructura Trie, muy similar a la presentada en (Sedgewick R., Wayne K., 2011).

### 2.2.4.1 LA CLASE TRIE Y REPRESENTACIÓN DE UN NODO

Dentro de la clase Trie se guarda un enlace a la raíz del trie, el cual será utilizado para la implementación de los métodos correspondientes al trie.

También es necesaria la creación de la clase Nodo donde se almacena el valor asociado a dicho nodo y los enlaces a los nodos hijos.

**public** **class** Trie {

**private** **static** **int** *R* = 256; // Tamaño del alfabeto

**private** Nodo raiz; // raiz del trie

**private** **static** **class** Nodo {

**private** Valor val;//valor asociado al nodo

**private** Nodo[] sig = **new** Nodo[*R*];

}

//.. Metodos: buscar, insertar, eliminar, etc.

}

**Figura 2.2:** Clase Trie y Clase Nodo

**Fuente:** [elaboración propia]

### 2.2.4.2 BÚSQUEDA EN UN TRIE

Para la búsqueda se implementa un método que realiza la búsqueda en el Trie como se describió anteriormente y se implementa otro método que sirve como interfaz para el usuario para facilitar su uso, sin la necesidad de que el usuario se entere del funcionamiento interno del Trie.

Notar que el código presentado en la figura 2.3 corresponde a un método de la clase Trie.

**public** Valor buscar(String key) {

Nodo x = buscar(raiz, key, 0);

**if** (x == **null**) **return** **null**;

**return** x.val;

}

// Retorna el valor asociado a key en el subtrie que tiene como raiz a x.

**private** Nodo buscar(Nodo x, String key, **int** d) {

**if** (x == **null**) **return** **null**;

**if** (d == key.length()) **return** x;

**char** c = key.charAt(d);//Usa el d-esimo caracter para identificar el subtrie.

**return** buscar(x.sig[c], key, d+1);

}

**Figura 2.3:** Búsqueda en un trie

**Fuente:** [elaboración propia]

### 2.2.4.3 INSERCIÓN EN UN TRIE

El código presentado en la figura 2.4 implementa la idea descrita anteriormente. De la misma forma que se realizó en la búsqueda también se cuenta con un método que sirve como interfaz para el usuario, el cual recibe la clave y el valor asociado a dicha clave.

**public** **void** insertar(String key, Valor val)

{ raiz = insertar(raiz, key, val, 0); }

// Cambia el valor asociado a key si esta en el subtrie con raiz en x.

**private** Nodo insertar(Nodo x, String key, Valor val, **int** d) {

**if** (x == **null**) x = **new** Nodo();

**if** (d == key.length()) { x.val = val; **return** x; }

**char** c = key.charAt(d);//Usa el d-esimo caracter para identificar el subtrie.

x.sig[c] = insertar(x.sig[c], key, val, d+1);

**return** x;

}

**Figura 2.4:** Inserción en un trie

**Fuente:** [elaboración propia]

### 2.2.4.4 UTILIZACIÓN

En la figura 2.5 se muestra el código para la utilización del código ya presentado en los apartados anteriores.

**public** **class** Principal {

**public** **static** **void** main(String[] args) {

Trie t = **new** Trie();

//Se inserta la clave "HELADO" con el valor asociado 4

t.insertar("HELADO", **new** Valor(4));

//Se inserta la clave "HERMANO" con el valor asociado 7

t.insertar("HERMANO", **new** Valor(7));

//Consulta la clave "HERMANO"

//Respuesta esperada: "7"

System.*out*.println(t.buscar("HERMANO"));

//Consulta la clave "HERMANA"

//Respuesta esperada: "null" la cadena no existe

System.*out*.println(t.buscar("HERMANA"));

//Se inserta la clave "MATERIAL" con el valor asociado 3

t.insertar("MATERIAL", **new** Valor(3));

//Consulta la clave "MATERIA"

//Respuesta esperada: "null", la palabra se encuentra como

//prefijo de otra cadena, pero no la cadena en si

System.*out*.println(t.buscar("MATERIA"));

//Se inserta la clave "MATERIA" con el valor asociado 8

t.insertar("MATERIA", **new** Valor(8));

//Consulta la clave "MATERIA"

//Respuesta esperada: "8"

System.*out*.println(t.buscar("MATERIA"));

}

}

**Figura 2.5:** Utilización de la implementación presentada del trie

**Fuente:** [elaboración propia]

A continuación, en la figura 2.6 se muestra la salida correspondiente a la ejecución del código presentado en la figura 2.5.

7

null

null

8

**Figura 2.6:** Utilización de la implementación del trie

**Fuente:** [elaboración propia]

## 2.2.5 COMPLEJIDAD

La estructura básica Trie almacena un conjunto de palabras sobre un alfabeto **A**. Soporta una operación de búsqueda de una cadena **q** en tiempo O(longitud(q)), la operación de inserción y eliminación en tiempo O(|A| longitud(q)). El espacio requerido para almacenar n cadenas w1 , ..., wn es . (Brass P., 2008).

# 2.4. ÁRBOL CARTESIANO(TREAP)

Treap es una estructura de datos que combina las propiedades de un árbol binario y un heap, de ahí el nombre tree+heap => Treap.

Mas específicamente, treap es una estructura de datos que almacena pares (X, Y) en un árbol binario de tal manera que es un árbol binario de búsqueda por X y un heap por Y. Asumiendo que todos los valores de X e Y son diferentes. Podemos ver que si algún nodo contiene valores (X0, Y0), todos los nodos en el subárbol izquierdo tienen X < X0, todos los nodos en el subárbol derecho tienen X > X0, y todos los nodos en ambos subárboles izquierdo como derecho tienen Y < Y0. (Polozov A., 2010)

Los treaps fueron propuestos por Siedel y Aragon en 1989. (Polozov A., 2010)

Tomando valores aleatorios para los valores Y de cada nodo el Treap es utilizado como un árbol de búsqueda aleatorio. Esta aleatoriedad hace con una alta probabilidad que la altura de un treap sea proporcional al logaritmo del número de nodos del Treap. Gracias esta propiedad se mantiene la eficiencia en las diferentes operaciones relacionadas al Treap como árbol binario de búsqueda. (Polozov A., 2010)

## 2.4.1 OPERACIONES DE UN TREAP

Un treap al ser un árbol binario de búsqueda cuenta con operaciones relacionadas a este tipo de estructuras como ser inserción, búsqueda y eliminación de datos. Para realizar estas operaciones el Treap se basa en dos operaciones muy particulares de esta estructura, estas son el *split* (dividir) y el *merge* (unión). (Polozov A., 2010)

### 2.4.1.1 OPERACION SPLIT

Esta operación toma de entrada una clave X y separa el árbol T en dos árboles L y R tal que L contiene todos los elementos con clave XL < X, y R contiene todos los elementos con clave XR >= X. Esta operación tiene una complejidad de O(log N) y se implementa mediante una recursión. (Polozov A., 2010)

### 2.4.1.2 OPERACION MERGE

Esta operación toma de entrada dos árboles T1 y T2 y retorna un nuevo árbol. Esta operación también tiene una complejidad O(log N). Trabaja bajo la asunción de que tanto T1 como T2 se encuentra ordenadas (todas las claves de T1 son menores que las claves de T2). Se necesita combinar esos dos árboles sin violar el orden de las prioridades. Para hacer esto, elegimos la raíz con más alta prioridad Y como nuevo nodo raíz y recursivamente llamamos la operación merge para el otro árbol y su correspondiente subárbol del árbol del cual seleccionamos el nodo raíz. (Polozov A., 2010)

### 2.4.1.3 OPERACION DE INSERSION

Una vez definidas las operaciones de *merge* y *split*, la operación de inserciónse realiza de una manera muy sencilla.

Dado un par (X, Y) a ser insertado en el Treap, con Y seleccionado aleatoriamente, se utiliza la operación *split* para dividir el árbol en dos árboles L y R donde L almacena todos los nodos con claves menores a X y R almacena todos los nodos con claves mayores a X. A continuación, se crea un nuevo árbol M teniendo como único nodo al par (X, Y). Finalmente se realiza la operación *merge(merge(L, M), R)* para fusionar los 3 árboles obtenidos*.* Al tener tanto merge como split una complejidad logarítmica, la complejidad de la operación de inserción se mantiene en O(log N). (Polozov A., 2010)

### 2.4.1.4 OPERACION DE ELIMINACIÓN

L a operación de eliminación de una clave X se realiza de una manera muy similar a la de inserción, primero se utiliza la operación merge para dividir el árbol en dos árboles L y TMP, donde L guarda todos los nodos con clave menor a X y tmp todos los nodos restantes.

A continuación, se divide TMP en dos arboles M y R utilizando la clave X+1 para realizar el split.

Finalmente se realiza una llamada a merge(L, R) para obtener el resultado final.

Una vez más al utilizar operaciones que toman tiempo logarítmico la complejidad de esta operación también se mantiene en O(log N). (Polozov A., 2010)

### 2.4.1.5 BUSQUEDA

El Treap al mantener la propiedad de árbol de búsqueda binaria, la búsqueda de datos(claves) se realiza utilizando los mismos métodos que un árbol de búsqueda binaria común y corriente. Al tener con alta probabilidad el Treap una altura proporcional al logaritmo del número de nodos la búsqueda también se puede realizar con una complejidad de O(log N). (Polozov A., 2010)

## 2.4.2 IMPLEMENTACIÓN DE UN TREAP

En el anexo A se muestra una implementación en el lenguaje C++ de la estructura de datos Treap utilizando las ideas descritas anteriormente. Se presenta una implementación muy parecida a la mostrada en (Ivanov M., 2013).

