Informe Estructuras de Datos

Luis Alberto Salazar, Álvaro José Bertín, Guido Ernesto Salazar

Noviembre de 2019

Resumen

En el presente informe se va a mostrar los algoritmos más relevantes o fundamentales de la bioinformática (KMP y búsqueda ingenua), además se va a mostrar de manera clara y precisa los aspectos de los tipos abstractos de datos con referencia a como se modelaría un arbol o un *trie*, por último pero aún más importante mostrar todo el trabajo realizado en base de la primera entrega, es decir, mostrar que en base de la abstracción de datos se pueden elaborar por medio de la programación orientada a objetos, para que se vea como funcionaría la implementación de un arbol y un *trie* en el paradigma de la informática, teniendo en cuenta las respectivas conclusiones sacadas a partir de toda la elaboración del proyecto, por eso mismo también es necesario reconocer que cosas fueron pertinentes corregir o elaborar dentro del proyecto.

1. Introducción

En la historia de la humanidad la época donde más hubieron avances tecnológicos en poco tiempo, fue en el siglo XX, por eso mismo en el campo de biología comenzaron a usar ayudas más tecnológicas y eficientes como los son los algoritmos de búsqueda. Principalmente estos algoritmos se comenzaron a usar en varios proyectos a partir del siglo XXI, además se comenzó a crear un nuevo campo tanto en la informática como en la biología llamado bioinformática, esto como su nombre lo indica se encargar de investigar los componentes biológicos a través de la informática, como se da a la hora de estudiar las cadenas de ADN, esto surgió porque se quería observar como se constituía de forma clara el genoma humano, aquí es donde más entra en juego la informática con proyectos muy ambiciosos (desde el 2003), en esta clase de proyectos se busca fundamentalmente de que las cadenas de ADN de algún organismo biológico tengan patrones en sus componentes (Timina, Adenina, Citosina, Guanina), partiendo de alguna cadena de componentes que se quiera investigar.

Ahora bien, los principales algoritmos de búsqueda en este campo de la bioinformática fueron los algoritmos de KMP y búsqueda ingenua, el algoritmo KMP llamado por estas siglas por sus creadores: Knuth, Pratt, Morris, además sus implementaciones han ayudado a la hora de investigar y conocer más sobre

el genoma humano, es decir, entender de forma más clara como a partir de su composición se puede llegar a comportar. Lo grandes avances que se han logrado en las investigaciones mencionadas anteriormente se fueron dando en la medida en que los investigadores lograran conocer que es lo que quieren llevar con su búsqueda y como lo harían de la mejor manera, es decir, encontrar la mejor solución a la necesidad de más eficiencia, como primera instancia surgió el algoritmo de búsqueda ingenua, que aunque pudiera cumplir con el objetivo de la investigación no era tan eficiente como lo necesitaban, a raíz de esta necesidad se desarrolla el algoritmo de búsqueda KMP, que funcionaba de manera muy eficiente para lo que necesitaban, a comparación del algoritmo de búsqueda ingenua.

Con respecto a lo anterior, todo este proceso de mejora a la hora de buscar el algoritmo de búsqueda más eficiente se dio por las diferencias de la complejidad y elaboración de cada uno, mientras un algoritmo solo necesitaba conocer los fallos de toda la cadena de compuestos a buscar para no tener que repetir las búsqueda con los elementos que ya recorrió, el otro simplemente recorría todos los elementos sin importar que ya los haya repetido, a simple vista no suena tan mal, pero eso solo se puede pensar con una cantidad pequeña de elementos, ahora, no sería lo mismo con millones y millones de elementos a procesar, tardaría mucho su tiempo de ejecución por lo cual no lo haría para nada eficiente.

2. KMP

Como se mencionó en la introducción el algoritmo de búsqueda KMP, es un algoritmo el cual a partir de los fallos encontrados en la cadena que recorre (en este caso, puesto que se está hablando de bioinfomática), no los vuelve a repetir a la hora de buscar los patrones que se necesitan, es decir, no repite elementos a comprar. El entendimiento del algoritmo KMP es sencillo ya que encuentra subcadenas extraídas de una cadena, sin necesidad de analizar más de una vez cada carácter que contiene la cadena.

