

Nmer: Ejemplo de uso del TDA Ktree
V1

Generado por Doxygen 1.8.6

Viernes, 3 de Febrero de 2017 23:46:40

Índice

1	genData	1
1.1	NOTA IMPORTANTE	2
1.2	Generar Documentación	2
1.3	Introducción	2
1.4	Implementación del TDA Nmer.	3
1.5	Árboles ktree	4
1.5.1	TDA ktree<T,K>	4
1.5.2	Implementación	5
1.6	Representando un Nmer como un ktree	7
1.6.1	Representación	7
1.7	¿Cómo implementar algunos de los métodos en Nmer?	8
1.7.1	Método 1 -> Nmer::sequenceADN(tama, adn) : Obteniendo los Nmers a partir de una cadena de ADN	8
1.7.2	Método 2 -> Nmer::loadSerialized(nombre)	10
1.7.3	Método 3 -> set<pair<string,int>,OrdenCre > Nmer::rareNmer(threshold)	10
1.7.4	Método 4 -> set<pair<string,int>,ordenDecre > commonNmer(threshold)	11
1.7.5	Método 5 -> set<pair<string,int>, ordenCrec > level(l);	11
1.7.6	Método 6 -> float Distance(Nmer) : Aplicación del TDA Nmer a la caracterización del genoma de distintas especies.	11
1.8	Entrega	12
1.9	Bibliografía	13
2	Lista de tareas pendientes	13
3	Lista de bugs	13
4	Índice de clases	13
4.1	Lista de clases	13
5	Índice de archivos	14
5.1	Lista de archivos	14
6	Documentación de las clases	14
6.1	Referencia de la Clase ktree< T, K >::node::child_iterator	14
6.1.1	Descripción detallada	15
6.1.2	Documentación del constructor y destructor	15
6.1.3	Documentación de las funciones miembro	15
6.1.4	Documentación de las funciones relacionadas y clases amigas	15
6.2	Referencia de la Clase ktree< T, K >::const_node::child_iterator	15
6.2.1	Descripción detallada	16
6.2.2	Documentación del constructor y destructor	16

6.2.3	Documentación de las funciones miembro	16
6.2.4	Documentación de las funciones relacionadas y clases amigas	16
6.3	Referencia de la Clase <code>ktree< T, K >::const_node</code>	16
6.3.1	Descripción detallada	17
6.3.2	Documentación del constructor y destructor	17
6.3.3	Documentación de las funciones miembro	18
6.3.4	Documentación de las funciones relacionadas y clases amigas	19
6.4	Referencia de la plantilla de la Clase <code>ktree< T, K ></code>	19
6.4.1	Descripción detallada	20
6.4.2	Documentación de los 'Typedef' miembros de la clase	21
6.4.3	Documentación del constructor y destructor	21
6.4.4	Documentación de las funciones miembro	21
6.4.5	Documentación de los datos miembro	25
6.5	Referencia de la Clase <code>Nmer</code>	25
6.5.1	Descripción detallada	26
6.5.2	Documentación de los 'Typedef' miembros de la clase	26
6.5.3	Documentación del constructor y destructor	26
6.5.4	Documentación de las funciones miembro	26
6.6	Referencia de la Clase <code>ktree< T, K >::node</code>	28
6.6.1	Descripción detallada	29
6.6.2	Documentación del constructor y destructor	29
6.6.3	Documentación de las funciones miembro	30
6.6.4	Documentación de las funciones relacionadas y clases amigas	31
7	Documentación de archivos	31
7.1	Referencia del Archivo <code>doc/doxys/genData.dox</code>	31
7.2	Referencia del Archivo <code>include/ktree.h</code>	31
7.2.1	Descripción detallada	32
7.3	Referencia del Archivo <code>include/Nmer.h</code>	36
7.3.1	Descripción detallada	36
7.4	Referencia del Archivo <code>src/ejemploKtree.cpp</code>	36
7.4.1	Documentación de las funciones	37
7.5	Referencia del Archivo <code>src/Nmer.cpp</code>	38
Índice		39

1. genData

Versión

v1

Autor

Juan F. Huete y Carlos Cano

1.1. NOTA IMPORTANTE

Esta práctica es individual, por lo que el alumno debe incluir una nota en la misma indicando que no ha utilizado material de otros compañeros o compañeras para su resolución.

En esta práctica deben seguirse los principios de compilación separada para las clases que no utilicen templates.

El alumno podrá dotar al tipo de dato de otros métodos que considere necesarios para la correcta realización de la práctica, pero debe respetar escrupulosamente la cabecera de los métodos detallados en este documento.

1.2. Generar Documentación

La práctica se entrega con un fichero pdf que contiene toda documentación así como un fichero .zip que contiene todos los ficheros necesarios para su realización. Al descomprimir el fichero nos encontraremos con un fichero Makefile y los siguientes directorios:

- bin <– Directorio de ejecutables
- datos <– Directorio donde se encuentran los ficheros de datos necesarios para la correcta ejecución de la misma
- doc <– Directorio para la documentación de la práctica
- include <– Donde se incluyen los ficheros cabecera .h,
- lib <– Directorio de bibliotecas
- obj <– Directorio de código objeto
- src <– Donde se localizan los fuentes

Para generar la documentación es necesario ejecutar

```
make documentacion
```

como resultado tendremos que la documentación (en versión html) se generará en el directorio `doc/html` y en versión latex se genera en `doc/latex`.

Para obtener el ejecutable podemos hacer

```
make clean  
make
```

que nos generará el ejecutable en el directorio `bin`, que podremos ejecutar

```
cd bin  
./ejemploKtree
```

1.3. Introducción

Un n-mer o n-grama se define como todas las posibles subcadenas de longitud n contenidas en una cadena [1]. En genética computacional, los n-mers son todas las posibles subsecuencias (de longitud igual a n) de una cadena de ADN (donde los nucleótidos posibles son A, G, C, T). El número máximo de n-mers diferentes de longitud n en una cadena de ADN será pues de 4^n .

El siguiente ejemplo muestra una lista de n-mers (para la longitud n especificada) obtenida de una secuencia de ADN:

```

Secuencia:      AGATCGAGTG
3-mers: AGA GAT ATC TCG CGA GAG AGT GTG

Secuencia :      GTAGAGCTGTAG
5-mers: GTAGA TAGAG AGAGC GAGCT AGCTG GCTGT CTGTA TGTAG

```

El conteo de n-mers para una secuencia de ADN es de especial interés en genética computacional porque estudios científicos muestran que estas distribuciones de n-mers caracterizan la especie a la que pertenece ese ADN (ver, por ejemplo, [2]). Es decir, a partir del conteo de n-mers de una secuencia de ADN dada (si la secuencia es lo bastante larga y el valor de n es suficientemente elevado), podríamos identificar la especie de la que procede ese ADN.

El problema del conteo de n-mers para una secuencia de ADN se puede definir como recontar cuántas veces aparece cada n-mer en la secuencia. En nuestro caso, nos interesará recontar cuántas veces aparece cada n-mer de longitud 1,2,...,n en la secuencia, es decir, para un valor dado de n, contaremos la frecuencia de sus 1-mers, 2-mers, 3-mers, ..., hasta n-mers. El siguiente ejemplo muestra el conteo de n-mers de longitud 1 hasta 5 en una secuencia de ADN:

```

Secuencia :      GTAGAGCTGTAG
1-mers: G:5 T:3 A:3 C:1
2-mers: GT:2 TA:2 AG:3 GA:1 GC:1 CT:1 TG:1
3-mers: GTA:2 TAG:2 AGA:1 GAG:1 AGC:1 GCT:1 CTG:1 TGT:1
4-mers: GTAG:2 TAGA:1 AGAG:1 GAGC:1 AGCT:1 GCTG: 1 CTGT:1 TGTA:1
5-mers: GTAGA:1 TAGAG:1 AGAGC:1 GAGCT:1 AGCTG:1 GCTGT:1 CTGTA:1 TGTAG:1

```

En esta práctica proponemos que el estudiante desarrolle el TDA [Nmer](#) para realizar el conteo de n-mers de longitud 1,2,...,n para una secuencia dada de ADN (ver sección [Implementación del TDA Nmer](#)). Para ello, el estudiante debe utilizar el TDA [ktree](#) (ver sección [Representando un Nmer como un ktree](#)). Una vez programado, el TDA [Nmer](#) se aplicará a la resolución de un problema real de biología computacional: se pondrán a disposición del estudiante secuencias reales de ADN de distintas especies, y el estudiante deberá determinar qué especies son más parecidas entre sí en base al recuento de los n-mers más frecuentes (ver sección [Método 6 -> float Distance\(Nmer\) : Aplicación del TDA Nmer a la caracterización del genoma de distintas especies](#)).

1.4. Implementación del TDA Nmer.

La primera tarea a la que se enfrenta el estudiante es la implementación del TDA [Nmer](#). Su especificación se detalla a continuación:

```

class Nmer {
    Nmer();
    Nmer(const Nmer & a);

    void list_Nmer( ) const; // lista todos los Nmer del árbol siguiendo un recorrido en preorden;

    unsigned int length() const; // Devuelve la longitud máxima de los Nmers almacenados

    size_type size() const; // Devuelve el número de Nmers distintos almacenados

    Nmer & operator=(const Nmer & a); // operador de asignacion

    Nmer Prefix(string adn); // Devuelve el Nmer (subarbol) asociado a un prefijo. Por ejemplo, si
    adn es "ACT", devuelve el Nmer que representa todas las subcadenas que empiezan por "ACT" (ACT*)

    Nmer union(const Nmer reference); // Se devuelve un Nmer donde para cada nodo (representa una
    secuencia) se computa la suma de las frecuencias en *this y en referencia,

    bool containsString(const string adn) const; // Devuelve true si la cadena adn está
    representada en el árbol.

    bool included(const Nmer reference) const; // Devuelve true si cada nodo de *this está
    también representado en reference, es decir, si todas las secuencias representadas en el árbol de *this están
    también incluidas en reference. False en otro caso.

    ...

    //El resto de cabeceras se describe en otra sección de este documento:

    void sequenceADN(unsigned int l, const string & adn);

    bool loadSerialized(const string nombre_fichero);

```

```

float Distance(const Nmer & x);

set<pair<string,int>,OrdenCre > rareNmer(int threshold);

set<pair<string,int>,ordenDecre > commonNmer(int threshold);

set<pair<string,int>, ordenCrec > level(int l);

...
}

```

El alumno podrá dotar al tipo de dato de otros métodos que considere necesarios para la correcta realización de la práctica, pero debe respetar escrupulosamente la cabecera de los métodos detallados anteriormente.

Para la implementación del TDA `Nmer` debemos hacer uso de TDA `ktree` (ver sección [Representando un Nmer como un ktree](#)).

1.5. Árboles ktree

Una de las desventajas de usar un vector o lista enlazada para almacenar datos es el tiempo necesario para buscar un elemento. Puesto que tanto vectores como listas enlazadas son estructuras lineales, el tiempo requerido para buscar una lista "lineal" es proporcional al tamaño del conjunto de datos, esto es $O(n)$. Una alternativa para resolver este problema es considerar un conjunto ordenado de elementos, pero igualmente ambas estructuras tienen comportamiento ineficiente cuando tenemos que abordar las tareas de inserción y borrado de elementos ($O(n)$).

Este comportamiento no es aceptable en el mundo de hoy, donde la velocidad a la que realizamos las operaciones es extremadamente importante. El tiempo es dinero. Por lo tanto, parece que se necesitan estructuras de datos mejores (más eficientes) para almacenar y buscar datos.

