TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Librería de Estructuras de Datos Compactas en Rust

Estudante: Jorge Hermo González

Dirección: Susana Ladra González

Diego Seco Naveiras

A Coruña, septiembre de 2023.

Viaje antes que destino

Agradecimientos

Estas son las últimas palabras que he escrito en esta memoria, pues agradecer a todas y cada una de las personas que me han acompañado en este viaje es lo que me ha resultado más complicado.

Primero de todo, le dedico este trabajo principalmente a mi madre, a quien las palabras no me llegan para agradecer su amor, sacrificio y apoyo incondicional desde que tengo memoria, que es a quien le debo la persona en la que me he convertido. También fueron pilares fundamentales mis abuelos y mi hermano, quienes siempre han estado disponibles y apoyándome, a pesar de no poder haber estado tanto tiempo con ellos como me hubiese gustado durante estos largos cuatro años. Gracias por todo.

A los tutores de este proyecto, Susana y Diego, quienes siempre han estado presentes para ayudarme y guiarme en todo momento, además de brindarme un sinfín de oportunidades que me han permitido crecer personal y profesionalmente. Les admiro profundamente y se han convertido en mis referentes académicos, no puedo agradecérselo lo suficiente. Aprecio enormemente la confianza que tuvieron en mí desde el primer momento que les presenté la idea de este proyecto. Haberlos conocido a ellos, y a todo el Laboratorio de Bases de Datos, ha sido uno de los mejores regalos que pude recibir en esta etapa de mi vida. Sé que nuestros caminos se volverán a cruzar, ya sea en lo personal o en lo profesional, así que esto es un hasta pronto.

A mis amigos Julián, Sergio y Diego, con los que he tenido la suerte de disfrutar todos estos años de la carrera y, espero, que sean muchos años más. Han sido un soporte fundamental, y no puedo estar más agradecido y orgulloso de ellos.

A Jorge, el amigo que conservo desde hace más tiempo y con el que he compartido una gran parte de mi vida. Por tu amistad incondicional y la capacidad que demuestras a diario para soportarme, espero que sigamos compartiendo muchos más años juntos. Ser tu amigo es de las mejores cosas que me han pasado, muchas gracias por todo.

Y a David, con quien más tiempo he compartido durante todos estos años, de lejos. Presente tanto en las buenas como en las malas. Has sido de lejos la persona con la que más he aprendido, académica y profesionalmente, siempre haciendo conseguir que me supere cada vez más. Sin ti, estoy seguro de que no me hubiese salido todo tan bien. Has sido mi compañero de clase, de piso, de fiesta, de viaje, de trabajo… Y aún así no me canso de pasar tiempo contigo. Estoy muy orgulloso de ti y disfruto tus éxitos como si fuesen míos. Me faltan las palabras para poder explicarte todo el aprecio que te tengo y lo mucho que significas para mí, así que únicamente te diré gracias.

Y por último, quiero dar las gracias a todas las demás personas que me han acompañado durante estos años y que me han servido para lograr crecer como persona. Lo más importante de esta etapa no ha sido el fin, sino que como dicen las Palabras, *“Viaje antes que destino”*.

Resumen

El crecimiento exponencial de los datos en la actualidad plantea desafíos significativos en términos de almacenamiento y procesamiento eficiente de los mismos. Este trabajo fin de grado se centra en la importancia de las estructuras de datos compactas como una solución clave en el tratamiento de datos a gran escala. A diferencia de las técnicas clásicas de compresión, estas estructuras permiten operar con datos sin necesidad de descomprimirlos por completo, lo que ahorra tiempo y espacio en memoria. Este enfoque se ha vuelto esencial en campos como la recuperación de información y la bioinformática debido al crecimiento masivo de datos.

El lenguaje de programación Rust, conocido por su seguridad, gestión automática de memoria y eficiencia, se ha convertido en una de las opciones preferidas en la actualidad en términos de innovación y modernización en la industria de la tecnología.

Ante la falta de una librería de estructuras de datos compactas en Rust que sea competitiva con el estado del arte en otros lenguajes de programación, este proyecto aprovechará las ventajas que nos proporciona este lenguaje para desarrollar una librería de estructuras de datos compactas de código abierto, proporcionando así a la comunidad científica y a los desarrolladores de Rust una herramienta flexible, potente y fácil de usar para sus proyectos.

Además, con este trabajo fin de grado se busca fomentar la reproducibilidad, la reutilización y el avance en la investigación en el campo de investigación en estructuras de datos compactas. De esta manera, se contribuirá a la expansión y adopción de Rust en la investigación y al desarrollo de software científico eficiente y confiable.

Abstract

The exponential growth of data nowadays poses significant challenges in terms of efficient storage and processing. This undergraduate thesis focuses on the importance of compact data structures as a key solution in handling large scale data. Unlike classical compression techniques, these structures allow for operations on data without the need for complete decompression, saving time and memory space. This approach has become essential in fields such as information retrieval and bioinformatics due to the massive growth of data.

The Rust programming language, known for its safety, automatic memory management and efficiency, has become one of the preferred options at present for innovation and modernization in the technology industry.

In the absence of a competitive library of compact data structures in Rust compared to the state of the art in other programming languages, this project will leverage the advantages provided by this language to develop an open source compact data structures library. This

will provide the scientific community and Rust developers a flexible, powerful, and easy to use tool for their projects.

Furthermore, this undergraduate thesis aims to promote reproducibility, reuse, and progress in research on the field of compact data structures. In this way, it will contribute to the expansion and adoption of Rust in research and the development of efficient and reliable scientific software.

Palabras clave:

- Estructuras de Datos Compactas
- Vectores de bits
- *Wavelet Trees*
- Rust
- Código Libre
- Reproducibilidad en investigación

Keywords:

- Compact Data Structures
- Bit vectors
- *Wavelet Trees*
- Rust
- Open Source
- Reproducibility in research

Índice general

1 Introducción 1
1.1 Motivaciones . . . 1
1.2 Objetivos . . . 3
1.3 Estructura de la memoria . . . 3

2 Fundamentos teóricos 5
2.1 Teoría de la información . . . 5
2.1.1 Entropía . . . 5
2.2 Estructuras de datos compactas . . . 7
2.3 Array compacto de enteros . . . 8
2.3.1 Elementos de tamaño fijo . . . 9
2.3.2 Elementos de tamaño variable . . . 11
2.4 Vector de bits . . . 12
2.4.1 Rank . . . 13
2.4.2 Select . . . 17
2.5 Vector de bits comprimido con H_0 . . . 19
2.5.1 Access . . . 21
2.5.2 Rank y Select . . . 21
2.6 Wavelet Tree . . . 22
2.6.1 Access . . . 24
2.6.2 Rank . . . 25
2.6.3 Select . . . 25
2.6.4 Wavelet Tree comprimido con H_0 . . . 25

3 Fundamentos tecnológicos 26
3.1 Estado del arte . . . 26
3.1.1 Análisis de soluciones existentes . . . 26
3.1.2 Conclusión de las soluciones analizadas . . . 29

3.2 Lenguaje de programación Rust 30

4 Metodología y Planificación **31**
4.1 Metodología de desarrollo 31
4.1.1 Herramientas utilizadas 32
4.2 Planificación 33
4.2.1 Recursos 34
4.2.2 Costes 35
4.3 Resultado del seguimiento 36

5 Análisis **38**
5.1 Selección de las estructuras de datos 38
5.2 Análisis de requisitos 39
5.2.1 Requisitos funcionales 39
5.2.2 Requisitos no funcionales 40

6 Diseño **41**
6.1 Vector de bits 41
6.2 Vector de enteros compacto 43
6.2.1 Vector de enteros compacto de tamaño fijo 44
6.2.2 Vector de enteros compacto de tamaño variable 44
6.3 Estructuras auxiliares 45
6.3.1 Rank con un nivel de directorio 48
6.3.2 Rank con dos niveles de directorio 49
6.4 Vector de bits comprimido con H_0 49
6.5 Wavelet Tree 52