2.1. Reconocimiento

Al momento de implementar cada cosa dentro del algoritmo se necesitó ciertas observaciones, como:

- ***Vectores:*** Los vectores son estructuras contenedoras que se pueden manipular de manera muy sencilla, esto ayuda a la hora de almacenar cosas dentro de este, como los índices de la función fallo o las subcadenas extraídas.
- ***Clases:*** En el paradigma de la programación orientada a objetos surgieron las clases como una forma de manejar los datos de manera ordena y

estructurada, permitiendo hacer uso de la encapsulación y de ver que cosas se quieren contener y que cosas se desean poder usar fuera de la clase, en esta entrega fue necesario el manejo de clases para la implementación estructurada de los métodos u operaciones, en este caso, que se deseaban realizar.

2.2. Implementación

Si se considera todo lo anterior, ya se puede tener una idea más clara de lo que se requirió para la elaboración del algoritmo KMP, ahora, se dará una explicación precisa del porqué de cada cosa usada dentro del algoritmo.

- Primeramente para usar el algoritmo KMP, se necesita una función fallo para poder reconocer los patrones de la cadena, por eso mismo se realizó una función que de entrada recibe el patrón y su funcionamiento es que dentro de un vector almacena la cantidad de veces que se repite cada carácter del patrón consecutivamente. Además la complejidad de este algoritmo es de $O(n)$, puesto que esto se hace manera lineal la forma de encontrar los fallos del patrón.

- Asimismo, KMP necesita una forma de encontrar los patrones en la cadena, por eso se implementa una función cuyo objetivo es encontrar el índice donde se encontró el patrón, es decir, compara de forma lineal si el primer índice del patrón coincide con los siguientes, de forma que si no coinciden los demás se aumenta el iterador respecto a la función de fallo que hay, en pocas palabras, aumenta no de forma lineal a partir de los fallos que hayan en el patrón, pero en sentido contrario si se haya el patrón la posición se guarda en un vector que se envía al final de la función. Adicionalmente la complejidad de este algoritmo es $O(n+m)$, puesto que la solución de este mismo lo aplica de manera lineal.

3. Algoritmo de Búsqueda Ingenua

En la búsqueda ingenua literalmente la complejidad aumenta debido a que se evalúan todos y cada uno de los elementos de la cadena con el patrón a buscar, es decir, el costo de este algoritmo se conoce como "*cuadrático*", a comparación del algoritmo de KMP, este, evalúa cada uno de los casos posibles que estén a disposición.

Lo que se entiende del algoritmo de búsqueda ingenua es que analiza de forma cuadrática toda la cadena con respecto al patrón, pero en este algoritmo si repite cada carácter de la cadena, por eso es que se diferencia del KMP, ya que puede llegar a recorrer toda la cadena repitiendo todo más de una vez, por eso este algoritmo no es tan eficiente como el KMP, porque analiza más casos de los que necesariamente debería. De esta forma se mostrará que contiene todo este algoritmo a continuación.

En primera instancia se toma en cuenta de que el límite recorrido en la cadena es con respecto al patrón que haya, esto con el fin de que al momento de comparar no se tomen valores que no existen dentro de la cadena, asimismo comienza verificando si los elementos de la cadena coinciden con los elementos del patrón, esto se verifica en todos los casos posibles, si se comprueba de que la subcadena corresponde con el patrón, se añade la posición inicial de la subcadena, que es lo que se busca para encontrar los distintos patrones, de caso contrario no se añade nada y pasa al siguiente elemento de la cadena a recorrer. Además este algoritmo tiene complejidad $O(n \cdot m)$, porque recorre toda la cadena y compara las subcadenas que encuentra con el patrón que hay.

4. TAD Arbol

Como bien se sabe, en todo este campo de la informática existe algo que se llama los TAD, que sirven para representar de forma general un objeto, en este caso se va a mostrar con un arbol, además el TAD sirve para idealizar de cualquier forma el objeto, es decir, el TAD no va ligado con ningún lenguaje de programación, es solo un tipo de modelamiento abstracto sobre algún objeto, sin embargo en los lenguajes de programación se puede adaptar algo según lo que se tenga en el TAD, es decir, que esa abstracción abstracta que se hace del árbol se pueda replicar propiamente en un lenguaje de programación, esto con el fin de ver de una manera más clara el concepto del objeto y como se representaría propiamente en un lenguaje.