En esta práctica analizaremos cómo una estructura de datos árbol nos puede ayudar a resolver algunos de nuestros problemas. Un árbol es una colección de nodos conectados formando una estructura jerárquica (y por tanto no lineal) donde para cada nodo tiene un único padre y una lista de hijos.

En principio, un nodo puede tener un número indeterminado de hijos, hablamos de árboles generales, pero en esta práctica nos centraremos en el estudio de un tipo particular de árboles, que denominaremos `ktree<T,K>` y que tiene la siguiente características:

- Un nodo tiene entre 0 y k hijos, que se ubican en una posición dada por un índice, así hablamos del j-ésimo hijo, con $0 \leq j < k$
- Por ser un árbol, cada nodo del mismo tiene un único padre.
- Que exista el hijo j-ésimo no implica que tengan que existir los hijos anteriores, esto es, si $k = 10$ puede ocurrir que un nodo tenga los hijos segundo, quinto y séptimo. El resto de sus descendientes pueden estar vacíos.

1.5.1. TDA `ktree<T,K>`

Se ha diseñado un TDA `ktree` (la documentación está adjunta con esta práctica). El tipo `ktree` tiene dos parámetros plantilla, `ktree<T,K>`, donde T representa al tipo de dato que se almacena en el nodo y K es un entero que representa el número máximo de hijos que se permiten en cada nodo. Es constante para todo el tipo.

Por ejemplo:

```

ktree<int,4> a; // árbol de enteros con 4 hijos como máximo
ktree<string,7> b; // árbol de cadenas con 7 hijos como máximo
ktree<int,2> c; //árbol de enteros con 2 hijos como máximo (es un árbol binario)

```

Indicar que a y c pertenecen a tipos distintos, esto es `ktree<int,4>` es un tipo distinto de `ktree<int,2>`.

Cuando trabajamos con el TDA `ktree` tenemos definidos los siguientes

- `ktree<T,K>` -> Representa el árbol

- `ktree<T,K>::node ->` Representa un nodo del árbol
- `ktree<T,K>::const_node ->` Representa un nodo constante del árbol (no podemos modificar su contenido)
- `ktree<T,K>::node::child_iterator ->` Iterador sobre los hijos (no nulos) de un nodo
- `ktree<T,K>::const_node::child_iterator ->` Iterador sobre los hijos (no nulos) de un nodo constante

Así por ejemplo, la figura siguiente representa un `ktree<char,5>`

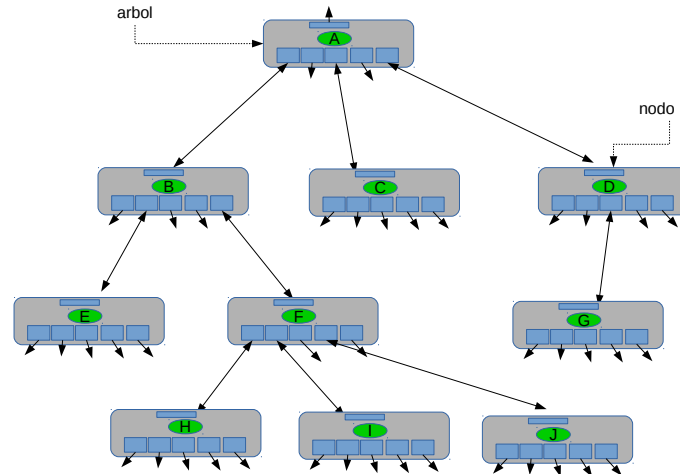


Figura 1: Ejemplo de ktree

```
ktree<char,5> arbol;
ktree<char,5>::node n, nodo;
n = arbol.root();
cout << *n // Nos imprime A
nodo = n.k_child(4); // <- Apunta al nodo con etiqueta D

for (ktree<char,5>::node::child_iterator ith : n)
    cout << *(ith) << endl; // <-- Imprime los hijos no nulos de n, esto es B,C,D
                                // *ith es el nodo hijo, *(ith) es la etiqueta que hay dentro de dicho
                                nodo.
```

Con el guión se entrega un código que ilustra el uso de ktree para distintos problemas (archivo [ejemploKtree.cpp](#)), como por ejemplo recorridos, cálculo de la altura de un ktree, inserción en un árbol binario de búsqueda representado mediante un `ktree<int,2>`, etc.

Este código, una vez compilado, debe ejecutarse desde la carpeta `bin`:

```
cd bin
./ejemploKtree
```

1.5.2. Implementación

Puesto que conocemos el número máximo de hijos de un nodo, fijo en todo momento, podemos acceder a ellos considerando un acceso aleatorio. Esto es consultar el hijo *j*-ésimo en orden constante. Para ello, se implementa un nodo usando la siguiente estructura,

```
struct celdaArbol {
    T etiqueta;
    celdaArbol * padre;
    celdaArbol * hijos[K];
};
```

donde

- etiqueta representa a la etiqueta almacenada en el nodo
- padre es un puntero a la celdaArbol que contiene al padre de dicho nodo
- hijos es un array de tamaño fijo, K, donde en cada posición almacenamos un puntero al hijo j-ésimo ($0 \leq j < K$) del nodo en el árbol

Considerando esta información un árbol ktree se implementará como un puntero a la celda que contiene el nodo raíz del mismo.

```
template <typename T, int K>
class ktree {
public:
    ktree();
    ....
private:
    celdaArbol * laraiz;
};
```

Como hemos comentado el TDA ktree contiene los siguientes tipos definidos

- `ktree<T,K>::node ->` Representa un nodo del árbol
- `ktree<T,K>::const_node ->` Representa un nodo constante (no podemos modificar su contenido) en el árbol

Donde para cada nodo (node o const_node), su representación interna no será mas que un puntero a la celda del árbol a la que apunta, diferenciándose en el tipo de operaciones (especificación) que permiten realizar sobre el mismo (ver documentación asociada).

Si consideramos los iteradores, esto es,

- `ktree<T,K>::node::child_iterator ->` Iterador sobre los hijos no nulos de un nodo
- `ktree<T,K>::const_node::child_iterator ->` Iterador sobre los hijos no nulos de un nodo constante

Ambos iteradores tienen una representación un poco más compleja, al considerar tanto el nodo sobre el que queremos iterar como el hijo en el que ubica en este momento.

La siguiente imagen ilustra la representación interna del ktree.

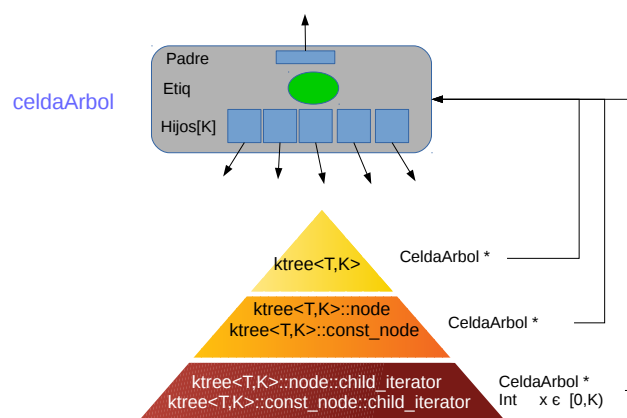


Figura 2: Atributos de la clase ktree

1.6. Representando un Nmer como un ktree

1.6.1. Representación

Podemos representar todos los **Nmer** hasta una determinada longitud mediante un ktree. Para ello, en cada nodo se almacenará un char que representa a una base (A,G,C,T), excepto el nodo raíz que almacena un char nulo, por ejemplo '-'.

El camino que existe desde la raíz del ktree hasta un nodo n en el nivel l codificará una subsecuencia de bases consecutivas en la cadena de tamaño l. Por ejemplo, supongamos la cadena "ATACATCATTGTGG". Las distintas subsecuencias (de tamaños 1 a 3) que nos pueden aparecer son:

- Tamaño 1: A; G; C; T
- Tamaño 2: AC; AT; GG; GT; CA; TA; TG; TC; TT
- Tamaño 3: ACA; ATA; ATC; ATT; GTG; CAT; TAC; TGG; TGT; TCA; TTG

Por tanto, cada nodo del árbol tendrá como máximo 4 descendientes directos (hijos) que representan las posibles bases que pueden aparecer en la cadena para formar las subsecuencias de tamaño l+1, asociando cada uno a una base distinta de la siguiente forma

- A -> hijo 0
- G -> hijo 1
- C -> hijo 2
- T -> hijo 3

No olvidemos que, además de identificar las subsecuencias, estamos interesados en contar el número de veces que se repita cada una de ellas en la cadena. Para ello, es suficiente con incluir también en cada nodo del árbol un valor entero que codifique dicha información. Así, la clase **Nmer** se puede representar como se indica:

```
class Nmer {
public:
    Nmer();
    ...
private:
    ktree<pair<char,int>,4> el_Nmer; // subsecuencias
    int max_long; // Mayor longitud de la cadena representada, esto es, el nivel máximo del árbol
};
```

La siguiente figura muestra un subconjunto del árbol para una longitud de subcadena máxima de 3 (max_long = 3) que se obtiene cuando consideramos la cadena "ATACATCATTGTGG", la lista todas las posibles subcadenas (junto a su frecuencia es: A 4; AC 1; ACA 1; AT 3; ATA 1; ATC 1; ATT 1; C 2; CA 2; CAT 2; G 3; GG 1; GT 1; GTG 1; T 5; TA 1; TAC 1; TC 1; TCA 1; TG 2; TGG 1; TGT 1; TT 1; TTG 1).

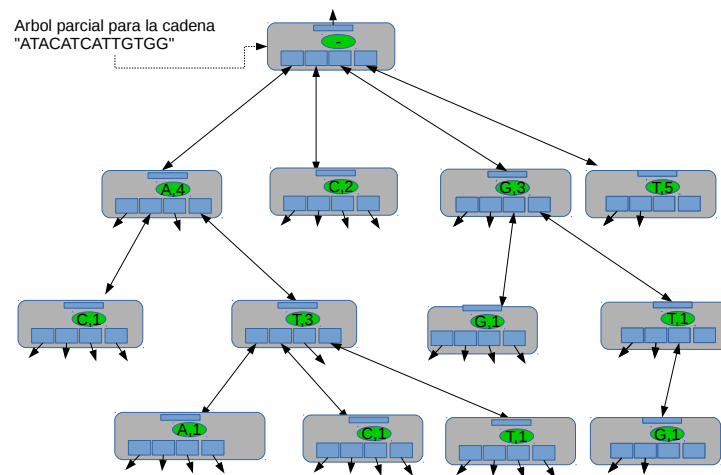


Figura 3: Cadena de ADN <ATACATCATTGTGG> representada como un ktree

1.7. ¿Cómo implementar algunos de los métodos en Nmer?

En esta sección presentaremos con mas detalle el comportamiento de algunos de los métodos de la clase `Nmer` que hay que implementar. Estos métodos permiten abordar distintos problemas de interés en genética computacional. Recordemos que son los métodos:

```
class Nmer {
    ...

    public:

        void sequenceADN(unsigned int tama, const string & adn); // Construir Nme a partir de cadena
            de ADN

        bool loadSerialized(const string nombre_fichero); // Leer fichero serializado que
            representa a un Nmer

        set<pair<string,int>,OrdenCre > rareNmer(int threshold); // devuelve la lista de todas las
            subcadenas (no prefijo) que aparecen menos de threshold veces en el Nmer ordenadas en orden creciente de
            frecuencia

        set<pair<string,int>,ordenDecre > commonNmer(int threshold); // Devuelve la lista de las
            cadenas de longitud mayor posible (no prefijo) que aparecen más de threshold veces en el Nmer, ordenadas en
            orden decreciente de frecuencia

        set<pair<string,int>, ordenCrec > level(int l); // Devuelve el conjunto de Nmers de longitud exacta
            l.

        float Distance(const Nmer & x); // Distancia entre dos Nmer

    ...
}
```

El alumno podrá dotar al tipo de dato de otros métodos que considere necesarios para la correcta realización de la práctica, pero debe respetar escrupulosamente la cabecera de los métodos detallados anteriormente.