# 2.5 TREAPS IMPLICITOS

Un Treap Implícito también conocida como Rope, es una simple modificación de un Treap el cual es una poderosa estructura de datos. Un treap implícito puede ser considerado como un arreglo(array) con los siguientes métodos implementados (todos con complejidad O(log N)) (Polozov A., 2010)

* Insertar un elemento en cualquier posición
* Remover un elemento en cualquier posición
* Buscar la suma, mínimo/máximo elemento en cualquier intervalo
* Adición en un rango
* Invertir los elementos en un rango

## 2.5.1 IDEA GENERAL

La idea es que las claves de los nodos sean los índices de los elementos en el array. Pero no se almacenarán esos valores explícitamente (de otro modo al insertar un nuevo elemento será necesario una modificación de todas las claves en O(n) nodos del árbol)

Notar que la clave de un nodo es el número de nodos menores a él (esos nodos pueden estar presentes no solo en el subárbol izquierdo, sino también en los subárboles de los ancestros a ese nodo). (Polozov A., 2010)

Para hacer esto posible será necesario almacenar un nuevo valor en cada nodo que indique la cantidad de nodos en el subárbol. Este valor puede ser fácilmente mantenido y actualizado mientras se realizan las operaciones de merge y split sin modificar la complejidad de estas. (Polozov A., 2010)

## 2.5.2 IMPLEMENTACION DE UN TREAP IMPLICITO

En el anexo B se muestra una implementación de un Treap Implícito utilizando el código del Treap presentado anteriormente e implementando la idea mencionada el apartado anterior. Se presenta una implementación muy parecida a la mostrada en (Ivanov M., 2013).

# 2.6 EL ALGORITMO DE MATANI D.

(Matani D., 2011) propone un algoritmo para solucionar el problema de predicción de texto utilizando un array estático de cadenas y una estructura de datos Segment Tree.

## 2.6.1 CONSTRUCCIÓN DE LA ESTRUCTURA

En primera instancia se construye un array con los todos los datos disponibles, a continuación, se procede a ordenar el array lexicográficamente, utilizando las cadenas como medio de comparación. Llamaremos DS a este array ordenado.

Se extrae un nuevo array tomando los pesos de cada dato en DS manteniendo el orden. Se procede a construir una estructura Segment Tree, esta estructura debe ser capaz de responder el elemento máximo en un rango [L, R) y además la posición de este en el intervalo. Llamaremos ST a esta estructura. (Matani D. 2011)

Este proceso de construcción toma una complejidad total igual a donde n es el número de cadenas que almacenara la estructura.

## 2**.6.2 CONSULTAS**

Para responder a una consulta de las K mejores cadenas con prefijo X, utilizando búsqueda binaria se encuentra el máximo rango [A, B) en DS, donde todos los elementos en el rango tienen a X como prefijo.

Con la ayuda de una cola de prioridad se procede a hallar la solución a la consulta. Partiendo del rango [A, B) se consulta en ST el máximo elemento en dicho rango y su posición, formando la tupla (máximo, índice, A, B), este dato es ingresado a la cola de prioridad.

Se va a tomando de una en una los elementos de la cola de prioridad. De aquí en adelante cada elemento que es obtenido de la cola de prioridad es la siguiente solución al problema. Cada vez que se obtiene un elemento de la cola de prioridad se toma como solución el **índice** de dicho elemento y se obtiene hasta 2 nuevos rangos de este rango, con limites [A, índice) y [índice + 1, B), se halla sus máximos en ST y se los inserta en la cola de prioridad, ignorando rangos vacíos. (Matani D. 2011).

En cada iteración del ciclo del algoritmo se obtiene una de las K respuestas necesarias, y cada ciclo toma en el peor caso una complejidad igual a para responder a las consultas del Segment Tree y para realizar la eliminación de elementos de la cola de prioridad. Resultando una complejidad total de para responder a una consulta.

# CAPITULO III

# MARCO APLICATIVO

En el anterior capítulo se pudo observar toda la teoría necesaria para la implementación de la estructura de datos que se propone en la presente tesis de grado. Ahora toca ver la implementación de la estructura, se definirá la idea general tras la estructura, la carga de datos, el funcionamiento de la estructura, y se realizará una comparación con otras estructuras de propósito similar.

# 3.1 ESPECIFICACION DE LA ESTRUCTURA DE DATOS

La estructura de datos que se presenta en la presente tesis de grado almacenara un conjunto de cadenas, donde cada cadena tiene asociado a un valor o peso, en forma de pares (X, Y), X es la cadena e Y es el valor o peso.

Una vez cargados los datos, la estructura de datos será capaz de responder a consultas o actualizaciones:

* **Construcción**: build(filename), la estructura leerá el archivo de texto filename compuesto de pares (X, K) separados por un espacio, donde X es una cadena y K un entero, la estructura insertara la cadena X dentro del Trie asociando el valor K a X. Después de realizar n inserciones se procederá a la construcción del Treap.
* **Consulta: query(X, K),** donde X es una cadena y K un entero, la estructura tiene como salida a esta consulta un conjunto de cadenas, donde el conjunto es de tamaño menor o igual a K, todas las cadenas del conjunto tienen a X como prefijo. Estas K cadenas son una selección de las K mejores (con mayor peso) cadenas de todo el conjunto de cadenas que tienen a X como prefijo en la estructura de datos.
* **Inserción: add(X, Y),** donde X es una cadena e Y un valor entero, la estructura adicionara la cadena X asociado al valor Y dentro del conjunto de cadenas disponibles.
* **Actualización: update(X, Y),** donde X es una cadena e Y un valor entero, la estructura actualizara el valor asociado a X asignándole el valor Y.

# 3.2 DISEÑO DE LA ESTRUCTURA

Al ser el Segment Tree una estructura de datos que no soporta cambio de dimensión, no permite agregar nuevos elementos sin que sea necesario reconstruir toda la estructura.

Para subsanar este problema se remplazará el Segment Tree por un Treap Implícito, el cual nos permitirá realizar todas las operaciones disponibles en el Segment Tree y además esta estructura permite realizar inserciones y eliminaciones en cualquier posición del array asociado. Sin embargo, la principal desventaja del Treap Implícito frente al Segment Tree es el tiempo de ejecución experimental.

También se remplazará el array estático ordenado de la idea original por una estructura de datos Trie permitirá realizar las mismas consultas que se realiza a DS de una manera teóricamente más eficiente.

## 3.2.1 UN TRIE QUE RESPONDE POSICIONES

Para responder a las consultas de tipo **add**, es necesario contar con una estructura de datos que permita almacenar todo el diccionario de respuestas disponibles, además de poder hacer consultas sobre la existencia o no de alguna cadena. Para ello se tomará de base la estructura de datos Trie, ya que esta estructura cuenta con las siguientes ventajas:

* Mantiene una estructura de árbol, el cual permite un fácil recorrido.
* Hace uso eficiente de la memoria

Se adiciona dos nuevos métodos a la estructura Trie para que este sea capaz de responder a dos nuevos tipos de consultas:

* Posición de una cadena en el Trie. Dada una cadena S, el Trie debe ser capaz de responder la posición que le correspondería a la cadena S en el conjunto lexicográficamente ordenado de todas las cadenas que almacena el Trie.
* Rango de dado un prefijo. Dada una cadena P, el Trie debe ser capaz de responder las posiciones de todas las cadenas almacenadas en el Trie que tienen a P como prefijo. Al representar el Trie un conjunto lexicográficamente ordenado de cadenas, estas posiciones representaran un intervalo de posiciones continuo [X, Y) en dicho conjunto.

Se implementa dichas consultas tomando en cuenta la eficiencia en el tiempo de respuesta.

Se tomará como base una implementación de Trie, en la cual cada nodo del Trie mantiene un array de punteros a los hijos directos del nodo, como se observa en la figura 3.1.

struct trie {

const static int R = 26;//tamaño alfabeto(A - Z)

const static int nullvalue = INT\_MAX;

struct nodo {

int val, cnt;

nodo\* sig[R];

nodo() : val(nullvalue), cnt(0) {

for (int i = 0; i < R; i++) sig[i] = nullptr;

}

};

nodo\* raiz = nullptr;

trie () {

raiz = new nodo();

}

}

**Figura 3.1:** Implementación de la estructura Trie con la utilización de punteros

Del código mostrado en la figura 3.1 se puede obtener una complejidad computacional igual a para la creación de un nodo del Trie. Donde representa el tamaño del alfabeto.