Una de las partes más importantes de este proyecto es mostrar de forma clara lo que es la estructura de un árbol, es por eso que para representar un conjunto de valores o datos, que hace referencia a los árboles, se tienen que hacer por medio del TAD que significa `Tipos Abstractos de Datos`, con el TAD se puede representar todo un conjunto de forma más clara y precisa, además de que no depende de ver esta representación en forma de código, en este caso un árbol es una estructura bastante compleja que necesita una representación especifica de todo lo que contiene y de sus propias operaciones.

4.1. Objeto Abstracto

Para el objeto abstracto se hizo la siguiente representación:

TAD Arbol
Arbol $= < R_0, A_1, A_2, A_3, \cdots, A_n >$
ramas = int
profundidad = int

Como se puede observar, R_0 es un elemento, se dice que es un elemento porque ese el nombre que se le dio en este caso a lo que contiene R_0, que pueden ser enteros, cadenas de texto, caracteres, etc. Además los A_i mencionados son árboles contenidos en un árbol, un subarbol, pero de la misma manera terminan siendo árboles contenidos dentro de un árbol, en pocas palabras lo que se quiere mostrar con es implementación abstracta, es que partiendo del árbol inicial ese árbol le sigue un árbol que a la misma vez le sigue otro árbol, sucesivamente hasta que se encuentre un árbol que no le siga otro árbol. Además lo que se conoce como ramas dentro de este TAD al final son los caminos que puede tener el arbol, también que la profundidad se debe tener presente porque debe haber algo que me diga cual es la rama más larga que para este caso serviría la profundidad ya que se cuenta hasta el último arbol.

4.2. Invariante

Para la invariante del TAD, se hizo de la siguiente forma:

$$inv : (\forall_{R_0}.\mathrm{R}_0 \in \mathit{Elementos}) \wedge [(\forall_{A_i}.\mathrm{A}_i \in \mathit{Arbol}) \wedge (0 \leq i \leq n)]$$

Esta invariante se traduce en que todos R_0 son elementos como se mencionó anteriormente, además de que todos los A_i son Arboles contenidos dentro de otro árbol, donde i puede tomar valores desde 0 hasta n, $i = 0$ cuando se llega a un árbol final, es decir, que no lo sigue ningún árbol y $i = n$ cuando se llega al árbol final como se estableció anteriormente.

4.3. Operaciones Primitivas

Operaciones primitivas
crearArbol: None → *Arbol*
maximaProfundidad: Arbol → *Entero*
contarRamas: Arbol → *Entero*
imprimirPreOrden: Arbol → *None*
imprimirPostOrden: Arbol → *None*
imprimirBusquedaAnchura: Arbol → *None*
imprimirBusquedaProfundidad: Arbol → *None*
listarPreOrden: Arbol → *String*
listarPostOrden: Arbol → *String*
listarBusquedaAnchura: Arbol → *String*
listarBusquedaProfundidad: Arbol → *String*

Con lo anterior se puede denotar como sería de manera abstracta la implementación de las operaciones dentro del TAD, en casi todas las funciones se tiene que de entrada recibe un arbol, esto con el fin de que todas las operaciones requieren que se use un arbol para hacer las distintas cosas que hacen, a continuación se va a especificar más detenidamente que hace cada función.

4.4. Crear Arbol

crearArbol( )
"Construye un nuevo árbol, totalmente vacío"
{ *pre:* **True** }
{ *post:* *Arbol* =<> }

Esta operación solamente requiere que se pueda usar la operación y devuelve un árbol que inicialmente está vacío.

4.5. Maxima Profundidad

maximaProfundidad( )
"Retorna en cantidad la longitud del arbol más largo"
{ *pre:* Arbol = <> v $Arbol = R_0, A_1, A_2, \cdots, A_n >$ }
{ *post:* m $\wedge$ $0 \leq m \leq n$ }

En esta operación solamente se requiere que exista un arbol vacío o un arbol con n cantidad de elementos, para retornar m que dependerá de la ramificación más larga que exista dentro del arbol.