7 Implementación y pruebas **55**
7.1 Implementación 55
7.1.1 Optimizaciones de compilación 56
7.1.2 Utilización del API *unsafe* de Rust 57
7.1.3 Optimización del vector de bits comprimido con H_0 59
7.1.4 Serialización y deserialización 62
7.1.5 Documentación 64
7.2 Pruebas 65
7.3 Licencia y distribución 68

8 Experimentos y comparativa **69**

8.1 Marco experimental 69

8.2 Evaluación experimental 70

8.2.1 Vectores de bits 70

8.2.2 Wavelet Trees 75

8.3 Comparativa cualitativa 78

9 Conclusiones y trabajo futuro **79**

9.1 Conclusiones 79

9.2 Trabajo futuro 80

A Material adicional **82**

A.1 Instalación y uso 82

A.2 Integración Continua 83

A.3 Algoritmo de cálculo de los coeficientes binomiales 83

A.4 Operación de select en la estructura *DenseSamplingRank* 84

A.5 Pruebas unitarias 87

Lista de acrónimos **88**

Glosario **89**

Bibliografía **90**

Índice de figuras

2.1 Ejemplo de representación de un array compacto de enteros 10
2.2 Ejemplo de representación de una estructura de $rank$ con un nivel de directorio 14
2.3 Ejemplo de representación de una estructura de $rank$ con dos niveles de directorio 16
2.4 Ejemplo de representación de un *wavelet tree* 23

6.1 Diagrama de clases del vector de bits plano 42
6.2 Diagrama de clases del vector compacto de enteros de tamaño fijo 44
6.3 Diagrama de clases del vector compacto de enteros de tamaño variable 45
6.4 Diagrama de clases de los elementos genéricos para las estructuras auxiliares . 46
6.5 Diagrama de clases de la estructura de $rank$ con un nivel de directorio 48
6.6 Diagrama de clases de la estructura de $rank$ con dos niveles de directorio . . . 50
6.7 Diagrama de clases del vector de bits comprimido con H_0 51
6.8 Diagrama de clases del *Wavelet Tree* 53

7.1 Página web de la documentación de la librería 65

8.1 Resultados experimentales para la operación de $rank$ sobre estructuras auxiliares 72
8.2 Resultados experimentales para la operación de $select$ sobre estructuras auxiliares 72
8.3 Resultados experimentales para la operación de $rank$ sobre vectores de bits comprimidos con H_0 74
8.4 Resultados experimentales para la operación de $select$ sobre vectores de bits comprimidos con H_0 75
8.5 Resultados experimentales de la estructura *wavelet tree* para el corpus de texto *proteins* 76

8.6 Resultados experimentales de la estructura *wavelet tree* para el corpus de texto *english* 77

Índice de tablas

4.1 Planificación estimada de las tareas del proyecto 34
4.2 Costes del proyecto 36
4.3 Fechas reales de las tareas del proyecto 36
4.4 Costes reales del proyecto 37

8.1 Propiedades de los corpus de texto utilizados en los experimentos 70

A.1 Número de pruebas unitarias de cada una de las estructura de datos implementadas 87

Capítulo 1

Introducción

En este capítulo se discute la motivación principal para este proyecto y los objetivos que debe cumplir para tener éxito.

1.1 Motivaciones

En la actualidad, el crecimiento de los datos que se generan en el mundo es exponencial, y cada vez es más difícil almacenarlos y procesarlos de forma eficiente. Por ello, es necesario utilizar técnicas que permitan almacenar dichos datos en el mínimo espacio posible, para posteriormente poder realizar operaciones sobre ellos de forma eficiente.

La estructuras de datos compactas [1] presentan una gran utilidad en el tratamiento de datos a gran escala, puesto que nos permiten almacenar grandes cantidades de datos en un espacio reducido y realizar operaciones sobre ellos de forma eficiente. La ventaja que presentan frente a técnicas clásicas de compresión es que permiten realizar dichas operaciones sin tener que descomprimir los datos totalmente en un paso previo, lo que supone un ahorro de tiempo y de espacio en memoria. Este campo de investigación está motivado por la necesidad de tratar con enormes cantidades en memoria principal, intentando aprovecharse de la gran diferencia de velocidad que existe entre los distintos niveles de la jerarquía de memoria (registros del procesador, cachés L1, L2, L3, *RAM*, disco duro, etc.).

Este tipo de estructuras de datos son muy utilizadas en el campo de la recuperación de la información (motores de búsqueda Web, bases de datos, recuperación de documentos, etc.) debido al enorme crecimiento de datos que se ha producido en los últimos años (alineado con el auge del *Big Data*), mostrando resultados muy satisfactorios [2, 3]. Algunas de las estructuras de datos compactas que se utilizan en este campo pueden ser los k^2*-trees* [4] o los *Inverted Treaps* [5].

También son muy utilizadas en el campo de la bioinformática [6, 7], donde se trabaja con grandes cantidades de datos genómicos. En este campo, se utilizan estructuras como puede

ser el FM-index [8], el cual nos permite almacenar un texto en un espacio cercano al de su forma comprimida y al mismo tiempo poder localizar cualquier subcadena de dicho texto en tiempo sublineal, o estructuras como los *Compressed Suffix Arrays* [9] o los *Wavelet Trees* [10].

Por otro lado, el lenguaje de programación Rust [11] es un lenguaje moderno de bajo nivel, el cual proporciona grandes ventajas frente a otras alternativas similares como C o C++, como puede ser la gestión automática de memoria, la seguridad de memoria, gestor de paquetes y de proyecto integrado y muy buena documentación, a la vez que no presenta un rendimiento menor que sus alternativas [12, 13]. Este lenguaje está ganando mucha popularidad en los últimos años, siendo el lenguaje votado como el más querido por los desarrolladores en el año 2023, en la encuesta anual de StackOverflow [14]. Son cada vez más los proyectos grandes que se desarrollan en este lenguaje [15, 16, 17, 18, 19, 20], y también cada vez más empresas grandes, como Microsoft [21], Amazon [22] o Google [23] están apostando por Rust. Este lenguaje tiene como objetivo permitir a todas las personas crear software fiable y eficiente, según dice su lema: *"Rust is a language empowering everyone to build reliable and efficient software"*.

Actualmente, son muchos los artículos científicos en el campo de la informática (y por lo tanto, también en el área de las estructuras de datos compactas) que o bien no proporcionan el código fuente de la implementación de sus aportaciones, o si lo hacen, proporcionan una implementación standalone que no puede ser utilizada en otros proyectos de forma sencilla, dificultando en gran medida la reproducibilidad de los resultados obtenidos [24, 25], la reutilización, el avance en investigación y las buenas prácticas en el desarrollo de software (puesto que muchas veces simplemente se copian los archivos fuente y no se utiliza una paquete de software probado y gestionado por herramientas de gestión de paquetes), problemas que provocan una crisis de reproducibilidad en la ciencia [26, 27, 28]. A pesar de esto, sí que existen algunas iniciativas que intentan solucionar estos problemas en el campo de las estructuras de datos compactas, como puede ser la Succinct Data Structures Library (SDSL) [29], una librería de estructuras de datos compactas escrita en C++ muy conocida en dicha área, que proporciona implementaciones de varias estructuras de datos compactas descritas en la literatura.

Sin embargo, no existe una alternativa en el lenguaje Rust que esté tan establecida como la anteriormente mencionada, por lo que el objetivo de este proyecto es desarrollar una librería de estructuras de datos compactas en Rust de código libre, que permita a la comunidad científica y a la comunidad de Rust poder utilizar dichas estructuras de datos en sus proyectos de forma sencilla. De esta forma, la comunidad científica podrá beneficiarse de las ventajas que proporciona el lenguaje Rust en sus investigaciones [30, 31]. Además, también se pretende que esta librería sirva como un marco sobre el que futuras aportaciones en el campo de las estructuras de datos compactas puedan ser implementadas, de forma que su distribución,

reutilización y reproducibilidad sea mucho más factible, utilizando un lenguaje que favorezca la fiabilidad, seguridad y eficiencia del software.