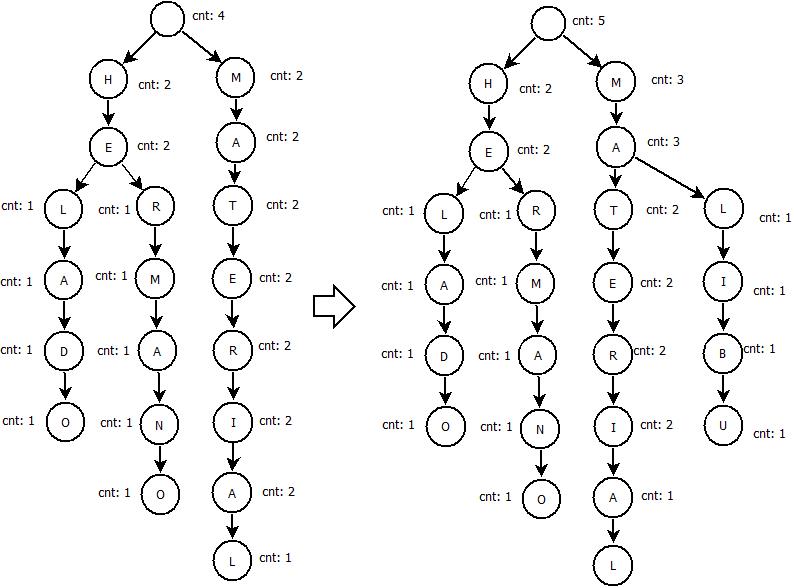
### 3.2.1.1 POSICION DE UNA CADENA EN EL TRIE

Para responder a esta consulta se agregará un nuevo dato a cada nodo del Trie. Este valor será llamado cnt. Dado un nodo, cnt representa el número de nodos terminales a los cuales se puede llegar partiendo en dicho nodo. Este nuevo valor será de mucha utilidad para poder responder a los diferentes tipos de consultas que se propone en la presente tesis.

Para mantener actualizado el valor de cnt en cada nodo del Trie, se actualizará su valor cada vez que se ingrese una nueva cadena en el Trie. Se debe tener cuidado en este aspecto ya que ingresar una cadena que ya existe en el trie modificara erróneamente los valores de cnt.

Este proceso mantiene la eficiencia porque solo es necesario modificar un numero de nodos igual al tamaño de la cadena que se ingresa, resultando una complejidad computacional de O(r), donde r es el número de caracteres en la cadena a insertar.

En la figura 3.2 se observa como solo es necesario modificar los nodos que representan el recorrido de la raíz al nodo terminal de dicha cadena



**Figura 3.2:** Inserción de un nuevo elemento al Trie actualizando el valor de cnt

A continuación, se muestra el código de para realizar dicho proceso, como se puede observar el cambio es trivial:

void insertar(string key, int val) {

nodo\* x = raiz;

for (int d = 0; d < (int)key.length(); d++) {

int c = key[d];

if (x->sig[c] == nullptr) {x->sig[c] = new nodo();}

x = x->sig[c];

**x->cnt++;**

}

x->val = val;

}

**Figura 3.3:** Código inserción de un nuevo dato en el Trie actualizando el valor de cnt

Para realizar la consulta es necesario recorrer solo los nodos que corresponden a la cadena consultada y realizar un pequeño proceso en cada nodo donde se cuenta la cantidad de nodos terminales de los cuales se realiza el salto, es decir la cantidad de nodos terminales menores lexicográficamente a los que se es posible llegar partiendo en dicho nodo, esto se hace revisando todos los posibles caminos menores lexicográficamente y sumando el valor de cnt.

int posicion(string key) {

nodo\* x = raiz;

int cnt = 0;

for (int d = 0; d < (int)key.length(); d++) {

int c = key[d] - 'A';

**for (int i = 0; i < c; i++)**

**if (x->sig[i] != nullptr)**

**cnt += x->sig[i]->cnt;**

x = x->sig[c];

**if (x->val != nullvalue)**

**cnt++;**

}

return cnt-1;

}

**Figura 3.4:** Consulta de posición de una cadena en el trie

#### 3.2.1.1.1 COMPLEJIDAD COMPUTACIONAL

Para el caso de inserción de una nueva cadena al Trie, de la figura 3.3 se puede obtener una complejidad computacional igual a en el peor caso, donde representa el tamaño de la cadena a insertar y el tamaño del alfabeto.

Del código mostrado en la figura 3.4 se puede obtener una complejidad computacional igual a en el peor caso, donde L es igual a la longitud de la cadena a consultar y R al tamaño del alfabeto, para la obtención de la posición de una cadena en el Trie.

Se menciona el peor caso ya que no siempre se da este, pero se asume este para medir la complejidad computacional en la notación O-grande.

### 3.2.1.2 RANGO DE POSICIONES DADO UN PREFIJO

Dado una cadena prefix el trie responderá un par de números [A,B) que representa el intervalo de elementos en el Trie si este fuera representado como un conjunto de cadenas ordenado lexicográficamente, donde todos los elementos en dicho rango tienen a prefix como prefijo.

Pare responder este nuevo tipo de consulta se procederá de una manera similar a la consulta de posicion ya que el límite inferior de este rango está acotado por la primera cadena en el conjunto que tenga a prefix como prefijo, el cual es equivalente a encontrar la posición en la que se encontraría la cadena prefix en el conjunto. Con esto en mente A tomara el valor de cnt que se mostró en el algoritmo para hallar la posicion.

Se llamará x al último nodo en el recorrido para encontrar el valor de A. El nodo x almacena en su valor cnt todas las cadenas en el Trie que tienen prefijo igual a prefix, entonces el valor de B estará dado por A + x.cnt.

En la figura 3.5 se puede observar el código del proceso descrito anteriormente, cabe considerar el caso en el que no exista ninguna cadena almacenada en el Trie que tenga a prefix como prefijo, en ese caso se debe devolver un intervalo vacío, en la figura se observa que se devuelve el intervalo vacío [-1,-1).

range query(const string &prefix) const {

nodo\* x = raiz;

int cnt = 0;

for (int d = 0; d < (int)prefix.length(); d++) {

int c = prefix[d] - 'A';

//El prefijo no existe en el trie

if (x->sig[c] == nullptr)

return make\_pair(-1, -1);

//Contamos la cantidad de cadenas

//lexicograficamente menores

**for (int i = 0; i < c; i++)**

**if (x->sig[i] != nullptr)**

**cnt += x->sig[i]->cnt;**

**if (x->val != nullvalue)**

**cnt++;**

x = x->sig[c];

}

return make\_pair(cnt, cnt + x->cnt);

}

**Figura 3.5:** Consulta de rango de posiciones con prefijo igual a prefix

#### 3.2.1.2.1 COMPLEJIDAD COMPUTACIONAL

De la figura 3.5 se obtiene una complejidad computacional igual a , donde L es una constante igual a la longitud de la cadena de la consulta y R es una constante igual al tamaño del alfabeto.

## 3.2.2 CADENA EN EL TRIE DADA UNA POSICIÓN

Dado un entero i, el Trie devolverá la i-esima cadena en el conjunto lexicográficamente ordenado de cadenas que almacena.

Una vez más se partirá de una idea similar a la del algoritmo para encontrar la posición de una cadena en el trie. Pero en este caso no se cuenta con una cadena de la cual seguir el camino de la raíz al nodo hoja en el trie.

La estrategia que se seguirá será tratar de construir todas las cadenas que almacena el Trie de forma ordenada, pero sin construir todas explícitamente, sino saltar aquellas que claramente no conducen a la solución,

Dado nodo X del cual se desea buscar la i-esima cadena relativa a X, se tienen R posibles caminos hacia los hijos de X, sean estos Y(0), Y(1), Y(2), …, Y(R-1), donde R es el tamaño del alfabeto, se debe buscar la forma de elegir uno de los R caminos, el cual conducirá a la solución deseada.

Si Y(s) es el camino que contiene la solución, entonces todos los caminos menores a s, Y(0), Y(1), …, Y(s-1) conducen a cadenas lexicográficamente menores a la i-esima cadena, y todos los caminos mayores a s, Y(s+1), Y(s+2), …, Y(R) conducen a cadenas lexicográficamente mayores.

Como Y(idx).cnt almacena el número de cadenas que representa el subárbol Y(idx), se debería cumplir que Y(0).cnt + Y(1).cnt + … + Y(s-1).cnt < i, ya que si no fuera así se estaría seleccionando un camino s que conduce a una cadena con índice mayor a i, si esto sucediera se podría ir reduciendo el valor de s hasta encontrar un valor que si cumpla la propiedad. O visto desde el otro lado, partiendo de s = 0 se podría ir incrementando el valor de s hasta encontrar el primer camino que cumpla la propiedad.

Una vez encontrado el valor de s, se conoce el carácter que se seleccionó para seguir ese camino en el Trie, ahora X tomara el valor de Y(s), pero se debe actualizar el valor i ya que este se encuentra relativo a X, simplemente se le debe restar el valor de Y(0).cnt + Y(1).cnt + … + Y(s-1).cnt a i ya que ese valor representa el número de cadenas lexicográficamente menores que se está saltando al seguir el camino Y(s). Al realizar estos saltos también se debe considerar el caso especial en el que X también representa un nodo terminal en el Trie, ya que también se está saltando esa cadena extra.

En el instante en el que se cumple i = 0 nos encontramos en el nodo terminal de la i-esima cadena.

En la figura 3.6 se implementa la idea descrita en este apartado, con la excepción que en lugar de ir reduciendo el valor i, se utiliza otra variable cnt que va contando los saltos que se realizaron en el proceso, en el instante que i == cnt, termina el algoritmo. La respuesta se va construyendo en la variable keyword.