1.7.1. Método 1 -> `Nmer::sequenceADN(tama, adn)` : Obteniendo los Nmers a partir de una cadena de ADN

El primer problema que nos tendremos que plantear a la hora de solucionar la práctica es el poder construir el ktree que representa los distintos Nmers. Como ya hemos comentado, utilizaremos un ktree con valor $k=4$, donde en cada nodo almacenamos no sólo la base, sino también su frecuencia dentro de la cadena de ADN. En cierto sentido, el nodo en el nivel i -ésimo representa un `Nmer` de longitud i (dicho `Nmer` se obtiene al considerar, en orden inverso, el camino que hay desde dicho nodo a la raíz del árbol). Recordad que en cada nodo almacenamos no sólo la base, sino también el contador que nos indica las ocurrencias del `Nmer` que representa el nodo en la secuencia de ADN.

En esta sección trataremos de ilustrar de forma detallada cómo podemos construir el árbol de Nmers (nuestro ktree) a partir de una secuencia de ADN. Para ello consideraremos dos métodos. El primero, `sequenceADN`, se encarga de recorrer la cadena de ADN para extraer una a una todas las subcadenas de longitud `tama` de las que se extraen los distintos Nmers a insertar. El segundo es un método privado, `insertar_cadena`, que, dada una subsecuencia de tamaño `tama`, se encarga de insertar los "tama" Nmers (subsecuencias que empiezan en `cadena[0]` y con longitud 1 hasta `cadena.size()`) en el árbol.

Por ejemplo, si `adn = "ATACATCATTGTGG"` y `tama` es 6, `sequenceADN` extraería las subsecuencias:

- "ATACAT"
- "TACATC"
- "ACATCA"
- "CATCAT"
- ...

En pseudo-código, `sequenceADN` podría implementarse como sigue:

```
void Nmer::sequenceADN(unsigned int tama, const string & adn){
    Inicializamos el árbol poniendo la etiqueta ('-',0) en el nodo raíz

    Para cada uno de las posiciones, i, de la cadena {
        Obtenemos un substring de tamaño tama que empiece en adn[i];
        insertar_cadena(subcadena);
    }
}
```

Si nos centramos en cada una de las cadenas, `insertar_cadena` permite considerar los Nmers que **empiezan** en `cadena.begin()`. Por ejemplo, considerando un tamaño hasta 6, los distintos Nmers para "ATACAT" considerados en este método serían:

1. "A"
2. "AT"
3. "ATA"
4. "ATAC"
5. "ATACA"
6. "ATACAT"

El esquema del algoritmo `insertar_cadena` es el siguiente: avanzar por la cadena a la vez que descendemos por el árbol para actualizar los valores del `Nmer` que representa, como indica el siguiente pseudo-código:

```
void insertar_cadena(const string & cadena){
    posicionamos un nodo, n_act, en la raíz del árbol.

    //Este nodo n lo utilizamos para descender por el árbol
    // cadena[i] nos indica el índice del nodo en el nivel i+1;
    // A -> hijo 0; G -> hijo 1; C-> hijo 2; T-> hijo 3

    Para cada uno de los caracteres, i, de la cadena {
        cadena[i] nos indica el índice del nodo en el nivel i+1;
        // si cadena[i] es : A -> hijo 0; G -> hijo 1; C-> hijo 2; T-> hijo 3
        si n_act tiene hijo en la posición dada por cadena[i] { // Ya esta insertado el nodo
            incrementamos el contador en 1;
        }
        en caso contrario { // No esta insertado el nodo
            insertamos el nuevo nodo en dicha posición con etiqueta cadena[i] y su contador a 1;
        }
        descendemos en el árbol, haciendo que n_act sea el nodo que representa cadena[i], esto es bajamos al
        hijo correspondiente;
    }
}
```

Para finalizar, os entregamos el código que nos recorre el ktree en preorden, donde por ejemplo la acción sobre el hijo podría ser obtener el **Nmer** que representa el nodo. En este algoritmo, n será el nodo a partir del cual queremos realizar el recorrido en preorden:

```
void recorrido_preorden( ktree<pair<char,int>,4>::node n){
    if (!n.null()){
        < accion sobre el nodo n >
        for ( ktree<base,4>::node hijo : n)
            recorrido_preorden(hijo);
    }
}
```

1.7.2. Método 2 -> Nmer::loadSerialized(nombre)

Por si el alumno tiene problema en la construcción del **Nmer**, se entrega este método ya implementado que, a partir de un fichero serializado (con extensión .srl), reconstruye el ktree que lo generó. Con la práctica se entregan algunos ejemplos de ficheros serializados que podrán ser utilizados por el alumno para obtener distintos objetos **Nmer** válidos (datos/cadenaSimple.srl, datos/cadenabosTaurus.srl o datos/cadenamus-Musculus.srl).

De este modo, podemos utilizar este método para ir avanzando en la implementación de los otros métodos de la clase incluso antes de tener implementado el método que construye el **Nmer** a partir de una secuencia (método sequenceADN).

Un ejemplo de cómo poder utilizar este método lo podemos encontrar en el fichero [ejemploKtree.cpp](#):

```
Nmer prueba;
prueba.loadSerialized("cadenaSimple.srl");
cout << prueba.length(); // devuelve 3
cout << prueba.size(); // devuelve 24
prueba.list_Nmer(); // Listaria todos los Nmers (desde longitud 1 hasta longitud prueba.length())
```

Sólo a nivel informativo, indicar que para poder reconvertir el fichero serializado necesitamos de un functor que nos permita transformar un string del tipo "A 45" o "x -1", donde el primer elemento de la cadena es un char y el segundo un entero, en un objeto de tipo pair<char,int>. Dicho functor, se encuentra dentro de la parte privada de la clase **Nmer** con el nombre String2Base.

El código para loadSerialized (ya incluido en el **Nmer.cpp**) es el siguiente:

```
bool Nmer::loadSerialized(const string & fichero) {
    string cadena;
    ifstream fe;
    pair<char,int> nulo('x',-1);
    Nmer::String2Base stb;

    fe.open(fichero.c_str(), ifstream::in);
    if (fe.fail()){
        cerr << "Error al abrir el fichero " << fichero << endl;
    }
    else {
        //leo la cabecera del fichero (líneas que comienzan con #)
        do{
            getline(fe,cadena,'\n');
        } while (cadena.find("#")==0 && !fe.eof());
        // leemos Nmer_length
        max_long = stoi(cadena);
        // leemos cadena serializada
        getline(fe,cadena,'\n');
        el_Nmer.deserialize(cadena,nulo,',',stb);
        fe.close();
        return true;
    } // else
    fe.close();
    return false;
}
```

1.7.3. Método 3 -> set<pair<string,int>,OrdenCre > Nmer::rareNmer(threshold)

Este método devuelve el conjunto de subcadenas (no prefijo) que aparecen un número de veces menor o igual a threshold veces en el **Nmer**. Cada cadena se almacenará en el primer campo del par, mientras que en el segundo

dispondremos de la frecuencia de aparición de la cadena. El conjunto deberá estar ordenado en orden creciente de frecuencia, esto es, la cadena menos frecuente aparecerá la primera en el set (esto resulta de utilidad, por ejemplo, para buscar peculiaridades en la secuencia de ADN).

¿Qué es una cadena prefijo?: decimos que una cadena, x , es prefijo de otra, y , si x está completamente incluida en y empezando desde el principio. Por ejemplo $x="abcde"$ es prefijo de $y="abcdefg"$ pero no lo es de $z="sabcdefg"$.

Por tanto, si realizamos una llamada al método `rareNmer(4)`, `threshold` vale 4, y encontramos que el `Nmer` "AC" aparece 5 veces, el `Nmer` "ACG" aparece 4 veces, el `Nmer` "ACGG" aparece 2 y "ACGT" aparece 1, la salida de `rareNmer` sólo debería devolver "ACGG" y "ACGT", pues "ACG", aun teniendo una frecuencia menor o igual que el `threshold` 4, es prefijo de otra cadena de longitud mayor que ya se devuelve como resultado.

1.7.4. Método 4 -> `set<pair<string,int>,ordenDecre > commonNmer(threshold)`

Devuelve el conjunto de las subcadenas de longitud mayor posible (sin ser prefijo) que aparecen un número de veces mayor que de `threshold` veces en el `Nmer`, ordenadas en orden decreciente de frecuencia (se utiliza para buscar repeticiones mas comunes, por ejemplo cuando tratamos de comprimir una cadena de ADN). Al igual que antes, el conjunto de cadenas deberá estar ordenado, pero en este caso en orden decreciente, esto es, aparecen primero las más frecuentes.

Por ejemplo, si consideramos que un `Nmer` contiene a "AC" que aparece 5 veces, a "AT" que aparece 5 veces, a "ACG" que aparece 4 veces, "ACT" que aparece 4 veces, "ATG" que aparece 3 veces "ACGG" que aparece 2 y "ACGT" que aparece 1, una llamada a `commonNmer(3)` dará como salida "AT", "ACG" y "ACT". En este caso "AC", aun teniendo una frecuencia mayor que el `threshold` 4, es prefijo de otra cadena de longitud mayor que ya se devuelve como resultado.

1.7.5. Método 5 -> `set<pair<string,int>,ordenCrec > level(l);`

Este método nos devuelve únicamente los Nmers de longitud l , ordenados en orden creciente de frecuencia.

Por ejemplo, si consideramos que un `Nmer` contiene a "AC" que aparece 5 veces, a "AT" que aparece 5 veces, a "ACG" que aparece 4 veces, "ACT" que aparece 4 veces, "ATG" que aparece 3 veces, "ACGG" que aparece 2 y "ACGT" que aparece 1, una llamada a `level(3)` dará como salida "ATG", "ACG" y "ACT", con sus frecuencias asociadas (3, 4 y 4, respectivamente).

1.7.6. Método 6 -> `float Distance(Nmer) : Aplicación del TDA Nmer a la caracterización del genoma de distintas especies.`

Una vez programado el TDA `Nmer` descrito en las secciones anteriores, proponemos que el estudiante aplique este TDA a un problema real: la identificación de especies en base a los Nmers de una secuencia de ADN. Para ello, se proporciona al estudiante extractos reales de longitud 10.000 de cadenas de ADN del genoma de 10 especies animales (ficheros `datos/1.txt`, `datos/2.txt`, ..., `datos/10.txt`). El estudiante debe construir un objeto `Nmer` para representar cada una de estas 10 secuencias de ADN. Una vez creados los Nmers, el estudiante deberá implementar una métrica para medir la distancia entre estos Nmers para identificar cuáles son las cadenas más parecidas entre sí. El fichero `datos/genomas disponibles.txt` muestra a qué especie corresponde cada cadena de ADN. Si los métodos están bien implementados y el tamaño de N es suficientemente elevado (recomendamos probar valores de $N \geq 10$), los resultados deben ir acordes a la distancia evolutiva entre las especies [3]: por ejemplo, las cadenas humanas serán más parecidas entre sí o a las de ratón que a las de gusano. En cualquier caso, nuestros resultados pueden fluctuar dado que estamos considerando extractos de longitud reducida (10.000 bases es un tamaño reducido dado el tamaño total de los genomas considerados).