1.2 Objetivos

El objetivo principal de este proyecto es el desarrollo de una librería de estructuras de datos compactas en el lenguaje de programación Rust cuya facilidad de uso y eficiencia sea comparable al estado del arte de las librerías de este campo. Para ello se han definido los siguientes objetivos concretos:

- Estudio, aprendizaje y análisis de la literatura sobre estructuras de datos compactas, seleccionando las destacadas como las más fundamentales.
- Aprendizaje del lenguaje de programación Rust y de sus conceptos avanzados, de forma que se pueda realizar un desarrollo de calidad en este lenguaje.
- Análisis y comprensión de las distintas librerías existentes en el campo de las estructuras de datos compactas, tanto en Rust como en otros lenguajes de programación.
- Diseño, implementación y pruebas de las estructuras de datos seleccionadas, agrupadas bajo una misma librería.
- Experimentación y comparación de las implementaciones propuestas en este proyecto con otras soluciones existentes, tanto en rendimiento en tiempo y espacio ocupado de las estructuras de datos como en funcionalidades disponibles.

1.3 Estructura de la memoria

La memoria ha sido estructurada en los siguientes capítulos:

1. Introducción: En este capítulo se proporciona una breve aproximación al proyecto a realizar, así como se discuten las motivaciones y los objetivos del mismo.
2. Fundamentos teóricos: Aquí se introducirán los conceptos teóricos necesarios para entender el resto de la memoria y, por lo tanto, del proyecto realizado. Se hablará principalmente de la teoría de la información y de estructuras de datos compactas, explicando las estructuras concretas que serán diseñadas e implementadas.
3. Fundamentos tecnológicos: En este capítulo se explicarán los fundamentos tecnológicos utilizados en el proyecto, centrándose principalmente en el lenguaje de programación

Rust. Además, también hablará del estado del arte en lo referente a las diferentes librerías existentes sobre estructuras de datos compactas, tanto en el lenguaje Rust como en otras alternativas como puede ser en el lenguaje C++.

4. Metodología y Planificación: Se tratará la metodología de desarrollo a seguir, la planificación inicial del proyecto y el seguimiento del mismo.

5. Análisis: En este capítulo se hablará de la viabilidad del lenguaje de programación Rust para este proyecto, así como de los requisitos funcionales y no funcionales que debe cumplir la librería implementada.

6. Diseño: Se explicará el diseño de la librería, así como de cada una de las estructuras de datos implementadas.

7. Implementación y pruebas: En este capítulo se explicarán los detalles de implementación de la librería y de las estructuras de datos implementadas, así como se detallará la aproximación utilizada para realizar las pruebas de las mismas.

8. Experimentos y comparativa: Se hablará de los resultados obtenidos en los experimentos realizados sobre las estructuras de datos seleccionadas, así como se realizará una comparación en rendimiento de espacio y tiempo contra implementaciones existentes. También se realizará una comparación cualitativa de la librería implementada con alguna de las alternativas existentes.

9. Conclusiones y trabajo futuro: En este capítulo se expondrán las conclusiones obtenidas tras la realización del proyecto, así como se propondrán posibles líneas de trabajo futuro.

Capítulo 2

Fundamentos teóricos

Debido al crecimiento explosivo de grandes colecciones de datos en los últimos años, es necesario utilizar técnicas como la compresión y el indexado para poder tratar con estos datos de una manera eficiente.

En este capítulo se introducirán conceptos básicos de teoría de la información y compresión de datos, a la vez que se detallarán distintas estructuras de datos compactas consideradas como las más fundamentales. Todos estos conceptos han sido estudiados en profundidad para poder desarrollar el trabajo de esta memoria y el diseño e implementación de las estructuras consideradas se basan principalmente en las descripción teórica de las mismas.

2.1 Teoría de la información

La teoría de la información trata sobre la medida y transmisión de la información a través de canales de comunicación. El trabajo de Shannon [32] sentó las bases sobre este campo, el cual tiene como objetivo encontrar los límites teóricos de la compresión, almacenamiento y transmisión de los datos, entre otros. Además, proporciona conceptos útiles como puede ser medir la cantidad de información en términos de bits.

Dada una variable aleatoria X, con una función de distribución de probabilidad p_X, la cantidad de información o sorpresa asociada con un suceso $x \in X$ se define por la cantidad $I_X(x) = \log_2 \frac{1}{p_X(x)}$, medida en bits. Si un suceso tiene una probabilidad de ocurrencia muy baja, entonces la cantidad de información (sorpresa) asociada a dicho suceso es alta. Por el contrario, si un suceso tiene probabilidad total de ocurrencia, entonces la sorpresa es nula, puesto que no se puede obtener ningún tipo de información sobre dicha observación.

2.1.1 Entropía

La entropía de una variable aleatoria X mide la cantidad de información media asociada con dicha variable aleatoria. Se define como $H(X) = E[I_X(x)] = \sum_{x \in X} p_X(x) I_X(x) =$

$\sum_{x \in X} p_X(x) \log_2 \frac{1}{p_X(x)}$, medida en bits.

A efectos prácticos, la entropía también se puede interpretar como el mínimo número medio de bits que se necesitan para identificar de forma única un determinado objeto dentro de un conjunto de objetos. La entropía indica el número óptimo de bits con el que poder representar dicho objeto y sirve como un límite inferior al espacio utilizado por una representación comprimida del objeto, es decir, no se puede utilizar un número menor de bits que el indicado por la entropía de forma que se puedan seguir representando todos los elementos del conjunto unívocamente.

Entropía del peor caso

Llamaremos a los identificadores de los objetos que queremos representar, códigos. Dichos códigos estarán formados por una secuencia de bits y tendrán un cierto tamaño. Entonces, la entropía nos indica cuál es la longitud media óptima de los códigos.

En el caso de que todos los objetos tengan la misma probabilidad de ocurrencia, la manera óptima de asociar un código a cada uno de los elementos de un conjunto U, es que todos los códigos tengan el mismo tamaño. Esto viene determinado por la llamada entropía del peor caso, denotada por $H_{wc}(U)$, y se puede ver que su valor es $H_{wc}(U) = \sum_{u \in U} \frac{1}{|U|} \log_2 |U| = \log_2 |U|$, la cual nos indica cual es la longitud en bits óptima que debe tener cada uno de los códigos anteriormente mencionados.

Si utilizamos códigos con longitud $l < H_{wc}(U)$, entonces únicamente tendríamos $2^l < 2^{H_{wc}(U)} = 2^{\log_2 |U|} = |U|$ códigos distintos, por lo que no podríamos representar todos los elementos de U de forma única. Por lo tanto, y debido a que la longitud de los códigos debe ser un número entero (y la entropía no tiene por qué serlo), la longitud de cada uno de los códigos debe ser de al menos $l = \lceil H_{wc}(U) \rceil$ bits. En caso de que no todos los códigos tengan el mismo tamaño, el más largo tiene que utilizar al menos $l = \lceil H_{wc}(U) \rceil$ bits.

Por ejemplo, la entropía del peor caso de todas las cadenas de texto de longitud n sobre un alfabeto Σ de tamaño σ es $\log_2 \sigma^n = n \log_2 \sigma$ bits. Por lo tanto, se necesitan como mínimo $n \log_2 \sigma$ bits para poder representar cualquier posible secuencia de n símbolos, si estos símbolos se consideran equiprobables.

Secuencias de bits

Si consideramos el caso en el que nuestro conjunto de objetos es $U = \{0,1\}$ (es decir, nuestros elementos son bits) y asumimos que un bit 1 tiene una probabilidad de ocurrencia p y un bit 0 tiene una probabilidad de ocurrencia $1-p$, entonces la entropía de dicha distribución (también llamada entropía binaria) es $H(p) = p \log_2 \frac{1}{p} + (1-p) \log_2 \frac{1}{1-p}$.