También se debe tener en cuenta el caso en el el que el valor de i sobrepasé el número de elementos que almacena el trie, en dicho caso no existe solución.

phrase get(int i) {

i++;

int cnt = 0;

string keyword;

nodo\* x = raiz;

bool found = false;

while (!found) {

bool path\_found = false;

for (int c = 0; c < R and !path\_found; c++) {

if (x->sig[c] != nullptr) {

int s = x->sig[c]->cnt;

if (cnt + s < i) {

cnt += s;

} else {

keyword += char('A' + c);

x = x->sig[c];

path\_found = true;

}

}

}

if (path\_found) {

if (x->val != nullvalue)

cnt++;

if (cnt == i)

found = true;

} else {

return phrase("Not found", -1);

}

}

return phrase(keyword, x->val);

}

**Figura 3.6:** Código para obtener la i-esima cadena del Trie

### 3.2.2.1 COMPLEJIDAD COMPUTACIONAL

A diferencia de las operaciones para el Trie analizadas previamente, el código mostrado en la figura 3.6 es un poco más complejo de analizar.

Es fácil notar que el código dentro del ciclo while se ejecutara con una complejidad computacional igual a O(R) en el peor caso, donde R es el tamaño del alfabeto.

En cada iteración del ciclo while, a lo mucho el nodo X avanzará una posición hacia abajo en la profundidad del Trie. Dando una complejidad igual a O(L) para el ciclo while, donde L representa la constante igual la longitud de la cadena más larga almacenada por el Trie en el peor caso.

Para finalmente obtener una complejidad total igual a O(L\*R), donde L representa la longitud de la cadena más larga en el Trie y R el tamaño del alfabeto en el peor caso.

## 3.2.3 TREAP IMPLICITO CON RESPUESTA EN RANGOS

Usualmente realizar modificaciones en un Treap Implícito para responder consultas es relativamente fácil, ya que la mayoría de las funciones de este no se modifican en absoluto.

En este apartado se realizarán modificaciones al Treap para que sea capaz de responder el máximo elemento en un rango además de su posición.

### 3.2.3.1 MAXIMO EN UN RANGO EN UN TREAP IMPLICITO

Dado un intervalo [a,b) el Treap Implícito devolverá un par (value, index), donde value es el valor del elemento con máximo valor en intervalo [a,b), e index su posición. Si es que existiesen varios elementos con valor igual al máximo en el intervalo, se devolverá máximo valor posible para index.

Para responder a este tipo de consultas es necesario guardar un valor extra en cada nodo del Treap, llamaremos maxi a este nuevo valor, maxi almacenara un par (value, index).

Dado un nodo X, maxi.value almacena el valor del máximo elemento en todo el subárbol que tiene a X como padre, y maxi.index la posición relativa a X de maxi.value si el este subárbol fuera representando como un array.

Para mantener actualizado el valor de maxi en las diferentes operaciones que se realice al Treap, se modificara el método recalc() de los nodos del Treap, ya que este método es llamado cada vez que el nodo necesita ser actualizado por que sus hijos sufrieron modificaciones.

Cada vez que el método recalc() es llamado se toma en cuenta tres posibles valores para maxi en el nodo X:

* El nodo actual contiene el valor máximo: En este caso maxi.value toma el valor del valor que almacena el nodo. El valor de maxi.index relativo al nodo X será igual al número de nodos en el subárbol izquierdo, ya que este valor representa el número de elementos en posición menor a X.
* El subárbol izquierdo contiene el valor máximo: En este caso los valores de maxi.value y maxi.index relativos al subárbol izquierdo de X, también son relativos al nodo X.
* El subárbol derecho contiene el valor máximo: maxi.value toma el valor de maxi.value del subárbol derecho y maxi.index tomar un valor igual a el número de elementos del subarbol izquierdo + 1 + maxi.index del subarbol derecho.

En la figura 3.7 se muestra la modificación al método recalc() y la aplicación de los tres casos mostrados. Se debe notar que los casos solos son aplicables si dichos nodos existen.

typedef struct \_node {

int value, y, cnt;

pair<int,int> maxi;//(value, index)

\_node \*l, \*r;

\_node(int \_value) : value(\_value), y((rand() << 16) ^ rand()), cnt(1), maxi(\_value, 0), l(nullptr), r(nullptr) {}

~\_node() {delete l; delete r;}

void recalc() {

cnt = 1;

maxi = make\_pair(value, l?l->cnt:0);

if (l) {

cnt += l->cnt;

maxi = max(maxi, l->maxi);

}

if (r) {

cnt += r->cnt;

maxi = max(maxi, make\_pair(r->maxi.first, (l?l->cnt:0) + 1 + r->maxi.second));

}

}

} \*node;

**Figura 3.7:** Código Nodo del Treap implícito para hallar máximos en rango

El método para retornar el maximo en el rango [a,b) es similar a otras operaciones clásicas en un Treap. Se divide el Treap que representa al intervalo [0, n) en tres intervalos [0, a), [a, b), [b, n) utilizando el método Split() que tiene una complejidad computacional igual O(log(n)). A continuación se toma el valor maxi del nodo que representa el rango [a,b) y para obtener el índice de dicho máximo se obtiene el valor index del nodo que representa [a,b) y a este valor se le adiciona el valor ”a” teniendo de esta manera el valor (maxi, index) relativo a todo el árbol en el rango [a,b). En la figura 3.8 se puede ver el código del método descrito.

pair<int,int> maxInRange(int a, int b) {//el rango es [a, b)

node L, m, R;

split(root, a, L, R);

split(R, b - a, m, R);

pair<int,int> ret = m->maxi;

ret.second += a;

root = merge(merge(L, m), R);

return ret;

}

**Figura 3.8:** Código Nodo del Treap implícito para hallar máximos en rango

#### 3.2.3.1.1 COMPLEJIDAD COMPUTACIONAL

Como se mencionó en el capítulo anterior todas las operaciones en un Treap se realizan con una complejidad aproximada de donde n es el número de elementos que almacena el Treap.

La función maxInRange mostrada en la figura 3.8 utiliza operaciones básicas en Treaps, obteniendo esta una complejidad computacional igual .

## 3.2.4 ALTERNATIVA A CONSTRUCCIÓN EN TIEMPO LINEAL DE UN TREAP IMPLICITO

Dado un array de N elementos con los cuales se desea construir un Treap Implicito, una forma natural y valida de construir el Treap es recorrer el array de izquierda a derecha e ir insertando cada elemento de a uno dentro del Treap.

Una desventaja de este método es que cada inserción tiene una complejidad computacional de O (log N), dándonos una complejidad total de O(N log N).

### 3.2.4.1 CONSTRUCCION DEL ARBOL BINARIO

Dado que se cuenta con los elementos para construir el Treap de antemano, es posible seguir una idea similar a la construcción de Segment Trees para construir el Treap, de esta manera no solo será posible construir el Treap en tiempo lineal, sino que también el árbol resultante será un árbol binario balanceado, lo cual garantiza una complejidad determinística de O(log N) para las consultas, y no así una complejidad no-determinística aproximada a O(log N) que se da en el caso en el que el Treap es modificado por los métodos insert y delete, ya que en estos interviene la aleatoriedad.

Tomando como base el intervalo que representa a todos los elementos del array del cual se desea construir el Treap [0, N). Dado un intervalo [a,b), se calcula el elemento que se encuentra en medio de dicho intervalo, en la posición m = (a+b)/2, a continuación se divide el intervalo en 3 partes, L = [a, m), M = [m, m+1), R = [m+1, b), se llama recursivamente a la función para construir el Treap para los intervalos L y R, una vez obtenido los dos resultados, se construye un nodo con el único elemento en el intervalo M, y se le asigna como hijo izquierdo y derecho la respuesta de las llamadas a las funciones para L y R respectivamente, al final se recalcula la información extra que tenga que actualizar dicho nodo ahora que tiene acceso a sus nodos hijos. En la figura 3.9 se muestra una implementación de lo descrito aquí.

void linear\_build(const vector<int> &v) {

int n = v.size();

root = \_build(0, n, v);

}

node \_build(int a, int b, const vector<int> &v) {

if (a == b) return nullptr;

int mid = (a + b) / 2;

node m = new \_node(v[mid]);

m->l = \_build(a, mid, v);

m->r = \_build(mid + 1, b, v);

m->recalc();

return m;

}

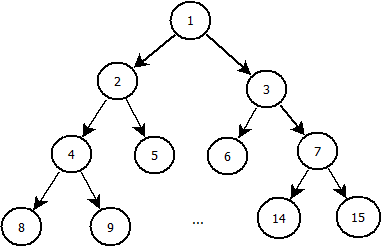
**Figura 3.9:** Código construcción del árbol binario para un Treap

El Treap se caracteriza por cumplir las propiedades de un árbol binario de búsqueda y un heap. Con el procedimiento descrito se logra cumplir la primera propiedad, pero los valores de las prioridades de cada nodo aún no se encuentran establecidos. Es decir que aún no puede se puede utilizar la estructura construida como un Treap.

### 3.2.4.2 ASIGNACIÓN DE PRIORIDADES

Se desea asignar prioridades a cada nodo de tal forma que cumpla la propiedad de Heap, es decir dado un nodo X, sus hijos L y R, hijo izquierdo e hijo derecho respectivamente, se debe cumplir que X.prioridad > L.prioridad y X.prioridad > R.prioridad.