1.7.6.1. Cálculo de la Distancia

En esta sección detallaremos cómo se calcula la distancia entre dos Nmers. Para ello consideremos el siguiente ejemplo, donde calcularemos la distancia entre dos Nmers, `nmX` e `nmY`, contruidos a partir de dos cadenas de ADN, `adnX` y `adnY`, respectivamente. Ilustraremos cómo sería al proceso para calcular:

```
nmX.distance(nmY);
```

En nuestro caso, asumiremos que ambos datos tienen como longitud máxima 2. Si representamos los Nmers como una secuencia de pares **Nmer**=frecuencia:

- nmX: (A=690) (G=1019) (C=818) (T=694) (AA=199) (AG=282) (AC=99) (AT=109) (GA=257) (GG=372) (GC=231) (GT=159) (CA=153) (CG=162) (CC=284) (CT=219) (TA=81) (TG=202) (TC=204) (TT=207)
- nmY: (A=985) (G=1202) (C=1262) (T=991) (AA=270) (AG=341) (AC=219) (AT=155) (GA=261) (GG=423) (GC=300) (GT=217) (CA=349) (CC=456) (CT=386) (TG=367) (TC=286) (TT=233)

El primer paso consiste en ordenar los distintos Nmers de forma que los más frecuentes ocupen las primeras posiciones en el orden. Una alternativa para abordar este problema es utilizar un contenedor asociativo, tipo set, que almacene cada uno de los pares pero considerando como criterio de comparación el de mayor frecuencia. Como resultado tendríamos esta secuencia de elementos (**Nmer** frecuencia).

- nmX: G 1019; C 818; T 694; A 690; GG 372; CC 284; AG 282; GA 257; GC 231; CT 219; TT 207; TC 204; TG 202; A 199; CG 162; GT 159; CA 153; AT 109; AC 99; TA 81;
- nmY: C 1262; G 1202; T 991; A 985; CC 456; GG 423; CT 386; TG 367; CA 349; AG 341; GC 300; TC 286; AA 270; GA 261; TT 233; AC 219; GT 217; AT 155;

Iterando sobre este contenedor es fácil asociar a cada **Nmer** su posición en el ranking. Así para adnX el **Nmer** G tendrá valor 1, **Nmer** C tendrá valor 2, **Nmer** T será 3, ... Esto es,

- rankingX: G 1; C 2; T 3; A 4; GG 5; CC 6; AG 7; GA 8; GC 9; CT 10; TT 11; TC 12; TG 13; AA 14; CG 15; GT 16; CA 17; AT 18; AC 19; TA 20;
- rankingY: C 1; G 2; T 3; A 4; CC 5; GG 6; CT 7; TG 8; CA 9; AG 10; GC 11; TC 12; AA 13; GA 14; TT 15; AC 16; GT 17; AT 18;

Una vez que tenemos este ranking podremos proceder a calcular la distancia teniendo en cuenta la distancia entre las posiciones en cada ranking. Así, por ejemplo G está a distancia 1 (diferencia entre 1 y 2), T está a distancia 0, TG está a distancia 5 (diferencia entre 13 y 8), ... y finalmente si consideramos tanto CG como TA diremos que están a distancia máxima (definida como la distancia máxima posible entre los rankings) ya que ambos Nmers no se encuentran en el rankingY.

Estas distancias (normalizadas) son acumuladas y debemos devolver el promedio.

El siguiente pseudo código nos permite computar dicha distancia:

```
max <- maximo(rankingX.size(), rankingY.size());
dist <- 0;
Para cada Nmer n en rankingX {
  Si (n pertenece a rankingY) {
    posX <- la posición de n en rankingX
    posY <- la posición de n en rankingY;
    valor = abs(posX-posY)/max
  } else valor = 1;
  dist+=valor;
}
return dist/rankingX.size() // calculamos el promedio, tomando valores en [0,1]
```

Finalmente, indicar que la distancia, tal y como está definida no es simétrica, esto es nmX.distance(nmY) != nmY.distance(nmX)

1.8. Entrega

El alumno debe empaquetar todos los archivos relacionados con el proyecto en un archivo con nombre "genData.zip". Tenga en cuenta que no se incluirán ficheros objeto, ni ejecutables. Es recomendable que haga una "limpieza" para eliminar los archivos temporales o que se puedan generar a partir de los fuentes.

El alumno debe incluir un fichero con un main src/ejemploNmer.cpp que exhiba toda la funcionalidad programada para el TDA **Nmer**, incluyendo el cálculo de las distancias entre las 10 secuencias de distinta especie (datos/1.txt, datos/2.txt, etc).

El alumno ya tiene disponible un fichero Makefile para realizar la compilación. Conviene estudiar con detalle este fichero y sus distintas opciones:

- `make` para generar los ejecutables
- `make documentacion` para generar la documentación en la carpeta `doc/html` y `doc/latex`
- `make clean` para eliminar ficheros temporales y objeto.

El alumno debe actualizar este fichero para compilar los nuevos módulos generados.

Tenga en cuenta que los archivos deben estar distribuidos en directorios:

genData.zip

- Makefile
- include – Carpeta con ficheros de cabecera (.h)
- src –Carpeta con código fuente (.cpp)
- doc –Carpeta con Documentación
- obj – Carpeta para código objeto (.o)
- lib – Carpeta para bibliotecas (.a)
- bin – Carpeta para ejecutables
- datos – Carpeta para ficheros de datos

La fecha límite de entrega es el día 25 de Enero de 2017 a las 23:59.

1.9. Bibliografía

[1] k-mer en wikipedia: <https://en.wikipedia.org/wiki/K-mer>

[2] Genomic DNA k-mer spectra: models and modalities. (2009) Benny Chor et al. Genome Biol. 10(10): R108.

[3] <https://genome-euro.ucsc.edu/cgi-bin/hgGateway>

2. Lista de tareas pendientes

Miembro `Nmer::list_Nmer()` const

Debemos de implementar este método de forma que nos garantizemos que se imprimen todos los Nmers.

3. Lista de bugs

Archivo `ktree.h`

Por especificar

Archivo `Nmer.h`

Por especificar

4. Índice de clases

4.1. Lista de clases

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

<code>ktree< T, K >::node::child_iterator</code>	14
<code>ktree< T, K >::const_node::child_iterator</code>	15
<code>ktree< T, K >::const_node</code>	16
<code>ktree< T, K ></code>	19
<code>Nmer</code>	25
<code>ktree< T, K >::node</code>	28

5. Índice de archivos

5.1. Lista de archivos

Lista de todos los archivos con descripciones breves:

<code>include/ktree.h</code>	
TDA <code>ktree<T,K></code> Representa un árbol general con nodos etiquetados con datos del tipo T. Cada nodo puede tener entre 0 y K hijos como máximo, numerados de 0 a K-1	31
<code>include/Nmer.h</code>	
TDA <code>Nmer</code> Representa un conjunto de <code>Nmer</code> subsecuencias de tamaño 1 hasta N que se pueden obtener a partir de una cadena de ADN	36
<code>src/ejemploKtree.cpp</code>	36
<code>src/Nmer.cpp</code>	38

6. Documentación de las clases

6.1. Referencia de la Clase `ktree< T, K >::node::child_iterator`

```
#include <ktree.h>
```

Métodos públicos

- `child_iterator ()`
- `child_iterator (const child_iterator &x)`
- `node & operator* ()`
- `child_iterator & operator++ ()`
- `child_iterator operator++ (int)`
- `child_iterator & operator= (const child_iterator &x)`
- `bool operator== (const child_iterator &x)`
- `bool operator!= (const child_iterator &x)`

Amigas

- class `node`

6.1.1. Descripción detallada

```
template<typename T, int K = 2>class ktree< T, K >::node::child_iterator
```

Definición en la línea 420 del archivo `ktree.h`.

6.1.2. Documentación del constructor y destructor

6.1.2.1. `template<typename T, int K = 2> ktree< T, K >::node::child_iterator::child_iterator ()`

6.1.2.2. `template<typename T, int K = 2> ktree< T, K >::node::child_iterator::child_iterator (const child_iterator & x)`

6.1.3. Documentación de las funciones miembro

6.1.3.1. `template<typename T, int K = 2> bool ktree< T, K >::node::child_iterator::operator!= (const child_iterator & x)`

6.1.3.2. `template<typename T, int K = 2> node& ktree< T, K >::node::child_iterator::operator* ()`

6.1.3.3. `template<typename T, int K = 2> child_iterator& ktree< T, K >::node::child_iterator::operator++ ()`

6.1.3.4. `template<typename T, int K = 2> child_iterator ktree< T, K >::node::child_iterator::operator++ (int)`

6.1.3.5. `template<typename T, int K = 2> child_iterator& ktree< T, K >::node::child_iterator::operator= (const child_iterator & x)`

6.1.3.6. `template<typename T, int K = 2> bool ktree< T, K >::node::child_iterator::operator== (const child_iterator & x)`

6.1.4. Documentación de las funciones relacionadas y clases amigas

6.1.4.1. `template<typename T, int K = 2> friend class node [friend]`

Definición en la línea 422 del archivo `ktree.h`.

La documentación para esta clase fue generada a partir del siguiente fichero:

- `include/ktree.h`

6.2. Referencia de la Clase `ktree< T, K >::const_node::child_iterator`

```
#include <ktree.h>
```

Métodos públicos

- `child_iterator ()`
- `child_iterator (const child_iterator &it)`
- `child_iterator (const typename ktree< T, K >::node::child_iterator &x)`
- `const_node operator* () const`
- `child_iterator & operator++ ()`
- `child_iterator operator++ (int)`
- `child_iterator & operator= (const child_iterator &x)`
- `bool operator== (const child_iterator &x)`
- `bool operator!= (const child_iterator &x)`

Amigas

- `class const_node`

6.2.1. Descripción detallada

```
template<typename T, int K = 2>class ktree< T, K >::const_node::child_iterator
```

Definición en la línea 533 del archivo ktree.h.

6.2.2. Documentación del constructor y destructor

6.2.2.1. `template<typename T, int K = 2> ktree< T, K >::const_node::child_iterator::child_iterator ()`

6.2.2.2. `template<typename T, int K = 2> ktree< T, K >::const_node::child_iterator::child_iterator (const child_iterator & it)`

6.2.2.3. `template<typename T, int K = 2> ktree< T, K >::const_node::child_iterator::child_iterator (const typename ktree< T, K >::node::child_iterator & x)`

6.2.3. Documentación de las funciones miembro

6.2.3.1. `template<typename T, int K = 2> bool ktree< T, K >::const_node::child_iterator::operator!= (const child_iterator & x)`

6.2.3.2. `template<typename T, int K = 2> const_node ktree< T, K >::const_node::child_iterator::operator* () const`

6.2.3.3. `template<typename T, int K = 2> child_iterator& ktree< T, K >::const_node::child_iterator::operator++ ()`

6.2.3.4. `template<typename T, int K = 2> child_iterator ktree< T, K >::const_node::child_iterator::operator++ (int)`

6.2.3.5. `template<typename T, int K = 2> child_iterator& ktree< T, K >::const_node::child_iterator::operator= (const child_iterator & x)`

6.2.3.6. `template<typename T, int K = 2> bool ktree< T, K >::const_node::child_iterator::operator== (const child_iterator & x)`

6.2.4. Documentación de las funciones relacionadas y clases amigas

6.2.4.1. `template<typename T, int K = 2> friend class const_node [friend]`

Definición en la línea 535 del archivo ktree.h.

La documentación para esta clase fue generada a partir del siguiente fichero:

- `include/ktree.h`

6.3. Referencia de la Clase `ktree< T, K >::const_node`

```
#include <ktree.h>
```

Clases

- class `child_iterator`

Métodos públicos

- `const_node ()`
Constructor primitivo.
- `const_node (const typename ktree< T, K >::node &n)`

Constructor para convertir node a [const_node](#).