Este concepto también se aplica cuando los elementos de nuestro conjunto son secuencias de bits que se emiten por una fuente dada. Asumimos que dicha fuente emite cada bit de forma

independiente del resto (es decir, la probabilidad de ocurrencia de un bit no depende de los bits emitidos anteriormente), a lo que se llama una fuente de orden cero. Dado un conjunto $U = \{0,1\}^n$ de secuencias de bits de longitud n y la misma función de probabilidad p del caso anterior, la entropía de orden cero de dicho conjunto es $H_0(U) = nH(p)$, es decir, la entropía de una serie de eventos independientes es la suma de sus entropías individuales.

Si tenemos una secuencia de bits concreta, $B[1,n]$, que queremos comprimir de alguna forma, una aproximación inicial sería calcular cuál es su entropía. En este caso, no tenemos una fuente ideal de bits que emite 1s y 0s con una probabilidad conocida, pero sí que podemos estimar dicha probabilidad a partir de la secuencia de bits de forma empírica. Primero, asumimos que B ha sido generado a partir de una fuente de orden cero que emite cada bit de forma independiente del resto. Sea m el número de 1s en B, entonces una estimación empírica de la probabilidad p de que un bit 1 sea emitido por la fuente es $p \approx \frac{m}{n}$. Esto nos permite definir el concepto de entropía empírica de orden cero, $H_0(B) = H(\frac{m}{n}) = \frac{m}{n} \log_2 \frac{n}{m} + (\frac{n-m}{n}) \log_2 \frac{n}{n-m}$.

El concepto de la entropía empírica es útil puesto que nos permite estimar la entropía de Shannon de una secuencia de bits a partir de las frecuencias de ocurrencia de cada elemento observadas en dicha secuencia.

2.2 Estructuras de datos compactas

La compresión está muy relacionada con el almacenamiento de grandes volúmenes de datos, la cual es una técnica que se aprovecha de la redundancia que existe en dichos datos para poder representarlos utilizando menos espacio [33]. La compresión sienta sus bases en la teoría de la información, y, como se mencionó en la sección 2.1, esta estudia el mínimo espacio necesario para representar los datos. Muchos métodos de compresión están basados en algoritmos como puede ser el algoritmo de Huffman [34] (que permite obtener códigos de longitud mínima), el algoritmo de Lempel-Ziv [35, 36] (el cual permite asignar códigos de longitud fija a frases de longitud variable) o métodos aritméticos [37].

Sin embargo, la mayoría de los algoritmos de compresión requieren descomprimir todos los datos completamente antes de poder acceder a posiciones aleatorias de la secuencia comprimida o realizar cualquier consulta sobre ellos. Por lo tanto, la compresión generalmente es útil como un método de archivado de los datos eficiente en términos de espacio, pero estos deben ser descomprimidos para que puedan ser utilizamos otra vez. Esto puede ser un problema cuando se trabaja con grandes volúmenes de datos, puesto que la descompresión previa puede ser un proceso costoso en términos de tiempo, y además puede no ser factible tener todos los datos descomprimidos en memoria principal.

Las estructuras de datos compactas [1, 38] consisten en representar los datos de una manera que no solo ocupen menos espacio, sino que también permitan acceder y realizar consultas

sobre los ellos en su forma compacta, sin necesidad de descomprimirlos en un paso anterior. Esto permite almacenar, realizar consultas y manipular en memoria principal conjuntos de datos mucho más grandes de lo que sería posible si utilizásemos la representación original de los datos con estructuras de datos tradicionales.

Estas estructuras buscan aprovecharse de la gran diferencia en velocidad que existe entre los distintos niveles de la jerarquía de memoria (registros del procesador, cachés L1, L2, L3, RAM, disco duro, etc.), de manera que se intenta almacenar una mayor cantidad de información (en su forma compacta) en los niveles más rápidos de la jerarquía de lo que permitiría una representación plana de los datos [39], logrando así un mayor rendimiento en muchas ocasiones. La representación compacta generalmente necesita emplear más ciclos de CPU para poder realizar las operaciones de consulta, pero debido a la diferencia de velocidad de los distintos niveles de la jerarquía de memoria, el tiempo total de las operaciones de consulta sobre los datos puede ser mucho menor (por ejemplo, la diferencia en tiempo de recuperar un dato de una caché L1 y de hacerlo en *RAM* es de varios órdenes de magnitud, y si lo comparamos con recuperar el dato de disco, la diferencia es todavía más notable).

La mejora de velocidad puede ocurrir en cualquier nivel de la jerarquía de memoria, ya sea porque la representación compacta cabe totalmente en caché mientras que la tradicional no, o porque la representación compacta cabe en *RAM* mientras que la tradicional necesita ser almacenada en disco (en este último caso, la mejora de rendimiento que nos proporciona la representación compacta es muy significativa). Además, en casos en dispositivos con memoria limitada (por ejemplo, dispositivos móviles, microcontroladores, etc.) la representación compacta puede ser la única opción que permita poder trabajar con grandes volúmenes de datos. En otras ocasiones, la representación comprimida permite una computación más eficiente directamente, aún trabajando en el mismo nivel de la jerarquía de memoria.

2.3 Array compacto de enteros

Definiremos un array (o vector) de enteros $A[1,n]$ como una secuencia de números enteros de la cual se puede leer y escribir en posiciones arbitrarias. Los elementos de dicho array, por razones prácticas, los consideraremos números enteros sin signo (es decir, dado un tamaño en bits l, guardaremos números enteros A_i que cumplan $0 \leq A_i < 2^l$), pero la idea es fácilmente generalizable a números enteros con signo, puesto que cualquier intervalo que contenga números negativos se puede transformar a un intervalo de números positivos sumando el valor absoluto del número más pequeño del intervalo a todos los elementos del mismo.

De aquí en adelante consideraremos el modelo de computación *RAM*, en el cual se asume que el acceso a cualquier posición de la memoria principal se puede realizar en un tiempo constante. Además, llamaremos w al tamaño en bits de una palabra de memoria, asumiremos

que el tipo de número entero proporcionado por el lenguaje ocupa w bits y que las operaciones aritméticas básicas sobre este tipo de dato se realizan en tiempo constante.

En esta sección no estaremos interesados en la implementación clásica de arrays de enteros, la cual consiste en almacenar cada elemento utilizando w bits, sino que se estudiará una representación compacta del mismo, aprovechando la información que nos proporcionan los datos (por ejemplo, si el valor máximo que toman los datos es 511, entonces son necesarios únicamente 9 bits para representar dichos datos).

2.3.1 Elementos de tamaño fijo

Si asumimos que cada elemento de A se podría almacenar en l bits (esto es $0 \leq A[i] < 2^l$), una solución básica en la que utilicemos un array de n enteros de tamaño w bits podría gastar mucho más tamaño del necesario, $n \cdot w$ bits en lugar de los $n \cdot l$ bits que serían suficientes. Además, en muchas ocasiones se cumple que l es mucho más pequeño que w, y si el número de elementos es grande, se estaría desperdiciando una gran cantidad de memoria.

Representación

Para lograr la representación compacta del array A, utilizaremos un array de enteros $W[1, \lceil \frac{n \cdot l}{w} \rceil]$ de tamaño $\lceil \frac{n \cdot l}{w} \rceil$, el cual tiene un número total de bits suficiente para poder almacenar n elementos de tamaño l bits. Por simplicidad, veremos el array W como si fuese un array de bits $B[1, n \cdot l]$. Entonces, almacenaremos la representación en binario de cada uno de los elementos $A[i]$ en el intervalo $B[(i-1) \cdot l + 1, i \cdot l]$, almacenando el bit más significativo del elemento el primero de dicho intervalo. De esta forma, el espacio en bits utilizado por esta representación compacta es de exactamente $\lceil \frac{n \cdot l}{w} \rceil \cdot w$ bits, lo cual es mucho menor que $n \cdot w$ bits si se cumple que l es mucho menor que w. Se muestra un ejemplo de esta representación en la figura 2.1.