Se partirá del método utilizado para implementar Segment Trees utilizando arrays estáticos, donde se etiqueta al nodo raíz con un id = 1, desde ese punto los hijos izquierdos tendrán una etiqueta igual a 2 \* id(padre), y el hijo derecho será igual a 2 \*id(padre)+1, obteniendo un id único para cada nodo, como se puede ver en la figura 3.10.



**Figura 3.10:** Asignación de ids a un árbol binario, utilizando 2\*id y 2\*id+1

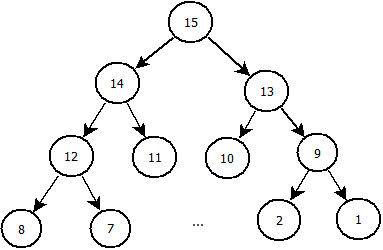
En este punto estos valores de id no son de gran utilidad ya que no cumplen con la propiedad buscada. Se procederá a buscar una forma de invertir los números para que los valores pequeños sean los más grandes y viceversa.

Para conseguir el objetivo deseado se buscará un numero lo suficientemente grande al cual llamaremos N para luego realizar la operación N-id en cada nodo.

El valor de N tiene que ser mayor que el máximo id dentro del árbol para que no se generen números negativos y se mantenga el orden.

Un valor fácil de calcular sin tener antes que construir todo el árbol y buscar el valor del máximo id, es la siguiente potencia de 2 del número de elementos del árbol. Se escoge este número ya que se puede demostrar que un árbol binario con esa cantidad de elementos sería un árbol binario completo y tendría la misma cantidad de niveles que el árbol que se acaba de formar.

Para el ejemplo de la figura 3.10 el valor buscado seria 16, realizando la operación mencionada se obtendría:



**Figura 3.11:** Inversión de íds asignados al árbol binario

Estos valores cumplen con la propiedad de Heap, pero debemos considerar que el árbol sufrirá modificaciones, ya sean inserciones o eliminaciones de nodos, al tener los valores de las prioridades tan cercanos unos de otro cabe la posibilidad que los nuevos nodos que sean insertados al Treap sean insertados fuera de este grupo compacto y de a poco ir degenerando el árbol. Para subsanar este problema multiplicaremos a cada valor por un número relativamente grande para que se alejen unos de otros, al multiplicar todos los valores por un mismo valor, estos mantienen su orden. Se debe tener cuidado al elegir este valor ya que se podría provocar overflow si es que se escoge un numero demasiado grande dependiendo del lenguaje de programación en el cual se implemente.

En la figura 3.12 se aprecia la implementación de esta idea al mismo tiempo que se construye el árbol binario.

void linear\_build(const vector<int> &v) {

int n = v.size();

int next\_pow\_2 = 1;

while (next\_pow\_2 <= n) next\_pow\_2 \*= 2;

int jump = max(INT\_MAX/2, n) / next\_pow\_2;

root = \_build(0, n, v, next\_pow\_2, 1, jump);

}

node \_build(int a, int b, const vector<int> &v, int N, int id, int jump) {

if (a == b) return nullptr;

int mid = (a + b) / 2;

node m = new \_node(v[mid]);

m->l = \_build(a, mid, v, N, 2 \* id, jump);

m->r = \_build(mid + 1, b, v, N, 2 \* id + 1, jump);

**m->y = (N - id) \* jump;**

m->recalc();

return m;

}

**Figura 3.12:** Código para la construcción en tiempo lineal de un Treap

### 3.2.4.3 COMPLEJIDAD COMPUTACIONAL

Se procede a realizar el análisis de complejidad del algoritmo \_build mostrado en la figura 3.12.

Dado que el algoritmo procesa una instancia de tamaño n en cada llamada, donde n es el tamaño del intervalo que representa [a, b), en cada llamada se divide el intervalo en 3 partes donde 1 de las partes es una constante, en cambio las otras dos partes con un tamaño de instancia aproximado a n/2 pasan a ser procesados recursivamente, mientras el resto de las operaciones dentro del procedimiento son funciones constantes.

De este análisis se puede deducir la siguiente ecuación de recurrencia.

La cual se puede resolver utilizando el teorema master. Emparejando las constantes a, b y k para el teorema master a:

Las cuales cumplen la propiedad del teorema master:

Remplazando valores se obtiene una complejidad computacional de para el método de construcción del Treap propuesto.

## 3.2.5 COMBINANDO LAS ESTRUCTURAS - LA ESTRUCTURA SUGGEST

Como ya se mencionó la idea original de Matani D. sobre el algoritmo se mantendrá. El array estático que almacena las cadenas será remplazado por una estructura de datos Trie con las operaciones básicas y las descritas anteriormente. El Segment Tree será remplazado por la estructura de datos Treap Implícito con las operaciones básicas y las descritas en apartados anteriores.

### 3.2.5.1 ESTRUCTURA BASE

La estructura final constara de 2 atributos, un trie y un treap.

struct Suggest {

trie ds;

treap st;

//Rest of code ...

}

**Figura 3.12:** Estructura base para la estructura Suggest

El Trie nos permitirá insertar nuevas cadenas al conjunto. Además de poder realizar operaciones consultas y actualizaciones necesarias para la estructura final, como se describió en los apartados anteriores.

El Treap implícito nos permitirá realizar las mismas consultas que son posibles realizar con el Segment Tree, además de poder insertar nuevos valores a la estructura en cualquier posición si este fuera representado como un array.

### 3.2.5.2 CARGA DE DATOS Y CONSTRUCCIÓN

En el caso en el que se cuente ya con un conjunto de datos con el cual se pretende construir la estructura de datos, por ejemplo, un archivo de texto, es posible construir la estructura utilizando solo operaciones de tipo adición.

Una alternativa que mejora la eficiencia de esta operación es utilizar el método de construcción lineal para el Treap, como ya se mencionó este ofrece una complejidad computacional de O(n).

Para ello se desarrollaron dos métodos para la estructura de datos:

* insert(str, weight): El cual inserta la cadena str asociado al peso weight dentro del Trie de la estructura.
* build(): Obtiene el array de pesos que se obtiene del Trie si este fuera representado como un array ordenado lexicográficamente de cadenas con sus pesos adjuntos. Posteriormente llama al método de construcción en tiempo lineal del Treap pasándole dicho array.

En la figura 3.13 se muestra el código implementado para dichas rutinas, además una rutina extra para poder cargar archivos desde un archivo de texto externo al programa.

void insert(const string &str, int weight) {

ds.insertar(str, weight);

}

void build() {

st.linear\_build(ds.obtenerPesos());

}

void build(const string &filename) {

ifstream in(filename);

string keyword;

int value;

while (in >> keyword >> value) {

insert(keyword, value);

}

in.close();

build();

}

**Figura 3.13:** Código para la carga de datos y construcción de la estructura Suggest

#### 3.2.5.2.1 COMPLEJIDAD COMPUTACIONAL

El algoritmo de construcción, en primera instancia inserta las n cadenas dentro del Trie asociadas a su peso, a continuación, pasa a construir el Treap luego de obtener los pesos asociados de las n cadenas en el Trie. Ya que cada inserción en el Trie toma un tiempo de ejecución equivalente a y se realizan n inserciones, la complejidad computacional de la primera parte resulta en una complejidad computacional igual a

La segunda parte del algoritmo, que corresponde a la construcción del Treap, ya se mostró que este tomara una complejidad igual a

Resultando en una complejidad computacional igual a , el cual es equivalente a en notación O-grande.

### 3.2.5.3 CONSULTAS

Se ha realizado una refactorización al código original de Matani D. que procesa las consultas en la estructura de datos, esto con el objetivo de hacer el código más simple y fácil de entender, además de reducir constantes en la complejidad computacional. En la figura 3.14 se muestra el código:

vector<int> query(const string &prefix, int k) {

range r = ds.query(prefix);//[first, second)

vector<int> answer;

if (r.first == r.second)

return answer;

priority\_queue<phrase\_range> pq;

pair<int,int> best = st.query(r.first, r.second);

pq.emplace(best.value, best.index, r.first, r.second);

while ((int)answer.size() < k && !pq.empty()) {

int index, low, up;

tie(std::ignore, index, low, up) = pq.top();

pq.pop();

answer.push\_back(index);

if (low < index) {

best = st.query(low, index);

pq.emplace(best.value, best.index, low, index);

}

if (index+1 < up) {

best = st.query(index+1, up);

pq.emplace(best.value, best.index, index+1, up);

}

}

return answer;

}

**Figura 3.14:** Código para las consultas en la estructura Suggest

En el apartado 2.6.2 se describe el algoritmo utilizando las estructuras originales Segment Tree y Array estático de cadenas. El código presentado en la figura 3.14 sigue la misma lógica que el original, pero se realizan los cambios mencionados Trie por Array Estatico y Treap por Segment Tree.

#### 3.2.5.3.1 COMPLEJIDAD COMPUTACIONAL

Se procede a realizar el cálculo de la complejidad computacional del código mostrado en la figura 3.14. La operación de consulta al Trie toma una complejidad computacional de Además se puede notar que todas las operaciones realizadas dentro del ciclo while se realizan con un complejidad igual a donde n es el número de elementos que almacena la estructura. También se puede notar que el ciclo while se ejecutara a lo sumo K veces.

Dando una complejidad final de en notación O-grande.