- [const_node](#) (const [const_node](#) &n)

Constructor de copia.

- bool [null](#) () const

Determina si el nodo es nulo.

- [const_node parent](#) () const

Devuelve el padre del nodo receptor.

- [const_node k_child](#) (int k) const

Devuelve el hijo k-ésimo del nodo receptor.

- [const_node operator\[\]](#) (int k) const

Devuelve el hijo k-ésimo del nodo receptor.

- const T & [operator*](#) () const

Devuelve la referencia constante a la etiqueta del nodo.

- int [child_number](#) () const

Devuelve el numero de descendiente de un hijo con respecto a su padre.

- [const_node & operator=](#) (const [const_node](#) &n)

Operador de asignación.

- bool [operator==](#) (const [const_node](#) &n) const

Operador de comparación de igualdad.

- bool [operator!=](#) (const [const_node](#) &n) const

Operador de comparación de desigualdad.

- [child_iterator begin](#) () const

- [child_iterator cbegin](#) () const

- [child_iterator end](#) () const

- [child_iterator cend](#) () const

Amigas

- class [child_iterator](#)

6.3.1. Descripción detallada

```
template<typename T, int K = 2>class ktree< T, K >::const_node
```

Definición en la línea 453 del archivo ktree.h.

6.3.2. Documentación del constructor y destructor

6.3.2.1. `template<typename T, int K = 2> ktree< T, K >::const_node::const_node ()`

Constructor primitivo.

6.3.2.2. `template<typename T, int K = 2> ktree< T, K >::const_node::const_node (const typename ktree< T, K >::node & n)`

Constructor para convertir node a [const_node](#).

Parámetros

<i>n</i>	Nodo que se copia
----------	-------------------

6.3.2.3. `template<typename T, int K = 2> ktree< T, K >::const_node::const_node (const const_node & n)`

Constructor de copia.

Parámetros

<i>n</i>	Nodo que se copia
----------	-------------------

6.3.3. Documentación de las funciones miembro

6.3.3.1. `template<typename T, int K = 2> child_iterator ktree< T, K >::const_node::begin () const`

6.3.3.2. `template<typename T, int K = 2> child_iterator ktree< T, K >::const_node::cbegin () const`

6.3.3.3. `template<typename T, int K = 2> child_iterator ktree< T, K >::const_node::cend () const`

6.3.3.4. `template<typename T, int K = 2> int ktree< T, K >::const_node::child_number () const`

Devuelve el numero de descendiente de un hijo con respecto a su padre.

Devuelve

un valor entre 0 y k-1 correspondiente a la posicion, o -1 si es el nodo raiz

6.3.3.5. `template<typename T, int K = 2> child_iterator ktree< T, K >::const_node::end () const`

6.3.3.6. `template<typename T, int K = 2> const_node ktree< T, K >::const_node::k_child (int k) const`

Devuelve el hijo k-ésimo del nodo receptor.

Precondición

`!null()`

6.3.3.7. `template<typename T, int K = 2> bool ktree< T, K >::const_node::null () const`

Determina si el nodo es nulo.

6.3.3.8. `template<typename T, int K = 2> bool ktree< T, K >::const_node::operator!= (const const_node & n) const`

Operador de comparación de desigualdad.

Parámetros

<i>n</i>	el nodo con el que se compara
----------	-------------------------------

6.3.3.9. `template<typename T, int K = 2> const T& ktree< T, K >::const_node::operator* () const`

Devuelve la referencia constante a la etiqueta del nodo.

Precondición

`!null()`

6.3.3.10. `template<typename T, int K = 2> const_node& ktree< T, K >::const_node::operator= (const const_node & n)`

Operador de asignación.

Parámetros

<i>n</i>	el nodo a asignar
----------	-------------------

6.3.3.11. `template<typename T, int K = 2> bool ktree< T, K >::const_node::operator==(const const_node & n) const`

Operador de comparación de igualdad.

Parámetros

<i>n</i>	el nodo con el que se compara
----------	-------------------------------

6.3.3.12. `template<typename T, int K = 2> const_node ktree< T, K >::const_node::operator[] (int k) const`

Devuelve el hijo k-ésimo del nodo receptor.

Precondición

`!null()`

6.3.3.13. `template<typename T, int K = 2> const_node ktree< T, K >::const_node::parent () const`

Devuelve el padre del nodo receptor.

Precondición

`!null()`

6.3.4. Documentación de las funciones relacionadas y clases amigas

6.3.4.1. `template<typename T, int K = 2> friend class child_iterator [friend]`

Definición en la línea 528 del archivo ktree.h.

La documentación para esta clase fue generada a partir del siguiente fichero:

- `include/ktree.h`

6.4. Referencia de la plantilla de la Clase ktree< T, K >

`#include <ktree.h>`

Clases

- class `const_node`
- class `node`

Tipos públicos

- `typedef unsigned int size_type`

Métodos públicos

- `ktree ()`
Constructor primitivo por defecto. Crea un árbol nulo.
- `ktree (const T &e)`

- Constructor primitivo.
- `ktree` (const `ktree`< T, K > &a)
- Constructor de copia.
- `~ktree` ()
- Destructor. Destruye el receptor liberando los recursos que ocupaba.
- `ktree`< T, K > & `operator=` (const `ktree`< T, K > &a)
- Operador de asignación.
- `const_node root` () const
- Obtener la raíz como nodo constante. Obtener el nodo raíz.
- `node root` ()
- Obtener el nodo raíz.
- void `prune_k_child` (`node` n, int k, `ktree`< T, K > &dest)
- Podar el subárbol localizado en el hijo k-ésimo de un nodo.
- void `insert_k_child` (`ktree`< T, K >::`node` &n, int k, const T &e)
- Insertar un nodo como hijo k-ésimo de un nodo n.
- void `insert_k_child` (`node` n, int k, `ktree`< T, K > &rama)
- Insertar un árbol como hijo k-ésimo de un nodo.
- void `assing` (const `ktree`< T, K > &org, `node` n)
- asignar subarbol. Hace del receptor una copia del subarbol de org con raíz en n
- void `assing` (const T &e)
- asignar un nodo. Hace que el receptor sea un árbol con un único nodo con etiqueta e
- void `clear` ()
- Hace nulo un árbol. Destruye todos los nodos del árbol receptor y lo hace un árbol nulo.
- `size_type size` () const
- Obtiene el número de nodos.
- bool `empty` () const
- Comprueba si un árbol está vacío (es nulo).
- bool `operator==` (const `ktree`< T, K > &a) const
- Operador de comparación de igualdad.
- bool `operator!=` (const `ktree`< T, K > &a) const
- Operador de comparación de desigualdad.
- template<class toString >
- string `serialize` (const T &valor_nulo, char delim, toString T2Str) const
 - Serializa el arbol.
- template<class String2T >
- void `deserialize` (const string &ser, const T &valor_nulo, char delim, String2T Str2T)
 - Reconstruye el árbol a partir de una serialización válida.
- void `recorrido_preorden` (`const_node` n)

Atributos públicos estáticos

- static const int `kvalue` = K
- Valor del número de descendientes. Es constante, tomando el valor por defecto de 2, árbol binario.

6.4.1. Descripción detallada

```
template<typename T, int K = 2>class ktree< T, K >
```

Definición en la línea 40 del archivo ktree.h.

6.4.2. Documentación de los 'Typedef' miembros de la clase

6.4.2.1. `template<typename T, int K = 2> typedef unsigned int ktree< T, K >::size_type`

Definición en la línea 43 del archivo ktree.h.

6.4.3. Documentación del constructor y destructor

6.4.3.1. `template<typename T, int K = 2> ktree< T, K >::ktree ()`

Constructor primitivo por defecto. Crea un árbol nulo.

6.4.3.2. `template<typename T, int K = 2> ktree< T, K >::ktree (const T & e)`

Constructor primitivo.

Parámetros

<i>e</i>	Etiqueta para la raíz. Crea un árbol con un único nodo etiquetado con e y sus k hijos todos nulos
----------	---

6.4.3.3. `template<typename T, int K = 2> ktree< T, K >::ktree (const ktree< T, K > & a)`

Constructor de copia.

Parámetros

<i>a</i>	árbol que se copia. Crea un árbol duplicado exacto de a.
----------	--

6.4.3.4. `template<typename T, int K = 2> ktree< T, K >::~~ktree ()`

Destructor. Destruye el receptor liberando los recursos que ocupaba.

6.4.4. Documentación de las funciones miembro

6.4.4.1. `template<typename T, int K = 2> void ktree< T, K >::assing (const ktree< T, K > & org, node n)`

asignar subarbol. Hace del receptor una copia del subarbol de org con raíz en n

Parámetros

<i>n</i>	nodo de org a partir del que se copia. n != nodo_nulo.
<i>org</i>	Arbol original (no se modifica).

Si el arbol receptor contiene elementos, estos son eliminados.

6.4.4.2. `template<typename T, int K = 2> void ktree< T, K >::assing (const T & e)`

asignar un nodo. Hace que el receptor sea un árbol con un único nodo con etiqueta e

Parámetros

<i>e</i>	Etiqueta a asignar
----------	--------------------

Si el arbol receptor contiene elementos previamente, éstos son eliminados.

6.4.4.3. `template<typename T, int K = 2> void ktree< T, K >::clear ()`

Hace nulo un árbol. Destruye todos los nodos del árbol receptor y lo hace un árbol nulo.

6.4.4.4. `template<typename T, int K = 2> template<class String2T > void ktree< T, K >::deserialize (const string & ser,
const T & valor_nulo, char delim, String2T Str2T)`

Reconstruye el árbol a partir de una serialización válida.

Parámetros

<i>ser</i>	cadena que representa la serialización
<i>valor_nulo</i>	valor nulo para el parámetro de tipo T
<i>delim</i>	delimitar utilizado para separar valores en el árbol
<i>Str2T</i>	functor que recibe un string, s, y lo convierte en un valor de tipo T correcto.

Precondición

el string *ser* ha sido obtenido mediante el proceso `serialize(valor_nulo,delim,T2Str)`
`Str2T` es el proceso inverso de `T2Str`

Genera el `ktree<T,K>` que dió origen a la serialización representada por *ser*.

6.4.4.5. `template<typename T, int K = 2> bool ktree< T, K >::empty () const`

Comprueba si un árbol está vacío (es nulo).

Devuelve

true, si el receptor está vacío (es nulo). false, en otro caso.

6.4.4.6. `template<typename T, int K = 2> void ktree< T, K >::insert_k_child (ktree< T, K >::node & n, int k, const T & e)`

Insertar un nodo como hijo k-ésimo de un nodo n.

Parámetros

<i>n</i>	nodo del receptor. <code>!n.null()</code> .
<i>k</i>	número de hijo de n
<i>e</i>	etiqueta del nuevo nodo.

Inserta un nuevo nodo con etiqueta e como hijo k-ésimo de n. Si n tuviese al menos un descendiente en dicha posición lo desconecta y destruye el subárbol.

6.4.4.7. `template<typename T, int K = 2> void ktree< T, K >::insert_k_child (node n, int k, ktree< T, K > & rama)`

Insertar un árbol como hijo k-ésimo de un nodo.

Parámetros

<i>n</i>	nodo del receptor. <code>n != nodo_nulo</code> .
<i>k</i>	número de hijo de n
<i>rama</i>	subárbol que se inserta. Es MODIFICADO quedando como árbol vacío tras la inserción-

Inserta la rama como subárbol en la posición k-ésima del nodo n. Si hubiese un subárbol en esa posición lo destruye. Además, rama se hace árbol nulo.

6.4.4.8. `template<typename T, int K = 2> bool ktree< T, K >::operator!= (const ktree< T, K > & a) const`

Operador de comparación de desigualdad.