Lectura y escritura

Con esta representación, podemos realizar la operación de lectura de un elemento $A[i]$ en tiempo constante. Los bits que queremos leer se corresponden con $B[j,j']$, donde $j = (i-1) \cdot l + 1$ y $j' = i \cdot l$. Para que la lectura sea en tiempo constante, tenemos que establecer la restricción de que $j' - j + 1 \leq w$, de forma que los bits que se quieran leer estén completamente contenidos dentro de un único elemento del array W, o como mucho se encuentren repartidos en exactamente dos posiciones del array W. Esta restricción implica que como máximo podremos leer enteros de tamaño w bits, estableciendo que siempre se tiene que cumplir que $l \leq w$.

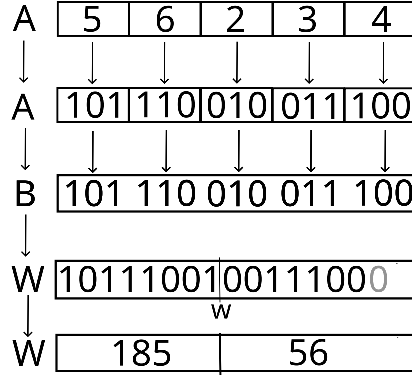

Figura 2.1: Ejemplo de representación de un array A de enteros de tamaño fijo en el que $n = 5$ y cada uno de los elementos es de tamaño $l = 3$ bits. En la segunda fila podemos ver la representación binaria de los elementos de A. El array B es una concatenación de los elementos del array A en binario. Por simplicidad del ejemplo, se utiliza un tamaño de palabra de $w = 8$ bits (en sistemas modernos, $w = 64$ bits). Entonces, el array W se puede ver como una partición del array B en bloques de 8 bits. Por último, el array W se interpreta como un array de $\lceil \frac{n \cdot l}{w} \rceil = 2$ enteros de tamaño $w = 8$ bits

Primero, veremos el caso de que el elemento que queremos leer está completamente contenido en un bloque de w bits del array W. Asumimos que $\lceil \frac{j}{w} \rceil = \lceil \frac{j'}{w} \rceil$, es decir, que tanto j como j' se encuentran dentro del mismo bloque de w bits del array W. Además, estableceremos $r' = ((j' - 1) \bmod w) + 1$ y podemos leer el elemento $A[i]$ en tiempo constante con la siguiente expresión:

$$A[i] = B[j,j'] = \left\lfloor \frac{W[\lceil \frac{j}{w} \rceil]}{2^{w-r'}} \right\rfloor \bmod 2^{j'-j+1}$$

El otro caso que tenemos que contemplar es el de que los bits que queremos leer se encuentren repartidos en dos bloques de w bits consecutivos, por lo que se cumple que $\lceil \frac{j'}{w} \rceil = \lceil \frac{j}{w} \rceil + 1$ y establecemos $r = ((j-1) \bmod w) + 1$. En este caso, podemos leer el elemento $A[i]$ en tiempo constante con la siguiente expresión:

$$A[i] = B[j,j'] = \left\lfloor \frac{W[\lceil \frac{j'}{w} \rceil]}{2^{w-r'}} \right\rfloor + \left(W[\lceil \frac{j}{w} \rceil] \bmod 2^{w-r} \right) \cdot 2^{r'}$$

Por lo tanto, si se realiza la comprobación de si los bits que se quieren leer están contenidos en un único bloque o en dos bloques consecutivos y se aplica la fórmula de lectura correspondiente, la lectura de un elemento $A[i]$ se puede realizar en tiempo constante $\mathcal{O}(1)$ utilizando esta representación compacta. Pero, cabe destacar que, aunque se realice en un tiempo constante, en la práctica la lectura de un único elemento es más lenta que si se utilizase la representación no compacta (es decir, un array clásico de enteros de w bits) puesto que

las operaciones aritméticas que se tienen que realizar introducen costes constantes que están ocultos en la notación asintótica. Aún así, la representación compacta puede llegar a ser más rápida que la representación no compacta en el caso de leer varios elementos consecutivos, puesto que, como se ha dicho anteriormente, estos pueden llegar a tener un mejor uso de la caché, pudiendo compensar el coste extra en ciclos de CPU de las operaciones aritméticas.

La escritura se puede conseguir en tiempo constante de forma análoga a la lectura, estableciendo el valor de las posiciones deseadas en el array W, por lo que se considera trivial y no se entrará en detalle.

2.3.2 Elementos de tamaño variable

En ocasiones, nos interesará que los elementos del array A no tengan un tamaño fijo, sino que cada elemento pueda tener un tamaño distinto. Esto es muy útil cuando por ejemplo, se sabe que los números más pequeños tienen una mayor probabilidad de ocurrencia que los números más grandes, por lo que se puede aprovechar esta información para representar los números más pequeños con un menor número de bits. De esta manera, asignaríamos a cada elemento un identificador (o código) de tamaño variable, reduciendo así el número medio de bits utilizado por el conjunto de datos. Estas ideas se fundamentan en gran medida en la teoría de la información, descrita en la sección 2.1.

Entonces, cada elemento $A[i]$ tendrá una longitud en bits l_i asociada. La lectura de un número se hará de la misma forma que como se hacía con el intervalo $B[j,j']$ en la sección anterior, pero en este caso no podremos calcular los extremos j y j' de dicho intervalo de una forma tan directa como multiplicar el índice i por el tamaño de los elementos l. Si establecemos que $p_i = \sum_{t=1}^{i-1} l_t$, entonces el intervalo de bits asociado al elemento $A[i]$ es $B[1 + p_i, p_i + l_i]$ y sobre este intervalo aplicamos las mismas fórmulas de lectura que en la sección anterior. Es importante destacar que la limitación de que el tamaño l_i máximo que pueden tener los elementos sigue presente y viene determinada por el tamaño de palabra w.

Pero de esta forma, la lectura no se hace en tiempo constante puesto que para poder leer el elemento $A[i]$ necesitamos conocer el valor de p_i, el cual se tiene que calcular como la suma de todas las longitudes de los elementos anteriores. Esto implica que la lectura se haría con una complejidad computacional de $O(n)$, siendo n la longitud del array A.

Para poder alcanzar una solución con mejor complejidad computacional, se hará uso de una técnica llamada sumas parciales, la cual consiste en precalcular los valores de p_i y guardarlos en un array de enteros de w bits, para posteriormente poder acceder a ellos en tiempo constante. El problema en esta técnica, es que, si guardamos los valores p_i para cada uno de los n elementos del array A, el array de sumas parciales tendrá un coste en espacio demasiado grande, $n \cdot w$ bits, lo cual haría que la representación compacta ocupase más espacio que la representación original no compacta.

Una solución a esto es de hacer un muestreo de las sumas parciales, es decir, en lugar de guardar el valor de p_i para todos los n elementos del array A, se guardará el valor de p_i en un array $P[1, \lceil \frac{n}{k} \rceil]$ de enteros de w bits únicamente para uno de cada k elementos, siendo k un parámetro que se puede ajustar para conseguir una compensación entre el espacio ocupado por la representación compacta y el tiempo de acceso a los datos. De esta forma, el valor $1 + P[j]$ será la posición donde empieza el elemento $A[(j-1) \cdot k + 1]$ en el array de bits B. Por ejemplo, si $k = 2$, se guardaría el valor de p_i en P para los elementos $A[1], A[3], A[5] \ldots$

Entonces, se puede calcular el valor p_i de cualquier elemento $A[i]$ de la siguiente forma:

$$p_i = P[\lceil \frac{i}{k} \rceil] + \sum_{t=(\lceil \frac{i}{k} \rceil - 1) \cdot k + 1}^{i-1} l_t$$

Se puede observar que entre $(\lceil \frac{i}{k} \rceil \cdot k + 1)$ y $i - 1$ (extremos del sumatorio) hay como mucho $k - 1$ elementos, por lo que la complejidad computacional de dicha operación es $\mathcal{O}(k)$. De esta forma, hemos obtenido una complejidad mucho mejor que la $\mathcal{O}(n)$ que teníamos anteriormente, puesto que habitualmente $k \ll n$. Una vez tenemos el valor p_i calculado con sumas parciales, su uso se hace exactamente igual que como se ha comentado antes, se leería el elemento $A[i]$ del intervalo $B[1 + p_i, p_i + l_i]$.