4.6. Contar Ramas

contarRamas( )
"Retorna la cantidad de arboles que tiene el arbol"
{ *pre:* Arbol = <> v Arbol = $< R_0, A_1, A_2, \cdots, A_n >$ }
{ *post:* k $\wedge$ $0 \leq k \leq n$}

En esta operación devuelve una cantidad numérica del arbol con respecto a las ramas, es decir, cuantas ramas tiene el arbol en total.

4.7. Imprimir Preorden

imprimirPreOrden( )
"Imprime de forma recursiva la forma estructurada del árbol"
{ *pre:* Arbol = <> v Arbol = $< R_0, A_1, A_2, \cdots, A_n >$ }
{ *post:* None }

Esta operación imprime el orden del arbol rama por rama, esto quiere decir que se va imprimiendo el orden de cada rama del arbol.

4.8. Imprimir Postorden

imprimirPostOrden( )
"Imprime de forma recursiva la forma estructurada del árbol"
{ *pre:* Arbol = <> v Arbol = $< R_0, A_1, A_2, \cdots, A_n >$ }
{ *post:* None}

En esta operación se imprime los nodos del árbol, pero imprime cada ramificación al revés, es decir, que imprime desde el nodo hoja hasta el nodo donde empieza cada rama.

4.9. Imprimir Búsqueda Anchura

imprimirBusquedaAnchura( )
"Imprime el arbol imprimiendo primero los Nodos"
{ *pre:* Arbol = <> v Arbol = $< R_0, A_1, A_2, \cdots, A_n >$ }
{ *post:* None}

Esta operación imprime el árbol por el método level-order, es decir, que comienza imprimiendo todo los nodos del nivel 0, luego todos los nodos del nivel 1, luego los del nivel 2 y así sucesivamente, hasta que llegue al nivel n que es donde van a estar la mayoría de los nodos hoja.

4.10. Listar Preorden

listarPreOrden( )
"Retorna una cadena de elementos siguiendo la estructura"
{ *pre:* Arbol = <> v Arbol = $< R_0, A_1, A_2, \cdots, A_n >$ }
{ *post:* $< R_0, A_1, A_2, \cdots, A_n > \wedge A_i \in Arbol$}

Esta operación permite retornar una lista con las ramificaciones que al final terminan conteniendo arboles, es decir, retorna una lista con los elementos de las ramas de los arboles siguiendo el orden de arriba para abajo y de izquierda a derecha.

4.11. Listar Postorden

listarPostOrden( )
"Retorna una cadena de elementos siguiendo la estructura" { *pre:* Arbol = <> v Arbol = $< R_0, A_1, A_2, \cdots, A_n >$} { *post:* $< A_n, \cdots, A_2, A_1, R_0 >$ }

En esta operación se permite visualizar todos los nodos de las ramas que sigue, es decir, que permite ver desde el nodo hoja hasta donde empieza la ramificación, parecido al anterior solamente que el orden está invertido pero el sentido sigue siendo el mismo de izquierda a derecha, pero ahora de abajo hacia arriba.

4.11.1. Listar Búsqueda Anchura

listarBusquedaAnchura( )
"Retorna una cadena al reccorrer el arbol según el nodo" { *pre:* Arbol = <> v Arbol = $< R_0, A_!, A_2, ; A_n >$ } { *post:* $< A1, A2, \cdots,_n > \wedge 0 \leq n \leq m \wedge m = profundidad$}

En esta operación se retorna una cadena de nodos obtenidos por el método level-order, que es el recorrido de nodos desde el nivel 0 hasta el nivel m, donde m es la máxima profundidad que tiene el arbol, quiere decir que los n arboles pueden llegar hasta m según su nivel.

4.12. Listar Búsqueda Profundidad

listarBusquedaProfundidad( )
"Retorna una cadena al reccorrer el arbol" { *pre:* Arbol = <> v Arbol = $< R_0, A_!, A_2, \cdots, A_n >$ } { *post:* $< A_1, A_2, \cdots, A_n >$}

En esta operación se recorre el árbol por medio de las ramas, entonces se van anexando a la cadena retornada los nodos que va encontrando a medida que va recorriendo cada rama dentro del arbol.

5. TAD Trie

Ahora bien, luego de hacer la base para el TAD Trie, pues este sirve para recuperar datos que los representa por medio de símbolos, es decir, así como el TAD Arbol se podía ver como varias tuplas de Nodos como claves y de valores los arboles, pues en este caso usa como claves los símbolos para guiar la manera de encontrar toda la información que se termina conteniendo en el nodo hoja, en pocas palabras, a medida que avanza en cada clave, obtiene más información de como va a quedar el nodo hoja que es el que tiene todas las claves almacenadas del arbol.