Parámetros

<i>a</i>	árbol con que se compara el receptor.
----------	---------------------------------------

Devuelve

true, si el receptor no es igual, en estructura o etiquetas a a. false, en otro caso.

6.4.4.9. `template<typename T, int K = 2> ktree<T,K>& ktree< T, K >::operator= (const ktree< T, K > & a)`

Operador de asignación.

Parámetros

<i>a</i>	árbol que se asigna.
----------	----------------------

Destruye el contenido previo del receptor y le asigna un duplicado de *a*.

6.4.4.10. `template<typename T, int K = 2> bool ktree< T, K >::operator== (const ktree< T, K > & a) const`

Operador de comparación de igualdad.

Parámetros

<i>a</i>	árbol con que se compara el receptor.
----------	---------------------------------------

Devuelve

true, si el receptor es igual, en estructura y etiquetas a *a*. false, en otro caso.

6.4.4.11. `template<typename T, int K = 2> void ktree< T, K >::prune_k_child (node n, int k, ktree< T, K > & dest)`

Podar el subárbol localizado en el hijo *k*-ésimo de un nodo.

Parámetros

<i>n</i>	nodo del receptor. !n.null()
<i>k</i>	número de hijo de <i>n</i>
<i>dest</i>	árbol destino donde se mueve el subárbol. Es MODIFICADO.

Desconecta el subárbol localizado en el hijo *k*-ésimo de *n*, que pasa a ser un árbol nulo en el receptor. El subárbol anterior se devuelve sobre *dest*.

6.4.4.12. `template<typename T, int K = 2> void ktree< T, K >::recorrido_preorden (const_node n)`

6.4.4.13. `template<typename T, int K = 2> const_node ktree< T, K >::root () const`

Obtener la raíz como nodo constante. Obtener el nodo raíz.

Devuelve

nodo raíz del receptor.

6.4.4.14. `template<typename T, int K = 2> node ktree< T, K >::root ()`

Obtener el nodo raíz.

Devuelve

nodo raíz del receptor.

6.4.4.15. `template<typename T, int K = 2> template<class toString> string ktree< T, K >::serialize (const T & valor_nulo, char delim, toString T2Str) const`

Serializa el árbol.

Parámetros

<i>valor_nulo</i>	valor nulo para el parámetro de tipo T
<i>delim</i>	delimitar utilizado para separar valores en el árbol
<i>T2Str</i>	functor que recibe un valor de tipo T y lo convierte en string

Devuelve

cadena que representa el árbol serializado, obtenido al realizar un recorrido por niveles del mismo (donde los nodos nulos se representan con *valor_nulo*)

6.4.4.16. `template<typename T, int K = 2> size_type ktree< T, K >::size () const`

Obtiene el número de nodos.

Devuelve

número de nodos del receptor.

6.4.5. Documentación de los datos miembro

6.4.5.1. `template<typename T, int K = 2> const int ktree< T, K >::kvalue = K [static]`

Valor del número de descendientes. Es constante, tomando el valor por defecto de 2, árbol binario.

`ktree<char, 5>::kvalue`

Definición en la línea 57 del archivo ktree.h.

La documentación para esta clase fue generada a partir del siguiente fichero:

- `include/ktree.h`

6.5. Referencia de la Clase Nmer

`#include <Nmer.h>`

Tipos públicos

- `typedef unsigned int size_type`

Métodos públicos

- `Nmer ()`
Constructor primitivo . Crea un Nmer de longitud maxima 0, con el valor ('-',0) en la raíz.
- `bool loadSerialized (const string &nombre_fichero)`
lectura fichero serializado
- `void list_Nmer () const`
Imprime los Nmers.
- `unsigned int length () const`
Máxima longitud de los Nmers almacenados.
- `ktree< pair< char, int > , 4 >::const_node root () const`
Devuelve el_Nmer.root()
- `size_type size () const`
Número de Nmers almacenados.
- `int altura_nodo (ktree< pair< char, int > , 4 >::const_node n) const`
Devuelve la altura del nodo argumentado.
- `void sequenceADN (unsigned int tama, const string &adn)`
Recorre la cadena de ADN para extraer subcadenas de longitud tama e insertarlas en el Nmer.
- `void recorrer_niveles () const`
Devuelve el conjunto de subcadenas que aparecen un numero de veces.
- `set< pair< string, int > > level (int l)`
Devuelve únicamente los Nmers de longitud ñ

- **Nmer Prefix** (string cadena)

*Devuelve el **Nmer** asociado a un prefijo.*

- bool **comparison** (typename **ktree**< pair< char, int >, 4 >::const_node n1, typename **ktree**< pair< char, int >, 4 >::const_node n2) const

- bool **containsString** (const string adn)

*Devuelve si la cadena esta contenida en alguna del **Nmer**.*

- bool **included** (const **Nmer** reference) const

*Devuelve true si cada nodo de *this está también representado en reference.*

- set< pair< string, int > > **rareNmer** (int threshold)

*Devuelve la lista de todas las subcadenas que aparecen menos de threshold veces en el **Nmer** ordenadas crecientemente.*

- set< pair< string, int > > **commonNmer** (int threshold)

*Devuelve la lista de las cadenas de longitud mayor posible que aparecen más de threshold veces en el **Nmer**, en orden decreciente.*

6.5.1. Descripción detallada

Definición en la línea 21 del archivo Nmer.h.

6.5.2. Documentación de los 'Typedef' miembros de la clase

6.5.2.1. typedef unsigned int **Nmer::size_type**

Definición en la línea 23 del archivo Nmer.h.

6.5.3. Documentación del constructor y destructor

6.5.3.1. **Nmer::Nmer** ()

Constructor primitivo . Crea un **Nmer** de longitud maxima 0, con el valor ('-',0) en la raíz.

Definición en la línea 8 del archivo Nmer.cpp.

6.5.4. Documentación de las funciones miembro

6.5.4.1. int **Nmer::altura_nodo** (**ktree**< pair< char, int >, 4 >::const_node *n*) const

Devuelve la altura del nodo argumentado.

Parámetros

<i>n</i>	nodo a calcular su altura
----------	---------------------------

Definición en la línea 234 del archivo Nmer.cpp.

6.5.4.2. set< pair< string, int > > **Nmer::commonNmer** (int *threshold*)

Devuelve la lista de las cadenas de longitud mayor posible que aparecen más de threshold veces en el **Nmer**, en orden decreciente.

Parámetros

<i>threshold</i>	numero de veces
------------------	-----------------

Definición en la línea 153 del archivo Nmer.cpp.

6.5.4.3. `bool Nmer::comparison (typename ktree< pair< char, int >, 4 >::const_node n1, typename ktree< pair< char, int >, 4 >::const_node n2) const`

Definición en la línea 213 del archivo Nmer.cpp.

6.5.4.4. `bool Nmer::containsString (const string adn)`

Devuelve si la cadena esta contenida en alguna del [Nmer](#).

Parámetros

<i>adn</i>	string a buscar en el Nmer
------------	--

Definición en la línea 125 del archivo Nmer.cpp.

6.5.4.5. `bool Nmer::included (const Nmer reference) const`

Devuelve true si cada nodo de *this está también representado en reference.

Parámetros

<i>reference</i>	Nmer donde comparar nodos
------------------	---

Definición en la línea 207 del archivo Nmer.cpp.

6.5.4.6. `unsigned int Nmer::length () const`

Máxima longitud de los Nmers almacenados.

Definición en la línea 48 del archivo Nmer.cpp.

6.5.4.7. `set< pair< string, int > > Nmer::level (int l)`

Devuelve únicamente los Nmers de longitud ñ

Parámetros

<i>l</i>	longitud
----------	----------

Definición en la línea 296 del archivo Nmer.cpp.

6.5.4.8. `void Nmer::list_Nmer () const`

Imprime los Nmers.

Tareas pendientes Debemos de implementar este método de forma que nos garantizemos que se imprimen todos los Nmers.

Definición en la línea 41 del archivo Nmer.cpp.

6.5.4.9. `bool Nmer::loadSerialized (const string & nombre_fichero)`

lectura fichero serializado

Parámetros

<i>nombre_fichero</i>	fichero serializado con extension .srl
-----------------------	--

Genera a partir del fichero el ktree que contiene todos los kmers asociados a una cadena de ADN

La cadena original viene descrita en el fichero serializado

Definición en la línea 14 del archivo Nmer.cpp.

6.5.4.10. `Nmer Nmer::Prefix (string cadena)`

Devuelve el [Nmer](#) asociado a un prefijo.

Parámetros

<i>cadena</i>	string a buscar de prefijo
---------------	----------------------------

Definición en la línea 60 del archivo Nmer.cpp.

6.5.4.11. `set< pair< string, int > > Nmer::rareNmer (int threshold)`

Devuelve la lista de todas las subcadenas que aparecen menos de *threshold* veces en el [Nmer](#) ordenadas crecientemente.

Parámetros

<i>threshold</i>	numero de veces
------------------	-----------------

Definición en la línea 141 del archivo Nmer.cpp.

6.5.4.12. `void Nmer::recorrer_niveles () const`

Devuelve el conjunto de subcadenas que aparecen un numero de veces.

Parámetros

<i>threshold</i>	numero de veces que aparecen las subcadenas
------------------	---

Devuelve el conjunto de subcadenas (no prefijo) que aparecen un número de veces menor o igual a *threshold* veces en el [Nmer](#).

Definición en la línea 262 del archivo Nmer.cpp.

6.5.4.13. `ktree< pair< char, int >, 4 >::const_node Nmer::root () const`

Devuelve el `_Nmer.root()`

Definición en la línea 247 del archivo Nmer.cpp.

6.5.4.14. `void Nmer::sequenceADN (unsigned int tama, const string & adn)`

Recorre la cadena de ADN para extraer subcadenas de longitud *tama* e insertarlas en el [Nmer](#).

Parámetros

<i>tama</i>	tamaño de las subcadenas del ADN a extraer
<i>adn</i>	cadena de ADN de la cual extraer subcadenas

Definición en la línea 89 del archivo Nmer.cpp.

6.5.4.15. `Nmer::size_type Nmer::size () const`

Número de Nmers almacenados.

Definición en la línea 52 del archivo Nmer.cpp.

La documentación para esta clase fue generada a partir de los siguientes ficheros:

- `include/Nmer.h`
- `src/Nmer.cpp`

6.6. Referencia de la Clase `ktree< T, K >::node`

```
#include <ktree.h>
```

Clases

- class [child_iterator](#)

Métodos públicos

- `node ()`
Constructor primitivo.
- `node (const node &n)`
Constructor de copia.
- `bool null () const`
Determina si el nodo es nulo.
- `node parent () const`
Devuelve el padre del nodo receptor.
- `node k_child (int k) const`
Devuelve el hijo k-ésimo del nodo receptor.
- `node operator[] (int k) const`
Devuelve el hijo k-ésimo del nodo receptor.
- `T & operator* ()`
Devuelve una referencia a la etiqueta del nodo.
- `const T & operator* () const`
Devuelve la referencia constante a la etiqueta del nodo.
- `int child_number () const`
Devuelve el número de descendiente de un hijo con respecto a su padre.
- `node & operator= (const node &n)`
Operador de asignación.
- `bool operator== (const node &n) const`
Operador de comparación de igualdad.
- `bool operator!= (const node &n) const`
Operador de comparación de desigualdad.
- `child_iterator begin ()`
- `child_iterator end ()`

Amigas

- `class ktree< T, K >`
- `class child_iterator`

6.6.1. Descripción detallada

```
template<typename T, int K = 2>class ktree< T, K >::node
```

TDA nodo. Modela los nodos del árbol binario.

Definición en la línea 315 del archivo `ktree.h`.