Además, cabe destacar que la construcción del array P se puede hacer en tiempo lineal $\mathcal{O}(n)$, puesto que se puede realizar un único recorrido secuencial de A, sumando las longitudes de los elementos que se vayan encontrando y guardando el valor de p_i en P cada vez que se hayan recorrido k elementos de A. Esto hace que la construcción de P no sea costosa en tiempo y sea factible en la práctica para arrays de gran tamaño.

Finalmente, el espacio extra utilizado por el array P es de $w \cdot \lceil \frac{n}{k} \rceil$ bits, siendo el parámetro k el que nos permite hacer un balance entre el espacio ocupado y el tiempo de acceso a los datos. El espacio utilizado por esta representación se puede optimizar si sabemos de antemano cual va a ser el número total de elementos de A y el tamaño de cada uno de ellos, de forma que se puede utilizar la representación compacta de la sección 2.3.1 para el array P, siendo el tamaño fijo de los enteros utilizados $l = \lceil \log_2(1 + \sum_{t=1}^{n} l_t) \rceil$. Con esta optimización se consigue que P ocupe un espacio de $\lceil \frac{\lceil \frac{n}{k} \rceil \cdot \lceil \log_2(1 + \sum_{t=1}^{n} l_t) \rceil}{w} \rceil \cdot w$ bits.

2.4 Vector de bits

Llamaremos vector de bits al array de bits $B[1,n]$ que se introdujo brevemente en la sección 2.3. En concreto, nos interesarán las siguientes operaciones sobre esta estructura:

- $access(B,i)$: Devuelve el valor del bit $B[i]$, cumpliéndose $1 \leq i \leq n$.
- $rank_b(B,i)$: Devuelve el número de bits con valor $b \in \{0,1\}$ que hay en el intervalo

$B[1,i]$, cumpliéndose $1 \leq i \leq n$. Como caso especial, se define $rank_b(B,0) = 0$. Si b se omite, se asume que $b = 1$.

- $select_b(B,j)$: Devuelve la posición del j-ésimo bit con valor $b \in \{0,1\}$ en el vector B, para todo $j \geq 0$. Como caso especial, se define $select_b(B,0) = 0$ y $select_b(B,j) = n+1$ si $j > rank_b(B,n)$. De nuevo, si b se omite, se asume que $b = 1$.

El vector de bits es el componente más básico y sobre el cual se construyen la mayor parte de las estructuras de datos compactas. Las operaciones anteriormente descritas son fundamentales para poder construir y realizar consultas en estructuras más complejas, por lo que el rendimiento de las operaciones sobre los vectores de bits determinará en gran medida el rendimiento de las estructuras de datos que los utilicen. Por lo tanto, es imperativo proveer de implementaciones que sean eficientes tanto en espacio como en tiempo para formar una base sólida sobre la que construir estructuras de datos compactas de más alto nivel.

Conceptualmente, se puede ver al vector de bits como un array compacto de enteros de tamaño fijo (descrito en la sección 2.3.1) en el que el tamaño de los elementos es de $l = 1$ bit. En la práctica, nos podemos aprovechar de que sabemos de antemano el tamaño que tendrán los elementos y podremos realizar una implementación especializada que sea más eficiente que la implementación general de la sección 2.3.1. En concreto, se pueden optimizar en gran medida las operaciones de lectura y escritura, teniendo en cuenta de que únicamente trataremos el caso de leer o escribir un único bit. Además, cabe destacar que llamaremos representación plana a esta representación del vector de bits, puesto que más adelante introduciremos una representación comprimida.

Por último, es importante mencionar que, tanto aquí como en las próximas secciones, se considerarán únicamente estructuras de datos inmutables (o estáticas), que no se podrán modificar tras su construcción, y no su versión mutable (o dinámica). Esto es debido a que permitir que se puedan modificar los vectores de bits después de que se construyan las estructuras de soporte para que las operaciones de $rank$ y $select$ sobre ellos sean eficientes hace que mantener la consistencia de las estructuras de soporte con dicho vector sea mucho más complicado y costoso en términos de espacio y tiempo. Esto es habitual en librerías de estructuras de datos compactas, como la ya mencionada SDSL.

2.4.1 Rank

En esta sección veremos dos estructuras de soporte para realizar la operación de $rank$ de forma eficiente sobre vectores de bits en su representación plana. De aquí en adelante, únicamente se contemplarán explicaciones para la operación de $rank_1(B,i)$, puesto que $rank_0(B,i) = i - rank_1(B,i)$.

Rank con un nivel de directorio

Conceptualmente, podemos definir $rank_1(B,i) = \sum_{j=1}^{i} B[j]$ si consideramos a los bits del vector B como números. Sin embargo, esta definición no es eficiente puesto que para poder calcular el valor de $rank_1(B,i)$ se tienen que recorrer todos los bits desde la posición 1 hasta la posición i, lo cual implica una complejidad computacional de $\mathcal{O}(n)$.

Este problema es muy similar a lo que ocurría en la sección 2.3.2 para calcular el valor p_i de un elemento $A[i]$ en un array de enteros de tamaño variable. Volveremos a aplicar aquí la técnica de sumas parciales, realizando un muestreo del $rank$ de forma que guardaremos la respuesta a la consulta $rank_1(B,i) = \sum_{j=1}^{i} B[j]$ para las posiciones del vector B que sean múltiplo de $s = k \cdot w$ (es decir, cada s bits), siendo k un parámetro que se puede ajustar para conseguir una compensación entre el espacio ocupado por la estructura de soporte y el tiempo empleado por la operación de $rank$.

Llamaremos a esta estructura de soporte **rank con un nivel de directorio** [40], y estará formada por un array $R[0, \lfloor \frac{n}{s} \rfloor]$ de enteros de w bits, de forma que la posición $R[j]$ contendrá el valor de $rank_1(B,j \cdot s)$, es decir, el número de bits con valor 1 que hay en el intervalo $B[1, j \cdot s]$. Además, llamaremos superbloque a cada uno de los trozos de s bits en los que se puede dividir el vector B, por lo tanto, en R se guardará el número de unos contenido en cada uno de los $\lfloor n/s \rfloor$ superbloques de B. Se puede ver un ejemplo de esta estructura de soporte en la figura 2.2.

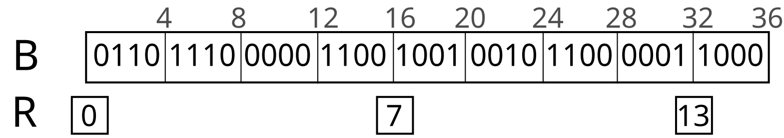

Figura 2.2: Ejemplo de la estructura de $rank$ con un nivel de directorio con un valor $k = 4$ sobre un vector de bits B de tamaño $n = 36$. Por simplicidad del ejemplo, se establece $w = 4$. En la primera fila se puede ver el vector B y en la segunda fila el array R que acumula el valor de $rank$ cada $4 \cdot 4 = 16$ bits

Entonces, definimos $i_s = \lfloor \frac{i}{s} \rfloor$ y podremos calcular el valor de $rank_1(B,i)$ de la siguiente forma:

$$rank_1(B,i) = R[i_s] + \sum_{t=i_s \cdot s+1}^{i} B[t]$$

Esta solución tiene una complejidad computacional de $\mathcal{O}(s) = \mathcal{O}(k \cdot w)$ puesto que en el peor caso, tendremos que sumar secuencialmente s bits. Aunque esta solución es mucho mejor que la que teníamos con la definición básica de $rank$ (que era de $\mathcal{O}(n)$), se puede mejorar todavía más sabiendo que la representación de nuestro vector de bits se hace de forma compacta utilizando el array W.