5.1. Objeto Abstracto

TAD Trie
Trie $= \{Arbol =< R_0, A_1, \cdots, A_n >,\ Patron =< R_0, R_1, \cdots, R_n >\}$

Para representar de manera más clara el objeto abstracto se tomó en cuenta como base el TAD Arbol, por eso mismo para entender la forma del *Trie* se requiere comprender de que *Trie* contiene un Arbol y con tiene una lista de elementos que se llaman Patrones porque es en donde se va a almacenar todos los patrones encontrados dentro del arbol interno que contiene al ir anexando cadenas y cadenas al arbol.

5.2. Invaraiantes

$\{\ inv:\ \ (\forall_{Ri}.R_i \in Elementos) \wedge (\forall_{Ai}.A_i \in Arbol) \wedge (0 \leq i \leq n)\}$

Esta invariante quiere decir que R_i al ser un elemento se caracteriza como un caracter y que internamente como se había mencionado antes el *Trie* contiene un arbol interno que es el que usa como su base para su funcionalidad, que A_i es un Arbol, es decir, que es un Arbol que contiene a otro Arbol, que contiene a otro Arbol, etc.

5.3. Operaciones primitivas

Operaciones primitivas
AgregarPatron: String $\rightarrow$
BuscarPatronesEnCadena: String $\rightarrow vector$
ImprimirPatrones: String $\rightarrow$
ObtenerPatrones: $\rightarrow vector$

Estas operaciones permiten de manera general ver el funcionamiento de los patrones que tiene el árbol y como se modelan a través de este, a continuación se va a explicar de manera más detallada el funcionamiento de cada operación.

5.4. Agregar Patron

agregarPatron(cad)
"Esta operación agrega una cadena de búsqueda."
{ *pre:* cad $\in String \wedge cad =< u_1, u_2, u_3, ..., u_n > \rightarrow u_n\{caracteres\ especiales\}$}
{ *post:* Trie.SubArbol $= < ..., SubArbol_n, cad > \wedge$ Trie.Nodo $= < Nodo_1, Nodo_2, Nodo_3, ..., Nodo_n >$, si el orden de los caracteres no aparece antes en los niveles correspondientes; Trie.SubArbol $= < ..., SubArbol_n >$ $\wedge$ Trie.Nodo $= < Nodo_{i+1}, Nodo_{i+2}, Nodo_{i+3}, ..., Nodo_n >$ Nodo$_i$ $\in$SubArbol$_i$ si un carácter aparece en el nivel correspondiente. Trie.Patron $= < ..., Patron_n, cad >$}

Esta operación permite que una cadena pasada como parámetro, que es un patrón, termine convirtiendo todos los patrones en nodos dentro del trie, pero esto depende del orden que tenga cada carácter de la cadena pasada, ya que para organizar esto dentro del árbol, se tiene en cuenta los niveles en donde están los demás nodos y los nodos hoja, es por eso que se comienza agregando un nodo inicial del patrón y este tiene que seguir hasta que haya un nodo hoja que termine conteniendo el patrón agregado.

5.5. Busca rPatrones En Cadena

buscarPatronesEnCadena(cad)
"Esta operación retorna las posiciones si encuentra la cadena." { *pre:* cad $\in$ *string*} { *post:* return card(vector< *iterator* >), 0 si cad $= < u_1, u_2, u_3, \cdots, u_n > \wedge$ $(\exists_{subcadena}$. subcadena $\subseteq$ *cad*$\wedge$ subcadena $= < u_i, u_{i+1}, \cdots, u_{i+m} >$ $\wedge$ subcadena $\in Trie.subarbol) \wedge$ i + m $\leq n$ return card(vector< *iterator* >) $= 0$; si cad $=$ ¡u$_1, u_2, u_3, \cdots, u_n >$ $\wedge$ $(\neg\exists_{subcadena}$.subcadena $\subseteq$ *cad*$\wedge$ subcadena $= < u_1, u_2, \cdots, u_n >$ $\wedge$ subcadena $\in Trie.subarbol) \wedge$ $i + m \leq n$}

En esta operación se permite buscar todos los patrones que puedan haber dentro de una cadena pasada coomo parámetro, después se encuentra con dos caminos esta operación, si la cadena pasada no coincide con ningún patrón el vector que se retorna contiene 0 posiciones, si tiene coincidencias con los patrones, entonces retorna un vector con las posiciones de los patrones encontrados.