6.6.2. Documentación del constructor y destructor

6.6.2.1. `template<typename T, int K = 2> ktree< T, K >::node::node ()`

Constructor primitivo.

6.6.2.2. `template<typename T, int K = 2> ktree< T, K >::node::node (const node & n)`

Constructor de copia.

Parámetros

<i>n</i>	Nodo que se copia
----------	-------------------

6.6.3. Documentación de las funciones miembro

6.6.3.1. `template<typename T, int K = 2> child_iterator ktree< T, K >::node::begin ()`

6.6.3.2. `template<typename T, int K = 2> int ktree< T, K >::node::child_number () const`

Devuelve el numero de descendiente de un hijo con respecto a su padre.

Devuelve

un valor entre 0 y k-1 correspondiente a la posicion, o -1 si es el nodo raiz

6.6.3.3. `template<typename T, int K = 2> child_iterator ktree< T, K >::node::end ()`

6.6.3.4. `template<typename T, int K = 2> node ktree< T, K >::node::k_child (int k) const`

Devuelve el hijo k-ésimo del nodo receptor.

Precondición

`!null()`

6.6.3.5. `template<typename T, int K = 2> bool ktree< T, K >::node::null () const`

Determina si el nodo es nulo.

6.6.3.6. `template<typename T, int K = 2> bool ktree< T, K >::node::operator!= (const node & n) const`

Operador de comparación de desigualdad.

Parámetros

<i>n</i>	el nodo con el que se compara
----------	-------------------------------

6.6.3.7. `template<typename T, int K = 2> T& ktree< T, K >::node::operator* ()`

Devuelve una referencia a la etiqueta del nodo.

Precondición

Si se usa como consultor, `!null()`

6.6.3.8. `template<typename T, int K = 2> const T& ktree< T, K >::node::operator* () const`

Devuelve la referecia constante a la etiqueta del nodo.

Precondición

`!null()`

6.6.3.9. `template<typename T, int K = 2> node& ktree< T, K >::node::operator= (const node & n)`

Operador de asignación.

Parámetros

<i>n</i>	el nodo a asignar
----------	-------------------

6.6.3.10. `template<typename T, int K = 2> bool ktree< T, K >::node::operator== (const node & n) const`

Operador de comparación de igualdad.

Parámetros

<i>n</i>	el nodo con el que se compara
----------	-------------------------------

6.6.3.11. `template<typename T, int K = 2> node ktree< T, K >::node::operator[] (int k) const`

Devuelve el hijo k-ésimo del nodo receptor.

Precondición

`!null()`

6.6.3.12. `template<typename T, int K = 2> node ktree< T, K >::node::parent () const`

Devuelve el padre del nodo receptor.

Precondición

`!null()`

6.6.4. Documentación de las funciones relacionadas y clases amigas

6.6.4.1. `template<typename T, int K = 2> friend class child_iterator [friend]`

Definición en la línea 391 del archivo ktree.h.

6.6.4.2. `template<typename T, int K = 2> friend class ktree< T, K > [friend]`

Definición en la línea 390 del archivo ktree.h.

La documentación para esta clase fue generada a partir del siguiente fichero:

- `include/ktree.h`

7. Documentación de archivos

7.1. Referencia del Archivo doc/doxys/genData.dox

7.2. Referencia del Archivo include/ktree.h

TDA `ktree<T,K>` Representa un árbol general con nodos etiquetados con datos del tipo T. Cada nodo puede tener entre 0 y K hijos como máximo, numerados de 0 a K-1.

```
#include <queue>
#include <string>
#include <iostream>
#include "../src/ktree.cpp"
#include "../src/ktreeNode.cpp"
```

Clases

- class `ktree< T, K >`
- class `ktree< T, K >::node`
- class `ktree< T, K >::node::child_iterator`
- class `ktree< T, K >::const_node`
- class `ktree< T, K >::const_node::child_iterator`

7.2.1. Descripción detallada

TDA `ktree<T,K>` Representa un árbol general con nodos etiquetados con datos del tipo T. Cada nodo puede tener entre 0 y K hijos como máximo, numerados de 0 a K-1.

Autor

Juan F. Huete

Bug Por esepificar

El `ktree<T,K>` exporta los siguientes tipos:

- `ktree<T,K>` árbol general
- `ktree<T,K>::node ->` Hace referencia a un nodo del árbol
- `ktree<T,K>::const_node ->` Hace referencia a un nodo constante (no se puede modificar su contenido) del árbol
- `ktree<T,k>::node::iterator ->` Iteradores sobre los descendientes directos de un nodo
- `ktree<T,K>::node::iterator ->` Iterador constante sobre los descendientes directos de un nodo
- `ktree<T,K>::size_type ->` Hace referencia al tamaño del árbol (unsigned int)

T debe tener definidas las operaciones:

- `T & operator=(const T & e);`
- `bool operator!=(const T & e);`
- `bool opertaor==(const T & e);`

Son mutables. Residen en memoria dinámica.

Un ejemplo de su uso:

```
#include <iostream>
#include <string>
#include <queue>
#include <set>
#include <utility>
#include <random>
#include <chrono>
#include <cmath>
#include "ktree.h"
#include "Nmer.h"

using namespace std;
using namespace std::chrono;

std::ostream& operator<<(ostream & s, const pair<char,int> & par){
    s << par.first << " " << par.second;
    return s;
}
```

```

template<typename T, int K>
vector<T> calcula_padres(typename ktree<T,K>::const_node n){
vector<T> salida;
while (!n.null()){
    salida.push_back(*n);
    n = n.parent();
}
return salida;
}

template<typename T, int K>
void recorrido_por_niveles(const ktree<T,K> & a){
recorrido_por_niveles<T,K>(a.root());
}

template<typename T, int K>
void recorrido_por_niveles(typename ktree<T,K>::const_node n){
typename ktree<T,K>::const_node aux;
queue<typename ktree<T,K>::const_node > cola;

if (!n.null())
    cola.push(n);

while (!cola.empty()){
    aux = cola.front();
    cola.pop();
    cout << "(" << (*aux) << ") ";
    for (auto hijo : aux)
        cola.push(hijo);
}
cout << endl;
}

template<typename T, int K>
int Altura(const ktree<T,K> & arbol){
return Altura<T,K>(arbol.root());
}

template<typename T, int K>
int Altura(typename ktree<T,K>::const_node n){
int hmax=-1;
if (!n.null()){
    for (auto hijo:n) {
        int hact = Altura<T,K>(hijo);
        if (hact > hmax)
            hmax = hact;
    }
    return hmax+1;
}
else return -1;
}

template<typename T, int K>
void recorrido_preorden(const ktree<T,K> & arbol){
recorrido_preorden<T,K>(arbol.root());
}

template<typename T, int K>
void recorrido_preorden(typename ktree<T,K>::const_node n){

if (!n.null()){
    cout << "(" << (*n) << ") ";
    typename ktree<T,K>::const_node::child_iterator ini = n.
        begin();
    typename ktree<T,K>::const_node::child_iterator fin = n.
        end();
    while (ini!=fin){
        recorrido_preorden<T,K>(*ini);
        ++ini;
    }
}
}

template<typename T, int K>
void recorrido_inorden(const ktree<T,K> & arbol){
recorrido_inorden<T,K>(arbol.root());
}

```

```

template<typename T, int = 2>
void recorrido_inorden(typename ktree<T,2>::const_node n){

if (!n.null()){
    recorrido_inorden<T,2>(n.k_child(0));
    cout << *n << "..";
    recorrido_inorden<T,2>(n.k_child(1));
}
}

template<typename T, int K>
void listar_hijos(typename ktree<T,K>::node n){

if (!n.null()){

    for (int i=0; i<K;i++){
        if (!n.k_child(i).null())
            cout << "uso kchild" << *(n.k_child(i)) << endl;
    }

    typename ktree<T,K>::node::child_iterator ini = n.
        begin();
    typename ktree<T,K>::node::child_iterator fin = n.
        end();
    while (ini!=fin){
        typename ktree<T,K>::node aux = *ini;
        cout << "uso iterador " << (*aux) << ".." << endl;
        ++ini;
    }

    for ( typename ktree<T,K>::node hijo : n)
        cout << "uso for en rango " << *hijo << endl;
    cout << "sss" << endl;
}
}

template <typename T>
void insertar_valorABB(ktree<T,2> & arbol, const T &valor){

    if (arbol.empty())
        arbol.assing(valor);
    else {
        typename ktree<T,2>::node n = arbol.root();
        bool insertado = false;
        int hijo;
        while (!insertado){
            if ( valor < *n) hijo = 0;
            else hijo = 1;

            if (n.k_child(hijo).null()){
                arbol.insert_k_child(n, hijo, valor);
                insertado = true;
            }
            else n=n.k_child(hijo);
        }
    }
}

template<typename T, int K>
void CambiaEtiquetas(typename ktree<T,K>::node n, const T & antigua, const
    T & nueva){
if (!n.null()){
    if (*n==antigua)
        *n=nueva;
    for (int i=0; i<K;i++){
        if (!n.k_child(i).null())
            CambiaEtiquetas<T,K>(n.k_child(i), antigua, nueva);
    }
}
}

/*
template<typename T, int K>
void CambiaEtiquetas2(typename ktree<T,K>::const_node n, const T & antigua, const T & nueva){
    cout << "Modificamos etiquetas, lo que puede ser error " << endl;
    if (!n.null()){

        cout << "(c" << (*n) << ") ";
    }
}

```

```

    if (*n==antigua)
        *n=nueva; <---- NO compila, es const_node y no se puede cambiar su valor
    for (int i=0; i<K;i++){
        if (!n.k_child(i).null())
            CambiaEtiquetas2<T,K>(n.k_child(i),antigua, nueva);
    }
}
}
*/

void TiempoComparativo( int tama){

    ktree<int,2> abb;
    multiset<int> cjto;
    vector<int> datos;
    datos.reserve(tama);
    cout<< "\n Analisis comparativo de tiempos para abb (con ktree) y multiset para "« tama « " datos"«endl;
    std::default_random_engine generator;
    std::uniform_int_distribution<int> distribution(1,5000);

    for (int i=0;i<tama;i++)
        datos.push_back(distribution(generator));

    high_resolution_clock::time_point tantes,tdespues;
    duration<double> tiempo_transcurrido;

    tantes = high_resolution_clock::now();
    for (int i = 0; i< tama; i++){
        insertar_valorABB<int>(abb,datos[i]);
    }
    tdespues = high_resolution_clock::now();
    tiempo_transcurrido = duration_cast<duration<double> >(tdespues - tantes);
    cout << "arbol (size "«abb.size() « ") t= " « tiempo_transcurrido.count() « endl;

    tantes = high_resolution_clock::now();
    for (int i = 0; i< tama; i++){
        cjto.insert(datos[i]);
    }
    tdespues = high_resolution_clock::now();
    tiempo_transcurrido = duration_cast<duration<double> >(tdespues - tantes);
    cout << "multiset size( "« cjto.size() « ") t= " « tiempo_transcurrido.count() « endl;

    cout << "Altura: abb " « Altura(abb) « " multiset (cota superior) " « 2.0*log2(tama+1.0)«endl;
}

int main(){

    ktree<int,2> abb;

    for (int i=3; i<10; i+=2)
        insertar_valorABB<int>(abb,i);

    for (int i=1; i<15; i+=3)
        insertar_valorABB<int>(abb,i);

    cout << "\n Listado hijos de la raiz" «endl;
    listar_hijos<int,2>(abb.root());

    cout << "\n Listado en Inorden" «endl;
    recorrido_inorden<int,2>(abb);

    cout << "\n Listado en Preorden" «endl;
    recorrido_preorden<int,2>(abb);

    cout << "\n Listado por Niveles" « endl;
    recorrido_por_niveles<int,2>(abb);
    int org = 5;
    int nv = 0;
    cout << "Modifico etiquetas "« org « "->" « nv« endl;
    cout << "(Si es arbol binario de busqueda puede violar el invariante)" «endl;
    CambiaEtiquetas<int,2>(abb.root(), org, nv);

    cout << "\n Listado en Inorden" «endl;
    recorrido_inorden<int,2>(abb);
}

```

```

TiempoComparativo(1000000);

ktree<pair<char,int>,4> X;
recorrido_preorden(X);

Nmer prueba;
prueba.loadSerialized("../datos/cadenaSimple.srl");
prueba.list_Nmer();
cout << prueba.size() << " " << prueba.length() << endl;

return 0;
}

```

Definición en el archivo [ktree.h](#).