### 3.2.5.4 ADICIONES Y ACTUALIZACIONES

Se desarrolló un método para realizar la adición de nuevas cadenas a la estructura, este se encarga de:

1. Insertar la nueva cadena y su valor asociado al Trie
2. Obtener la posición que ocupa la cadena en el array ordenado lexicográficamente que representa el Trie
3. Insertar en el Treap en dicha posición el valor asociado a la cadena

De igual manera se desarrolló un método para realizar actualizaciones en la estructura, este ejecuta las siguientes operaciones:

1. Obtener la posición que ocupa la cadena en el array ordenado lexicográficamente que representa el Trie
2. Actualizar el Trie con el nuevo valor asociado para dicha cadena
3. Actualizar el Treap en la posición obtenida en el primer paso con el nuevo valor para la cadena

En la figura 3.15 se muestra la implementación de los dos métodos

void add(string key, int weight) {

insert(key, weight);

int position = ds.posicion(key);

st.insert(position, weight);

}

void update() {

int position = ds.posicion(key);

ds.actualizar(key, weight);

st.update(position, weight);

}

**Figura 3.15:** Código para realizar inserción y actualización de datos en la estructura Suggest

#### 3.2.5.4.1 COMPLEJIDAD COMPUTACIONAL METODO DE ADICIÓN

Del método de adición de nuevos datos a la estructura mostrado en la figura 3.15 se obtiene la complejidad computacional de cada instrucción, previamente analizada en los apartados anteriores:

Obteniendo una complejidad computacional final igual a en notación O-grande.

#### 3.2.5.4.2 COMPLEJIDAD COMPUTACIONAL METODO DE ACTUALIZACIÓN

Analizando las instrucciones del método de actualización mostrado en la figura 3.15, se obtiene:

De la cual se obtiene una complejidad computacional igual a en notación O-grande.

# 3.5 CODIGO FUENTE

Todo el código expuesto en este capítulo se encuentra disponible en los anexos C y D. Además, se encuentra disponible para su descarga junto a un archivo con instrucciones para su compilación y ejecución en el repositorio de GitHub: https://github.com/BranimirE/online-cpp-libface

# 3.6 COMPARACIÓN DE COMPLEJIDADES

Dado que parte de los objetivos del presente trabajo es mantener la eficiencia de la estructura de datos frente a la versión original se procede a realizar una comparación de las complejidades obtenidas frente a las originales. En la tabla 3.1 se observa una comparativa de las complejidades obtenidas en los apartados anteriores.

Se debe notar que en las expresiones mostradas en la tabla 3.1 para la estructura propuesta se mantiene las constantes relacionadas al tamaño del alfabeto y tamaño de las cadenas en el Trie, esto debido a que posiblemente repercutirán en el tiempo de ejecución en la fase experimental.

|  |  |  |
| --- | --- | --- |
| **Operación\Complejidad O(n)** | **Versión Original** | **Estructura Propuesta** |
| **Construcción** |  |  |
| **Consulta** |  |  |
| **Adición** | No soportado |  |
| **Actualización** | No soportado |  |

**Tabla 3.1:** Tabla comparación de complejidad computacional entre la versión original y la estructura de datos propuesta

Cuando se realiza análisis asintótico se considera solamente el termino más grande de una expresión, ignorando los coeficientes de este, es decir que la tabla 3.1 será expresada en realidad como:

|  |  |  |
| --- | --- | --- |
| **Operación\Complejidad O(n)** | **Versión Original** | **Estructura Propuesta** |
| **Construcción** |  |  |
| **Consulta** |  |  |
| **Adición** | No soportado |  |
| **Actualización** | No soportado |  |

**Tabla 3.2:** Tabla comparación de complejidad computacional entre la versión original y la estructura de datos propuesta ignorando constantes

Con los resultados mostrados en la tabla 3.2, se podría concluir que se ha mejorado la complejidad para el proceso de construcción.

En cuanto a las consultas, se ha mantenido la eficiencia en ellas.

También se cumplió el objetivo de adicionar nuevas operaciones de adición y actualización a la estructura de datos.

Sin embargo, este resultado es obtenido desde un punto de vista teórico, en el capítulo siguiente se podrá ver si las constantes ignoradas en realidad repercuten en el tiempo de ejecución de las repuestas.

CAPÍTULO IV

MARCO EXPERIMENTAL

4.1 DATASET

4.1.1 PREPARACION DEL DATASET

Tabla numero de cadenas en el data set

Promedio del tamaño de las cadenas en el data set

Cadena mas larga

Cadena mas corta

Tabla de cantidad de cadenas por tamaño

Random suffle

4.1.2 PREPARACION DE DATASET DE CONSULTAS

4.2. TIEMPO DE EJECUCIÓN EXPERIMENTAL

4.2.1 ESPECIFICACIONES DE LA MÁQUINA

4.3 TIEMPO DE EJECUCIÓN ESTRUCTURA PROPUESTA VS VERSION ORIGINAL

4.1.2. TEST DE PERFORMANCE DE LA ESTRUCTURA PROPUESTA

CAPÍTULO V

CONCLUSIONES Y RECOMENDACIONES

5.1. CONCLUSIONES

5.2. RECOMENDACIONES

# 

# BIBLIOGRAFÍA

Laaksonen Antti. (3 de Julio de 2018) Competitive Programmer’s Handbook

Recuperado a partir de

<https://cses.fi/book/index.html>

Ajanki A. (7 de junio de 2016). *Data structures for fast autocomplete*

Recuperado a partir de

https://futurice.com/blog/data-structures-for-fast-autocomplete

Brass P. (2008).  *Advanced Data Structures.* Cambridge University Press, Cambridge,

New York.

Diethelm N. (2014). *Suggest Trees: A Data Structure for Efficient Autocomplete*

Recuperado a partir de

http://suggesttree.sourceforge.net/

Hopcroft John E., Motwani Rajeev y Ullman Jeffrey D. (2001). I*ntroduction to Automata Theory, Languages, and Computation.* Estados Unidos: Addison-Wesley.

Martin john C. (2011). *Introduction to languages and the Theory of computation.*

New York, Estados Unidos: McGraw-Hill.

Matani D. (2 de septiembre de 2011). *An O(k log n) algorithm for prefix based ranked autocomplete*.

Recuperado a partir de

https://www.researchgate.net/publication/268420493\_An\_Ok\_log\_n\_algorithm\_for\_prefix\_based\_ranked\_autocomplete

. (2011). *Algorithms.* Estados Unidos: Addison-Wesley.

Vaijapurkar A. Patani R. Kashyap Jha V. (2 de abril de 2016), *An Auto-completion Algorithm Using Conditional Probability*, Departament of Computer science, Vellore Institute of Technology, Vellore, India.

Recuperado a partir de

ijpsat.ijsht-journals.org/index.php/ijpsat/article/download/39/23

Torrico L. (2008). *Lenguajes Formales Gramática y Autómatas en Informática.* La Paz,

Bolivia.

Sedgewick R. Wayne K. (2011). *Algorithms.* Estados Unidos: Addison-Wesley.

Polozov Alexander (16 de Agosto de 2010) Árbol cartesiano: Parte 1. Descripción, operaciones, aplicaciones.

Recuperado a partir de:

https://habr.com/en/post/101818/

Max Ivanov (9 de Marzo de 2013) Árbol cartesiano

Recuperado a partir de:

<http://e-maxx.ru/algo/treap>

# ANEXOS

**ANEXO A: IMPLEMTANCIÓN DE TREAP**

struct treap {

typedef struct \_node {

int x, y;

\_node \*l, \*r;

\_node(int \_x) : x(\_x), y((rand() << 16) ^ rand()), l(nullptr), r(nullptr) {}

~\_node() {delete l; delete r;}

} \*node;

treap(): root(nullptr) {}

~treap() {delete root;}

node root;

/\*

Divide el arbol que tiene como raiz a "t", guarda en

L todos los nodos con key menor o igual a "x"

R todos los nodos con key mayor estricto a x

t es destruido/modificado

\*/

void split(node t, int x, node &L, node &R) {

if (t == nullptr) {L = R = nullptr; return;}

if (t->x <= x) {

split(t->r, x, t->r, R);

L = t;

} else {

split(t->l, x, L, t->l);

R = t;

}

}

/\*

Une los dos nodos L y R en un solo arbol y los devuelve.

L y R son modificados

\*/

node merge(node L, node R) {

if (L == nullptr) return R;

if (R == nullptr) return L;

if (L->y > R->y) {

L->r = merge(L->r, R);

return L;

} else {

R->l = merge(L, R->l);

return R;

}

}

/\*

Inserta un solo nodo con key igual a "x"

\*/

void insert(int x) {

//verificar que no se inserten elementos repetidos

//pueden ocasionar que ya no cumplan la propiedad de BST

//pueden haber keys iguales a izquierda y derecha

node L, R;

split(root, x, L, R);

root = merge(merge(L, new \_node(x)), R);

}

/\*

Borra todos los nodos con key igual a "x"

No pasa nada si no hay nodos con key igual a "x"

\*/

void erase(int x) {

node L, m, R;

split(root, x, L, R);

split(L, x - 1, L, m);

root = merge(L, R);

}

};

**ANEXO B: IMPLEMENTACIÓN TREAP IMPLICITO**

struct rope {

typedef struct \_node {

int value, y, cnt;

\_node \*l, \*r;

\_node(int \_value) : value(\_value), y((rand() << 16) ^ rand()), cnt(1), l(nullptr), r(nullptr) {}

~\_node() {delete l; delete r;}

void recalc() {

cnt = 1;

if (l) cnt += l->cnt;

if (r) cnt += r->cnt;

}

} \*node;

rope(): root(nullptr) {}

~rope() {delete root;}

node root;

/\*

Divide el arbol que tiene como raiz a "t", guarda en

L los primeros "x" elementos del array

R el primer elemento de R es el elemento en posicion x(indexado desde 0) del array

t es destruido/modificado

L = [0, x)

R = [x, n)

\*/

void split(node t, int x, node &L, node &R) {

if (t == nullptr) {L = R = nullptr; return;}

int curIndex = cnt(t->l) + 1;

if (curIndex <= x) {

split(t->r, x - curIndex, t->r, R);

L = t;

} else {

split(t->l, x, L, t->l);

R = t;

}

t->recalc();

}

/\*

Une los dos nodos L y R en un solo arbol y los devuelve.