5.6. Imrpimir Patrones

imprimirPatrones( )
"Esta operacion imprime por pantalla los patrones que se van a buscar." { *pre:* **True** } { *post:* }

Esta operación permite visualizar por pantalla los patrones que se tienen y que se desean mostrar.

5.7. ObtenerPatrones

ObtenerPatrones( )
"Esta operación devuelve en un vector los patrones que se van a buscar ordenados." { *pre:* **True** } { *post:* return vector< *string* >, string $\in$ *Patrones* $\wedge$ card(vector) = card(patrones) $\wedge$ *sort*(*vector*)}

En esta operación permite obtener todos los patrones de manera ordenada que se quieren buscar, es decir, los patrones que se desean buscar, esta operación los obtiene y los organiza.

6. Implementación del TAD Arbol

Para la implementación del TAD Arbol en C++, se puede notar que fácilmente como se tiene representado el objeto abstracto se puede representar de la misma lógica programándolo.

6.1. Implementación en clases

Primeramente como atributos privados están el elemento, es decir el nodo que es donde se guardan los caracteres, seguido de la profundidad que llevaría el elemento hasta el momento y la rama en la que se encuentra, esto con el fin de saber exactamente con que características en cuestión de profundidad y en que rama se encuentra para que sea más sencillos diferenciarlos de los otros nodo.

En lo atributos públicos están los prototipos de los métodos que debería llevar la clase, sin mencionar de las funciones que no hacen parte de la clase Arbol, pero que sirven como ayuda para hacer más operaciones necesarias dentro de la clase.

6.2. Implementación de Métodos

- **Funciones de ayuda:** En estos dos métodos se les usa para hallar de distintas formas los elementos que se piden en los métodos que tienen que ver con profundidad, anchura, preorden postorden. En el método *devolver* se usa para dos cosas para el método listar con profundidad y listar con preorden, por eso recibe un parámetro j que permite saber que parte del algoritmo debe usar, en el segundo método que aparece *buscarAnchura*, que permite buscar los nodos por level-orden, además se usaron funciones recursivas ya que las funciones recursivas nos ahorran el hecho de tener que devolvernos dentro de un arbol, ya que con cada rama que se recorre

se devuelve automáticamente a donde empieza cada rama evaluando si se debe seguir o se debe devolver al otro principio de rama, y así sucesivamente, el problema con los ciclos es que hay que tener en cuenta donde habría que devolverse, donde no y eso cambia a medida de que vayan cambiando los datos a recorrer.

- **Procedimiento para crear nuevas ramas en el arbol:** en esta parte solamente se encuentra el método agregarPatron que evalúa si un arbol está vacío o no, busca en que punto es diferente el patrón con una rama del arbol que ya tiene, después de buscar comienza a añadir los nodos del patrón, esto si directamente el arbol está vacío o después de hacer la primera búsqueda anterior.

7. Implementación TAD Trie

Para la implementación del TAD Trie, anexando el porqué también del TAD Arbol, se usaron vectores con el fin de que se necesitaba un estructura para que cuando se guardasen los elementos, se pudieran acceder a ellos de forma más sencilla, ya que a comparación de otras estructuras como las listas circulares dobles, pilas o colas, ninguna de las estructuras anteriores cumplían el los requerimientos que se necesitaban para hacer la implementación, por eso es que los vectores cumplen esta función, además de que en sus operaciones la complejidad no es tan grande a comparación de otras estructuras.

7.1. Implementación en Clases

Para los atributos solamente se necesitaron dos, un arbol y un vector que contuviera los patrones encontrados, esto porque para hacer un *Trie* se necesita la base de un arbol y todo lo que lleva ligado a esto, y algo en lo cual almacenar los patrones que se encuentre en las diferentes cadenas ingresadas, una vez más se usó un vector con el fin de tener más comodidad y simpleza a la hora de almacenar los datos encontrados.