Puesto que guardamos los bits del vector B en el array W de enteros de w bits, la expresión $\sum_{t=i_s \cdot s+1}^{i} B[t]$ se corresponde con contar el número de unos que hay en las palabras $W[i_s \cdot k + 1, \lfloor \frac{i}{w} \rfloor]$ y en los primeros i mod w bits más significativos de la palabra $W[\lfloor \frac{i}{w} \rfloor + 1]$ (se pueden obtener dichos primeros bits de forma sencilla utilizando operaciones de desplazamiento y enmascaramiento de bits en tiempo constante).

El problema de contar el número de unos en la representación binaria de números enteros se llama *population count*, o simplemente *popcount*. La operación de *popcount* [41] está disponible de forma nativa y eficiente en muchos lenguajes de programación (en especial, en los de bajo nivel), pero es especialmente interesante ver que también está disponible como una instrucción del procesador implementada en hardware en muchos sistemas Intel y AMD [42], lo que hace que sea una operación extremadamente rápida (en tiempo constante $\mathcal{O}(1)$) y eficiente para realizar sobre enteros de w bits.

Por lo tanto, si utilizamos la operación de *popcount* en tiempo constante para contar el número de unos en las palabras, la operación de $rank$ se puede definir de la siguiente forma:

$$rank_1(B,i) = R[i_s] + \left(\sum_{t=i_s \cdot k+1}^{\lfloor \frac{i}{w} \rfloor} popcount\,(W[t]) \right) + popcount \left(\left\lfloor \frac{W\left[\lfloor \frac{i}{w} \rfloor + 1\right]}{2^{w-(i \bmod w)}} \right\rfloor \right)$$

Consiguiendo entonces una complejidad computacional de $\mathcal{O}(k)$ para la operación de $rank$, puesto que a lo sumo se tendrán que recorrer secuencialmente $k-1$ enteros de w bits (esto se puede ver en el sumatorio de la expresión anterior).

Además, la construcción del array R se puede realizar en tiempo lineal $\mathcal{O}(n)$, puesto que para ello se realiza un único recorrido secuencial de B, sumando los unos que se vayan encontrando y guardando el valor de $rank_1(B,i)$ que hemos acumulado en R cada vez que se hayan recorrido s bits de B.

Finalmente, el espacio extra utilizado por esta estructura de soporte es de exactamente $w \cdot \lfloor \frac{n}{s} \rfloor$ bits, que viene determinado por el tamaño del array R. Se puede observar que el sobrecoste en espacio de esta estructura es de $\frac{1}{k}$ bits por cada bit del vector de bits B, por lo que se puede ajustar el parámetro k para conseguir una compensación entre el espacio ocupado y el tiempo que se emplea en realizar la consulta de $rank$. Por ejemplo, utilizando $k = 4$, se consigue que el espacio extra ocupado sea un 25 % del tamaño del vector de bits B, mientras que si $k = 20$, el espacio extra será de un 5 %.

Una optimización en espacio que se puede realizar en este caso es la de la compactación del vector R. Si establecemos m como el número de unos en el vector B, $m = rank_1(B,n)$, entonces el valor máximo que se va a almacenar en el vector R es m. Por lo tanto, podemos utilizar la representación compacta de arrays de enteros de la sección 2.3.1 con un tamaño

de $l = \lceil \log_2(m+1) \rceil$ bits. El problema es que para esto, tenemos que realizar un recorrido secuencial adicional en la construcción para poder calcular previamente el valor de m, haciéndola el doble de lenta. Además, el acceso al elemento $R[i]$ utilizando la representación compacta puede traer desventajas en tiempo, como ya se ha mencionado en otras ocasiones, por lo que en la práctica quizá no sea beneficioso emplear esta optimización.

Rank con dos niveles de directorio

Aunque la estructura de la sección anterior nos permite realizar la operación de $rank$ en un tiempo aceptable $\mathcal{O}(k)$, se puede mejorar todavía más, pudiendo alcanzar una solución en tiempo constante $\mathcal{O}(1)$ que sea mucho más rápida en la práctica, a expensas de ocupar más espacio extra.

La idea consiste en volver a aplicar una vez más la técnica utilizada en la estructura de soporte anterior, pero en lugar de utilizar un único nivel de directorio, utilizaremos dos niveles de directorio y llamaremos a esta estructura de soporte **rank con dos niveles de directorio** [40, 43] [44, 45], o simplemente $rank$ en tiempo constante.

En esta estructura se utilizará el mismo array $R[0, \lfloor \frac{n}{s} \rfloor]$ de enteros de w bits que se utilizaba en el $rank$ con un nivel de directorio, definiendo $s = k \cdot w$ para un parámetro k. Además, se utilizará un segundo array $R'[0, \lfloor \frac{n}{w} \rfloor]$ compacto de enteros de tamaño fijo donde $l = \lceil \log_2(s - w + 1) \rceil$.

Para la interpretación de este segundo array, dividiremos conceptualmente cada uno de superbloques de tamaño s en trozos de tamaño w a los que llamaremos bloques (que coinciden con cada uno de los elementos de W). Entonces, el array R' contendrá el número de unos que hay en cada uno de los bloques de B, pero de forma relativa al superbloque al que pertenecen, esto es, $R'[i] = rank_1(B, i \cdot w) - R[\lfloor \frac{i \cdot w}{s} \rfloor]$. Por lo tanto, $R'[i]$ indica el número de unos que hay hasta el i-ésimo bloque, contando desde el inicio de su superbloque correspondiente. Se puede ver un ejemplo de esta estructura de soporte en la figura 2.3.

		4	8	12	16	20	24	28	32	36
B	0110	1110	0000	1100	1001	0010	1100	0001	1000	
R'	0	2	5	5	0	2	3	5	0	1
R	0				7				13	

Figura 2.3: Ejemplo de la estructura de rank con dos niveles de directorio con un valor $k = 4$ sobre un vector de bits B de tamaño $n = 36$. Por simplicidad del ejemplo, se establece $w = 4$. En la primera fila se puede ver el vector B, en la segunda fila podemos ver el array R' que acumula cada $w = 4$ bits el valor del $rank$ del inicio del bloque relativo a su superbloque correspondiente y en la tercera fila se puede ver el array R que acumula el valor del $rank$ del inicio de cada superbloque cada $k \cdot w = 4 \cdot 4 = 16$ bits

De esta interpretación se justifica el tamaño l definido anteriormente, puesto que dentro de cada superbloque habrá a lo sumo s bits con valor 1 y el valor máximo de $rank$ relativo al superbloque que guardaremos será $s - w$ (debido a que almacenamos el valor de $rank$ al inicio de cada bloque) y por lo tanto, se necesitarán $l = \lceil \log_2(s - w + 1) \rceil$ bits para poder representar dicho valor.

Entonces, se puede ver que $R'[\lfloor \frac{i}{w} \rfloor]$ indica el número de unos que hay en el intervalo $B[\lfloor \frac{i}{s} \rfloor \cdot s + 1, \lfloor \frac{i}{w} \rfloor \cdot w]$ y podemos definir la operación de $rank$ como:

$$rank_1(B,i) = R[\lfloor \frac{i}{s} \rfloor] + R'[\lfloor \frac{i}{w} \rfloor] + popcount \left(\left\lfloor \frac{W[\lfloor \frac{i}{w} \rfloor + 1]}{2^{w-(i \bmod w)}} \right\rfloor \right)$$

De esta forma, se consigue así una complejidad computacional en tiempo constante $\mathcal{O}(1)$, puesto que todas las operaciones que se realizan son en tiempo constante (indexado de un array clásico de enteros, lectura de un elemento de un array compacto de enteros de tamaño fijo, *popcount* y operaciones aritméticas sobre enteros). Además, la construcción de esta estructura de soporte se puede realizar en tiempo lineal $\mathcal{O}(n)$ de forma muy similar a la del $rank$ con un nivel de directorio, la única diferencia es que en este caso hay que almacenar también los valores de rank de cada bloque relativo a su superbloque en el vector R', que se calculan utilizando la operación de *popcount* en cada bloque (palabra de w bits) de B, de la forma descrita anteriormente.