L y R son modificados

\*/

node merge(node L, node R) {

if (L == nullptr) return R;

if (R == nullptr) return L;

if (L->y > R->y) {

L->r = merge(L->r, R);

L->recalc();

return L;

} else {

R->l = merge(L, R->l);

R->recalc();

return R;

}

}

/\*

Inserta "value" en la posicion "pos"(indexado desde 0) recorre todos los elementos a la derecha desde la posicion pos

\*/

void insert(int pos, int value) {

node L, R;

split(root, pos, L, R);//en R esta pos

root = merge(merge(L, new \_node(value)), R);

}

/\*

Borra el elemento en posicion pos

\*/

void erase(int pos) {

node L, m, R;

split(root, pos, L, R);

split(R, 1, m, R);

root = merge(L, R);

}

int get(int pos) {

node L, m, R;

split(root, pos, L, R);

split(R, 1, m, R);

int ret = m->value;

root = merge(merge(L, m), R);

return ret;

}

int cnt(node t) const {

return t ? t->cnt : 0;

}

int size() const {

return cnt(root);

}

};

# ANEXO C: CODIGO DE LA ESTRUCTURA DE DATOS PROPUESTA

## Archivo trie.hpp

#include <bits/stdc++.h>

using namespace std;

typedef pair<int,int> range;

typedef pair<string,int> phrase;

#define str first

#define weight second

struct trie {

const static int R = 26;//tamaño alfabeto(A - Z)

const static int nullvalue = INT\_MAX;

struct nodo {

int val, cnt;

nodo\* sig[R];

nodo() : val(nullvalue), cnt(0) {

for (int i = 0; i < R; i++) sig[i] = nullptr;

}

};

nodo\* raiz = nullptr;

trie () {

raiz = new nodo();

}

//Almacena la cadena key dentro del trie, asociado al valor val

//Precondición: key no se encuentra dentro del trie

void insertar(const string &key, int value) {

nodo\* x = raiz;

for (int d = 0; d < (int)key.length(); d++) {

int c = key[d]-'A';

if (x->sig[c] == nullptr) x->sig[c] = new nodo();

x = x->sig[c];

x->cnt++;

}

x->val = value;

}

//Retorna el valor asociado a key en el trie

//Ó nullvalue si el key no se encuentra en el trie

int buscar(const string &key) {

nodo\* x = raiz;

for (int d = 0; d < (int)key.length(); d++) {

int c = key[d]-'A';

if (x->sig[c] != nullptr) x = x->sig[c];

else return nullvalue;

}

return x->val;

}

//Actualiza el valor de asociado a key con val

//Precondición: key se encuentra almacenado en el trie

void actualizar(string key, int val) {

nodo\* x = raiz;

for (int d = 0; d < (int)key.length(); d++)

x = x->sig[(int)key[d]-'A'];

x->val = val;

}

//Muestra por pantalla todas las cadenas almacenadas

//en el trie en orden lexicografico

void mostrar() {

int printed = 0;

mostrar(raiz, "", printed);

}

void mostrar(nodo\* x, string key, int &cnt) {

if (x->val != nullvalue) {

printf("[%d] %s -> %d\n", cnt, key.c\_str(), x->val);

cnt++;

}

for (int i = 0; i < R; i++)

if (x->sig[i] != nullptr) {

mostrar(x->sig[i], key + char('A'+i), cnt);

}

}

vector<int> obtenerPesos() {

vector<int> ret;

recorrer(raiz, ret);

return ret;

}

void recorrer(nodo\* x, vector<int> &ret) {

if (x->val != nullvalue) {

ret.push\_back(x->val);

}

for (int i = 0; i < R; i++)

if (x->sig[i] != nullptr)

recorrer(x->sig[i], ret);

}

/\*

Calcula el rango de indices en el trie si este

representara un array ordenado de cadenas, tal que

todas las cadenas en el rango tienen a prefix como

prefijo

\*/

range query(const string &prefix) const {

nodo\* x = raiz;

int cnt = 0;

for (int d = 0; d < (int)prefix.length(); d++) {

int c = prefix[d] - 'A';

//El prefijo no existe en el trie

if (x->sig[c] == nullptr) {

//cout << "NA" << endl;

return make\_pair(-1, -1);

}

//Contamos la cantidad de cadenas

//lexicograficamente menores

for (int i = 0; i < c; i++)

if (x->sig[i] != nullptr)

cnt += x->sig[i]->cnt;

if (x->val != nullvalue)

cnt++;

x = x->sig[c];

}

return make\_pair(cnt, cnt + x->cnt);

}

/\*

Obtiene la i-esima cadena(y su valor asociado) lexicografica

dentro del trie si este estuviera indexado desde 0

Precondición: 0 <= i < numero de cadenas en el trie

\*/

phrase get(int i) {

i++;

int cnt = 0;

string keyword;

nodo\* x = raiz;

bool found = false;

while (!found) {

bool path\_found = false;

for (int c = 0; c < R and !path\_found; c++) {

if (x->sig[c] != nullptr) {

int s = x->sig[c]->cnt;

if (cnt + s < i) {

cnt += s;

} else {

keyword += char('A' + c);

x = x->sig[c];

path\_found = true;

}

}

}

if (path\_found) {

if (x->val != nullvalue)

cnt++;

if (cnt == i)

found = true;

} else {

return phrase("Not found", -1);

}

}

return phrase(keyword, x->val);

}

// Calcula la posicion en la que se encontraria key

// si todas las cadenas del trie formaran un vector

// ordenado lexicograficamente

// Precondición: key se encuentra en el trie

int posicion(string key) {

nodo\* x = raiz;

int cnt = 0;

for (int d = 0; d < (int)key.length(); d++) {

int c = key[d]-'A';

for (int i = 0; i < c; i++)

if (x->sig[i] != nullptr)

cnt += x->sig[i]->cnt;

x = x->sig[c];

if (x->val != nullvalue)

cnt++;

}

return cnt-1;

}

};

## Archivo treap.hpp

#define value first

#define index second

typedef struct \_node {

int value, y, cnt;

pair<int,int> maxi;//(value, index)

\_node \*l, \*r;

\_node(int \_value) : value(\_value), y((rand() << 16) ^ rand()), cnt(1), maxi(\_value, 0), l(nullptr), r(nullptr) {}

~\_node() {delete l; delete r;}

void recalc() {

cnt = 1;

maxi = make\_pair(value, l?l->cnt:0);

if (l) {

cnt += l->cnt;

maxi = max(maxi, l->maxi);

}

if (r) {

cnt += r->cnt;

maxi = max(maxi, make\_pair(r->maxi.first, (l?l->cnt:0) + 1 + r->maxi.second));

}

}

} \*node;

bool operator < (const \_node &a, const \_node &b) {

return a.maxi < b.maxi;

}

struct treap {

treap(): root(nullptr) {}

~treap() {delete root;}

node root;

/\*

Divide el arbol que tiene como raiz a "t", guarda en

L los primeros "x" elementos del array

R el primer elemento de R es el elemento en posicion x(indexado desde 0) del array

t es destruido/modificado

L = [0, x)

R = [x, n)

\*/

void split(node t, int x, node &L, node &R) {

if (t == nullptr) {L = R = nullptr; return;}

int curIndex = cnt(t->l);

if (x <= curIndex) {

split(t->l, x, L, t->l);

R = t;

} else {

split(t->r, x - curIndex - 1, t->r, R);

L = t;

}

t->recalc();

}

/\*

Une los dos nodos L y R en un solo arbol y los devuelve.