7.3. Referencia del Archivo include/Nmer.h

TDA [Nmer](#) Representa un conjunto de [Nmer](#) subsecuencias de tamaño 1 hasta N que se pueden obtener a partir de una cadena de ADN.

```

#include <string>
#include <set>
#include <algorithm>
#include "ktree.h"

```

Clases

- class [Nmer](#)

7.3.1. Descripción detallada

TDA [Nmer](#) Representa un conjunto de [Nmer](#) subsecuencias de tamaño 1 hasta N que se pueden obtener a partir de una cadena de ADN.

Autor

alumno

Bug Por especificar

Definición en el archivo [Nmer.h](#).

7.4. Referencia del Archivo src/ejemploKtree.cpp

```

#include <iostream>
#include <string>
#include <queue>
#include <set>
#include <utility>
#include <random>
#include <chrono>
#include <cmath>
#include "ktree.h"
#include "Nmer.h"

```

Funciones

- `std::ostream & operator<< (ostream &s, const pair< char, int > &par)`
- `template<typename T, int K>
vector< T > calcula_padres (typename ktree< T, K >::const_node n)`
- `template<typename T, int K>
void recorrido_por_niveles (const ktree< T, K > &a)`
- `template<typename T, int K>
void recorrido_por_niveles (typename ktree< T, K >::const_node n)`
- `template<typename T, int K>
int Altura (const ktree< T, K > &arbol)`
- `template<typename T, int K>
int Altura (typename ktree< T, K >::const_node n)`
- `template<typename T, int K>
void recorrido_preorden (const ktree< T, K > &arbol)`
- `template<typename T, int K>
void recorrido_preorden (typename ktree< T, K >::const_node n)`
- `template<typename T, int K>
void recorrido_inorden (const ktree< T, K > &arbol)`
- `template<typename T, int = 2>
void recorrido_inorden (typename ktree< T, 2 >::const_node n)`
- `template<typename T, int K>
void listar_hijos (typename ktree< T, K >::node n)`
- `template<typename T >
void insertar_valorABB (ktree< T, 2 > &arbol, const T &valor)`
- `template<typename T, int K>
void CambiaEtiquetas (typename ktree< T, K >::node n, const T &antigua, const T &nueva)`
- `void TiempoComparativo (int tama)`
- `int main ()`

7.4.1. Documentación de las funciones

7.4.1.1. `template<typename T, int K> int Altura (const ktree< T, K > & arbol)`

Definición en la línea 59 del archivo ejemploKtree.cpp.

7.4.1.2. `template<typename T, int K> int Altura (typename ktree< T, K >::const_node n)`

Definición en la línea 65 del archivo ejemploKtree.cpp.

7.4.1.3. `template<typename T, int K> vector<T> calcula_padres (typename ktree< T, K >::const_node n)`

Definición en la línea 22 del archivo ejemploKtree.cpp.

7.4.1.4. `template<typename T, int K> void CambiaEtiquetas (typename ktree< T, K >::node n, const T & antigua, const T & nueva)`

Definición en la línea 172 del archivo ejemploKtree.cpp.

7.4.1.5. `template<typename T > void insertar_valorABB (ktree< T, 2 > & arbol, const T & valor)`

Definición en la línea 146 del archivo ejemploKtree.cpp.

7.4.1.6. `template<typename T, int K> void listar_hijos (typename ktree< T, K >::node n)`

Definición en la línea 119 del archivo ejemploKtree.cpp.

7.4.1.7. `int main ()`

Definición en la línea 244 del archivo `ejemploKtree.cpp`.

7.4.1.8. `std::ostream& operator<< (ostream & s, const pair< char, int > & par)`

Definición en la línea 16 del archivo `ejemploKtree.cpp`.

7.4.1.9. `template<typename T , int K> void recorrido_inorden (const ktree< T, K > & arbol)`

Definición en la línea 102 del archivo `ejemploKtree.cpp`.

7.4.1.10. `template<typename T , int = 2> void recorrido_inorden (typename ktree< T, 2 >::const_node n)`

Definición en la línea 108 del archivo `ejemploKtree.cpp`.

7.4.1.11. `template<typename T , int K> void recorrido_por_niveles (const ktree< T, K > & a)`

Definición en la línea 33 del archivo `ejemploKtree.cpp`.

7.4.1.12. `template<typename T , int K> void recorrido_por_niveles (typename ktree< T, K >::const_node n)`

Definición en la línea 40 del archivo `ejemploKtree.cpp`.

7.4.1.13. `template<typename T , int K> void recorrido_preorden (const ktree< T, K > & arbol)`

Definición en la línea 81 del archivo `ejemploKtree.cpp`.

7.4.1.14. `template<typename T , int K> void recorrido_preorden (typename ktree< T, K >::const_node n)`

Definición en la línea 87 del archivo `ejemploKtree.cpp`.

7.4.1.15. `void TiempoComparativo (int tama)`

Definición en la línea 203 del archivo `ejemploKtree.cpp`.

7.5. Referencia del Archivo `src/Nmer.cpp`

```
#include <fstream>
#include <string>
#include "ktree.h"
#include "Nmer.h"
```


Índice alfabético

- ~ktree
 - ktree, 21
- Altura
 - ejemploKtree.cpp, 37
- altura_nodo
 - Nmer, 26
- assing
 - ktree, 21
- begin
 - ktree::const_node, 18
 - ktree::node, 30
- calcula_padres
 - ejemploKtree.cpp, 37
- CambiaEtiquetas
 - ejemploKtree.cpp, 37
- cbegin
 - ktree::const_node, 18
- cend
 - ktree::const_node, 18
- child_iterator
 - ktree::const_node, 19
 - ktree::const_node::child_iterator, 16
 - ktree::node, 31
 - ktree::node::child_iterator, 15
- child_number
 - ktree::const_node, 18
 - ktree::node, 30
- clear
 - ktree, 21
- commonNmer
 - Nmer, 26
- comparison
 - Nmer, 26
- const_node
 - ktree::const_node, 17
 - ktree::const_node::child_iterator, 16
- containsString
 - Nmer, 27
- deserialize
 - ktree, 21
- doc/doxys/genData.dox, 31
- ejemploKtree.cpp
 - Altura, 37
 - calcula_padres, 37
 - CambiaEtiquetas, 37
 - insertar_valorABB, 37
 - listar_hijos, 37
 - main, 37
 - operator<<, 38
 - recorrido_inorden, 38
 - recorrido_por_niveles, 38
 - recorrido_preorden, 38
 - TiempoComparativo, 38
- empty
 - ktree, 23
- end
 - ktree::const_node, 18
 - ktree::node, 30
- include/Nmer.h, 36
- include/ktree.h, 31
- included
 - Nmer, 27
- insert_k_child
 - ktree, 23
- insertar_valorABB
 - ejemploKtree.cpp, 37
- k_child
 - ktree::const_node, 18
 - ktree::node, 30
- ktree
 - ~ktree, 21
 - assing, 21
 - clear, 21
 - deserialize, 21
 - empty, 23
 - insert_k_child, 23
 - ktree, 21
 - kvalue, 25
 - operator=, 23
 - operator==, 24
 - prune_k_child, 24
 - recorrido_preorden, 24
 - root, 24
 - serialize, 24
 - size, 24
 - size_type, 21
- ktree< T, K >, 19
 - ktree::node, 31
- ktree< T, K >::const_node, 16
- ktree< T, K >::const_node::child_iterator, 15
- ktree< T, K >::node, 28
- ktree< T, K >::node::child_iterator, 14
- ktree::const_node
 - begin, 18
 - cbegin, 18
 - cend, 18
 - child_iterator, 19
 - child_number, 18
 - const_node, 17
 - end, 18
 - k_child, 18
 - null, 18
 - operator*, 18
 - operator=, 18

- operator==, 19
- parent, 19
- ktree::const_node::child_iterator
 - child_iterator, 16
 - const_node, 16
 - operator*, 16
 - operator++, 16
 - operator=, 16
 - operator==, 16
- ktree::node
 - begin, 30
 - child_iterator, 31
 - child_number, 30
 - end, 30
 - k_child, 30
 - ktree< T, K >, 31
 - node, 29
 - null, 30
 - operator*, 30
 - operator=, 30
 - operator==, 31
 - parent, 31
- ktree::node::child_iterator
 - child_iterator, 15
 - node, 15
 - operator*, 15
 - operator++, 15
 - operator=, 15
 - operator==, 15
- kvalue
 - ktree, 25
- length
 - Nmer, 27
- level
 - Nmer, 27
- list_Nmer
 - Nmer, 27
- listar_hijos
 - ejemploKtree.cpp, 37
- loadSerialized
 - Nmer, 27
- main
 - ejemploKtree.cpp, 37
- Nmer, 25
 - altura_nodo, 26
 - commonNmer, 26
 - comparison, 26
 - containsString, 27
 - included, 27
 - length, 27
 - level, 27
 - list_Nmer, 27
 - loadSerialized, 27
 - Nmer, 26
 - Prefix, 27
 - rareNmer, 28
- recorrer_niveles, 28
- root, 28
- sequenceADN, 28
- size, 28
- size_type, 26
- node
 - ktree::node, 29
 - ktree::node::child_iterator, 15
- null
 - ktree::const_node, 18
 - ktree::node, 30
- operator<<
 - ejemploKtree.cpp, 38
- operator*
 - ktree::const_node, 18
 - ktree::const_node::child_iterator, 16
 - ktree::node, 30
 - ktree::node::child_iterator, 15
- operator++
 - ktree::const_node::child_iterator, 16
 - ktree::node::child_iterator, 15
- operator=
 - ktree, 23
 - ktree::const_node, 18
 - ktree::const_node::child_iterator, 16
 - ktree::node, 30
 - ktree::node::child_iterator, 15
- operator==
 - ktree, 24
 - ktree::const_node, 19
 - ktree::const_node::child_iterator, 16
 - ktree::node, 31
 - ktree::node::child_iterator, 15
- parent
 - ktree::const_node, 19
 - ktree::node, 31
- Prefix
 - Nmer, 27
- prune_k_child
 - ktree, 24
- rareNmer
 - Nmer, 28
- recorrer_niveles
 - Nmer, 28
- recorrido_inorden
 - ejemploKtree.cpp, 38
- recorrido_por_niveles
 - ejemploKtree.cpp, 38
- recorrido_preorden
 - ejemploKtree.cpp, 38
 - ktree, 24
- root
 - ktree, 24
 - Nmer, 28
- sequenceADN

- Nmer, [28](#)
- serialize
 - ktree, [24](#)
- size
 - ktree, [24](#)
 - Nmer, [28](#)
- size_type
 - ktree, [21](#)
 - Nmer, [26](#)
- src/Nmer.cpp, [38](#)
- src/ejemploKtree.cpp, [36](#)
- TiempoComparativo
 - ejemploKtree.cpp, [38](#)