En los atributos públicos se encuentran los prototipos de cada uno de los métodos usados en el *Trie*.

7.2. Implementación de Métodos

- **existeHijo:** En esta función se compara si dentro de lo que tiene en el arbol encuentra una letra a la misma que está comparando retorna la posición, de lo contrario y además si el arbol está vacío retorna -1.

- **buscarPatronesEnCadena:** En este método se encuentra un patrón en la cadena pasada a dicho método, por eso primero se verifica de que el caracter exista dentro del arbol, de ser diferente a -1 verifica si es nuevo patrón a medida que se van almacenando los caracteres, por último se tiene que verificar de que el arbol no sobrepase el tamaño de la cadena

para que en la proxima letra que se esté añadiendo estén los hijos del caracter en el que se encuentra.

- Además se usa la función sort para ordenar los patrones, por lo que se necesitan devolver todos los patrones pero de forma ordena lexicográficamente.

8. Conclusiones y Aspectos a Mejorar

Por último solo queda recalcar que al momento de querer mostrar un conjunto de valores de forma muy específica se debe usar el TAD como medio para visualizar de mejor manera toda la información que se quiere implementar, otra cosa importante que cabe recalcar es que el uso de árboles como forma para encontrar y seguir una guía de patrones, es pertinente ya que facilita la forma no solo conceptual, sino algorítmica de como encontrar y organizar los patrones que se tienen de cualquier cadena de ADN.

De la misma forma se aprendió la implementación de los algoritmos de búsqueda de patrones, como se usan, y que tan eficiente pueden llegar a ser dependiendo de lo que se quiera buscar. Asimismo, todas estas bases sirvieron como una buena guía a la hora de implementar las cosas en la programación, por ejemplo, si no se tuviese un bueno diseño del TAD Arbol, muy difícilmente se podría hacer lo mismo pero en código, ya que el TAD sirve a la hora de que se quiera implementar algo abstracto con programación.

Por otro lado, después de haber realizados las implementaciones del TAD Arbol y el TAD Trie se concluyeron tres cosas:

- La primera es que por cuestiones de complejidad dentro de sus operaciones internas era más sencillo usar listas circulares dobles que vectores, ya que el costo de la mayoría de operaciones de los circulares dobles son constantes, lo cual por un lado lo hacer mejor que un vector, además se trabajó un vector como si fuera una lista simple, ya que no se podía devolver, por esto mismo fue necesario la recursión para poder devolver a un punto especifico de una rama, sin embargo con listas no hubiera sido necesario la utilización de la recursión, porque directamente se hubiera podido acceder a la posición de memoria y directamente desarrollar un método para volver a la posición, que hubiera se hubiera facilitado más la búsqueda.
- La segunda es que al programación orientada a objetos realiza una agrupación precisa de los elementos que debería tener para la realización de alguna de estas implementaciones, es decir, que sin la programación orientada a objetos no hubiera sido tan específico saber que cosas debería tener cada implementación, como se debieron haber implementado, da un orden y una estructura organizada de lo que se quiere hacer para cada cosa, lo cual permite una abstracción de datos más precisa y amplia de que es lo que en verdad se quería hacer.

- La tercera es que si se asume que se mete un patrón a un cola de prioridad y esta trabaja con la complejidad de un heap que es O($log_2 n$), entonces en este caso se metería en una pila que el costo de la operación es O(n) y luego a un vector que seguiría siendo O(n), por lo tanto el costo de las operaciones sigue siendo menor que el costo de un algoritmo de ordenamiento que es O($nlogn$), al momento de implementarlo esto hubiera sido mejor por cuestiones de complejidad que trabajar con un algoritmo de ordenamiento.

Como última instancia, cabe recalcar que un aspecto a mejorar dentro de todo el proyecto fue haber implementado en el TAD Arbol el método *Construir Arbol*, ya que dentro de todo el marco de realización no hubo oportunidad de terminarlo por cuestiones de modificación del mismo proyecto. Otro aspecto a mejorar sería que en el TAD Trie se debió haber modificado primero antes de implementarlo en C++, ya que al mejorarlo hubiera sido más sencillo la interpretación del TAD a la hora de codificar y no tener que tener problemas conceptuales porque no se sabe que poner de atributos o no se sabe como implementar los métodos con los atributos, etc.