L y R son modificados

\*/

node merge(node L, node R) {

if (L == nullptr) return R;

if (R == nullptr) return L;

if (L->y > R->y) {

L->r = merge(L->r, R);

L->recalc();

return L;

} else {

R->l = merge(L, R->l);

R->recalc();

return R;

}

}

// Inserta "value" en la posicion "pos"(indexado desde 0) recorre todos los elementos a la derecha desde la posicion pos

void insert(int pos, int value) {

node L, R;

split(root, pos, L, R);//en R esta pos

root = merge(merge(L, new \_node(value)), R);

}

// Borra el elemento la posicion pos

void erase(int pos) {

node L, m, R;

split(root, pos, L, R);

split(R, 1, m, R);

root = merge(L, R);

}

int cnt(node t) const {

return t ? t->cnt : 0;

}

int size() const {

return cnt(root);

}

//Retorna el elemento almacenado en la posicion "pos"

int get(int pos) {

node L, m, R;

split(root, pos, L, R);

split(R, 1, m, R);

int ret = m->value;

root = merge(merge(L, m), R);

return ret;

}

//Actualiza el valor asiado en la posicion "pos"

void update(int pos, int value) {

node L, m, R;

split(root, pos, L, R);

split(R, 1, m, R);

m->value = value;

root = merge(merge(L, m), R);

}

//Retorna el maximo elemento en el rango [a, b)

//Retorna (value, index) donde value el maximo elemento

//en el rango [a, b) e index es su indice

//Si es que existieran varios elementos con ese mismo valor

//se retornara el que se encuentre mas a la derecha

pair<int,int> maxInRange(int a, int b) {//el rango es [a, b)

node L, m, R;

split(root, a, L, R);

split(R, b - a, m, R);

pair<int,int> ret = m->maxi;

ret.second += a;

root = merge(merge(L, m), R);

return ret;

}

pair<int,int> query(int a, int b) {

return maxInRange(a, b);

}

// Imprime todos los elementos del treap

void print() {

for (int i = 0, tam = size(); i < tam; i++) {

cout << get(i) << " ";

}

cout << endl;

}

void build(const vector<int>& v) {

for (int i = 0; i < (int)v.size(); i++) {

insert(i, v[i]);

}

}

void linear\_build(const vector<int> &v) {

int n = v.size();

int next\_pow\_2 = 1;

while (next\_pow\_2 <= n) next\_pow\_2 \*= 2;

int jump = max(INT\_MAX/2, n) / next\_pow\_2;

root = \_build(0, n, v, next\_pow\_2, 1, jump);

}

node \_build(int a, int b, const vector<int> &v, int N, int id, int jump) {

if (a == b) return nullptr;

int mid = (a + b) / 2;

node m = new \_node(v[mid]);

m->l = \_build(a, mid, v, N, 2 \* id, jump);

m->r = \_build(mid + 1, b, v, N, 2 \* id + 1, jump);

m->y = (N - id) \* jump;

m->recalc();

assert(m->y > (m->l==nullptr?0:m->l->y));

assert(m->y > (m->r==nullptr?0:m->r->y));

return m;

}

};

## Archivo principal

#define value first

#define index second

typedef tuple<int,int,int,int> phrase\_range;

//(value, index, a, b)

struct Suggest {

trie ds;

treap st;

void insert(const string &str, int weight) {

ds.insertar(str, weight);

}

void build() {

st.linear\_build(ds.obtenerPesos());

}

void build(const string &filename) {

ifstream in(filename);

string keyword;

int value;

while (in >> keyword >> value) {

insert(keyword, value);

}

in.close();

build();

}

vector<int> query(const string &prefix, int k) {

range r = ds.query(prefix);//[first, second)

vector<int> answer;

if (r.first == r.second) {

return answer;

}

priority\_queue<phrase\_range> pq;

pair<int,int> best = st.query(r.first, r.second);

pq.emplace(best.value, best.index, r.first, r.second);

while ((int)answer.size() < k && !pq.empty()) {

int index, low, up;

tie(std::ignore, index, low, up) = pq.top();

pq.pop();

answer.push\_back(index);

if (low < index) {

best = st.query(low, index);

pq.emplace(best.value, best.index, low, index);

}

if (index+1 < up) {

best = st.query(index+1, up);

pq.emplace(best.value, best.index, index+1, up);

}

}

return answer;

}

phrase get(int index) {

return ds.get(index);

}

void add(string key, int weight) {

if (ds.buscar(key) != trie::nullvalue) {

cout << "Entry found in the set, updating its value" << endl;

int position = ds.posicion(key);

ds.actualizar(key, weight);

st.update(position, weight);

} else {

cout << "Adding the new entry" << endl;

insert(key, weight);

int position = ds.posicion(key);

st.insert(position, weight);

}

}

};

void print(Suggest &sug) {

sug.ds.mostrar();

}

int main(int argc, char const \*argv[]) {

Suggest sug;

if (argc > 1) {

string filename(argv[1]);

sug.build(filename);

cout << "DS builded using " << filename << " file." << endl;

#ifdef ACMTUYO

print(sug);

#endif

} else {

cout << "Data structure empty, not using a dataset." << endl;

}

string command;

while (cin >> command) {

char type = toupper(command[0]);

string keyword;

int value;

cin >> keyword >> value;

if (type == 'Q') {//query

auto indexes = sug.query(keyword, value);

for (int index: indexes) {

phrase res = sug.get(index);

cout << index << ":" << res.str << '(' << res.weight << ") ";

}

cout << endl;

} else {//add or update

sug.add(keyword, value);

#ifdef ACMTUYO

print(sug);

#endif

}

}

return 0;

}

# ANEXO D: CODIGO DE LA VERSION ORIGINAL(REFACTORIZADO)

const char SEPARATOR = ' ';

typedef pair<string,int> phrase;

#define str first

#define weight second

typedef pair<int,int> range;

struct Comp {

bool operator()(string const& prefix, phrase const &target) {

const int ppos = target.str.compare(0, prefix.size(), prefix);

return ppos > 0;

}

bool operator()(phrase const &target, string const& prefix) {

const int ppos = target.str.compare(0, prefix.size(), prefix);

return ppos < 0;

}

};

struct data\_store {

vector<phrase> data;//(str, weight)

void insert(const string &str) {

auto found = str.find(SEPARATOR);

if (found != string::npos) {

insert(str.substr(0, found), atoi(str.substr(found).c\_str()));

} else {

insert(str, 0);

}

}

void insert(const string &str, int weight) {

data.emplace\_back(str, weight);

}

void build() {

sort(data.begin(), data.end());

}

range query(const string &prefix) const {

auto its = equal\_range(data.begin(), data.end(), prefix, Comp());

return make\_pair(its.first - data.begin(), its.second - data.begin());

}

phrase get(int index) const {

return data[index];

}

};

#define value first

#define index second

struct segment\_tree {

typedef pair<int,int> node;//pairs(value, index)

const node neutral = make\_pair(INT\_MIN, -1);

vector<node> t;

int n;//Original array's size

void init(const vector<int> &arr) {

n = arr.size();

//next pow of 2

int next\_pow\_of\_2 = (n and (n&(n-1)) == 0)?n:(1<<(32-\_\_builtin\_clz(n)));

t.assign(next\_pow\_of\_2 \* 2, neutral);

build(0, n, 1, arr);

}

node build(int l, int r, int u, const vector<int> &arr) {

if (l >= r) return neutral;

if (l+1 == r) return t[u] = node(arr[l], l);

int m = (l+r)>>1;

return t[u] = max(build(l, m, 2\*u, arr), build(m, r, 2\*u+1, arr));

}

//It returns max query on range [i, j)

node query(int i, int j) {

return query(i, j, 1, 0, n);

}

node query(int i, int j, int u, int l, int r) {

if (i >= r or l >= j) return neutral;

if (i <= l and r <= j) return t[u];

int m = (l+r)>>1;

return max(query(i,j,2\*u,l,m), query(i,j,2\*u+1,m,r));

}

};

typedef tuple<int,int,int,int> phrase\_range;

//(value, index, a, b)

struct Suggest {

data\_store ds;

segment\_tree st;

void insert(const string &str, int weight) {

ds.insert(str, weight);

}

void build() {

ds.build();

print(ds);

vector<int> weights;

for (phrase &item: ds.data) {

weights.push\_back(item.weight);

}

st.init(weights);

}

void build(const string &filename) {

ifstream in(filename);

string line;

while (getline(in, line)) {

ds.insert(line);

}

ds.build();

//print(ds);

vector<int> weights;

for (phrase &item: ds.data) {

weights.push\_back(item.weight);

}

st.init(weights);

}

vector<int> query(const string &prefix, int k) {

range r = ds.query(prefix);//[first, second)

vector<int> answer;

if (r.first == r.second) {

return answer;

}

priority\_queue<phrase\_range> pq;

pair<int,int> best = st.query(r.first, r.second);

pq.emplace(best.value, best.index, r.first, r.second);

while ((int)answer.size() < k && !pq.empty()) {

int index, low, up;

tie(std::ignore, index, low, up) = pq.top();

pq.pop();

answer.push\_back(index);

if (low < index) {

best = st.query(low, index);

pq.emplace(best.value, best.index, low, index);

}

if (index+1 < up) {

best = st.query(index+1, up);

pq.emplace(best.value, best.index, index+1, up);

}

}

return answer;

}

phrase get(int index) {

return ds.get(index);

}

};

int main(int argc, char const \*argv[]) {

Suggest sug;

if (argc > 1) {

string filename(argv[1]);

sug.build(filename);

cout << "DS builded using " << filename << " file." << endl;

} else {

cout << "Data structure empty, not using a dataset." << endl;

}

string command;

while (cin >> command) {

char type = toupper(command[0]);

string keyword;

int value;

cin >> keyword >> value;

if (type == 'Q') {//query

auto indexes = sug.query(keyword, value);

for (int index: indexes) {

phrase res = sug.get(index);

cout << index << ":" << res.str << '(' << res.weight << ") ";

}

cout << endl;

} else {//add or update

//sug.add(keyword, value);

//Don't support updates

}

}

return 0;

}