Finalmente, el espacio ocupado por esta estructura es de exactamente $w \cdot \lfloor \frac{n}{s} \rfloor + \lceil \frac{l \cdot \lfloor \frac{n}{w} \rfloor}{w} \rceil \cdot w$ bits, que se corresponde con el espacio ocupado por los arrays R y R' respectivamente. Esta estructura es muy adecuada cuando queremos que la operación de $rank$ sea muy rápida y el espacio que debe ocupar no es crítico. En la práctica, un valor típico que se suele utilizar que nos ofrece una buena relación entre espacio y tiempo es $k = 4$, lo que haría que esta estructura ocupe un espacio extra del 37,5 % sobre tamaño del vector de bits B.

2.4.2 Select

La operación de $select_b$ se puede ver como la inversa de la operación de $rank_b$, puesto que se cumple que $rank_b(B, select_b(B,i)) = j$.

Debido a que $rank$ es una función monótona creciente, se puede resolver $select_b(B,j)$ realizando una búsqueda binaria sobre el vector de bits B para encontrar la posición i que esté más a la izquierda tal que se cumpla $rank_b(B,i) = j$. Sin embargo, pese a que esta solución de alto nivel se puede aplicar a cualquier estructura que soporte $rank$, se puede realizar de forma más eficiente si sabemos como está implementada la estructura de soporte de $rank$. En las próximas secciones, veremos cómo se puede hacer que las estructuras de rank en un nivel y en dos niveles de directorio soporten la operación de $select$ de forma eficiente. También es posible implementar estructuras especializadas para la operación de $select$ que sean más

rápidas que las que se verán a continuación, pero estas son mucho más complejas y no se tendrán en cuenta en este trabajo.

Rank con un nivel de directorio

Para implementar la operación de $select$ en esta estructura, realizaremos primero una búsqueda binaria sobre el array R para encontrar el superbloque en el que se alcance o sea superado el valor de $rank$ deseado. Una vez encontrado el superbloque, se realizará una búsqueda secuencial en los bloques que lo contienen (utilizando la operación de *popcount*) para encontrar el bloque en el que, de nuevo, se alcance o sea superado el valor de $rank$ que se está buscando. Después de estos dos pasos, se habrá encontrado el bloque de w bits en el que se encuentra el bit cuyo valor de $rank$ es el deseado, por lo que tendremos que hacer una operación de $select$ dentro de una palabra. Para hacer esto, lo más sencillo es realizar una búsqueda secuencial en los w bits, empleando desplazamientos y enmascaramiento de bits para poder acceder a cada uno de ellos e iterar hasta que se encuentre la posición j deseada que cumpla $rank(B,j) = i$. Para esto último también se pueden utilizar algoritmos más eficientes y complejos que nos permitan hacer $select$ dentro de una palabra [41].

Esta implementación tiene una complejidad computacional de $\mathcal{O}(\log \frac{n}{k} + k + w)$, puesto que se tiene que realizar una búsqueda binaria sobre R, una búsqueda secuencial sobre los posibles k bloques dentro del superbloque y finalmente una búsqueda secuencial sobre los w bits dentro de la palabra. La ventaja de esta implementación de $select$ es que es muy sencilla de implementar encima de esta estructura de soporte y no necesita ningún espacio extra del que ya se utiliza para las consultas de $rank$. En la práctica, utilizar valores grandes para k hace que se degrade mucho el tiempo de consulta, puesto se gastaría la mayor parte del tiempo de consulta realizando la búsqueda secuencial sobre los k bloques dentro de un superbloque.

Rank con dos niveles de directorio

La implementación de $select$ en esta estructura es muy similar a la de la estructura de $rank$ con un nivel de directorio, pero se puede hacer una optimización sobre la búsqueda secuencial dentro del superbloque. Podemos utilizar el array R' para volver a realizar el mismo tipo de búsqueda binaria que se hace sobre el array R, para así encontrar el bloque en el que se alcance o sea superado el valor de $rank$ deseado. Una vez encontrado el bloque, se realiza el mismo paso descrito en la sección anterior.

Por lo tanto, la complejidad de este algoritmo es de $\mathcal{O}(\log \frac{n}{k} + \log k + w)$. Se puede notar que la diferencia entre esta implementación y la anterior es que se ha sustituido el término de la búsqueda secuencial sobre los k bloques ($\mathcal{O}(k)$) por una búsqueda binaria sobre esos bloques ($\mathcal{O}(\log k)$), lo que hace que este algoritmo sea mucho más rápido y el tiempo de consulta se degrade mucho menos para valores grandes de k. Cabe destacar que en la práctica

si utilizamos valores pequeños de k, es posible que, debido a la sobrecarga de realizar una búsqueda binaria sobre los bloques, la implementación que realice una búsqueda secuencial sea más rápida.

2.5 Vector de bits comprimido con $\mathbf{H_0}$

En ocasiones, el vector B puede tener muchos más 0s que 1s o viceversa, es decir, está desbalanceado. En esas ocasiones podemos aprovecharnos de esta información sobre los datos para comprimir el vector de bits de forma que utilice un espacio cercano a su entropía empírica de orden cero, $nH_0(B)$ bits, y que siga permitiendo realizar las operaciones básicas de $access$, $rank$ y $select$ de forma eficiente. Utilizaremos una representación que nos permita realizar esta compresión y la llamaremos **vector de bits comprimido con *H_0*** [46] [1, 47].

Primero, dividiremos conceptualmente el vector B en bloques de tamaño b bits, siendo b un parámetro. Cada bloque $B_i = B[(i-1) \cdot b + 1, i \cdot b]$ tendrá una clase asociada $c_i = \sum_{j=(i-1)\cdot b+1}^{i\cdot b} B[j]$, que se corresponde con el número de unos que hay en dicho bloque [48]. Cada clase tendrá un distinto número de elementos que pertenecen a ella. En concreto, la clase c_i contendrá exactamente $\binom{b}{c_i} = \frac{b!}{c_i!(b-c_i)!}$ elementos, que coincide con el número maneras en las que se pueden elegir las c_i posiciones de los 1s entre las b posiciones dentro del bloque.

Entonces, se puede identificar de forma única un determinado bloque (secuencia de 1s y 0s de tamaño b) utilizando el par (c_i,o_i) [46], donde o_i se corresponde con un desplazamiento. El desplazamiento es un número que sirve para identificar a un determinado bloque entre los pertenecientes a su misma clase. Por lo tanto el desplazamiento debe cumplir que $0 \leq o_i < \binom{b}{c_i}$.

Puesto que ya tenemos una forma de identificar a un determinado bloque del vector B, la representación comprimida del vector b estará formada por un array $C[1, \lceil \frac{n}{b} \rceil]$, donde $C[i] = c_i$ y un array $O[1, \lceil \frac{n}{b} \rceil]$, donde $O[i] = o_i$. Para el array C se utilizará un array compacto de enteros de tamaño fijo donde $l = \lceil \log_2(b+1) \rceil$ bits, puesto que el valor máximo que se guardará en C será b.

A continuación, de forma que se pueda obtener compresión con esta representación, para el array O tendremos que utilizar un array compacto de enteros de tamaño variable, puesto que, como se ha mencionado anteriormente, cada clase tendrá un número diferente de elementos (identificados por el desplazamiento o_i), y dicho número de elementos crece de forma factorial con el tamaño del bloque b. Utilizaremos la representación descrita en la sección 2.3.2, donde la longitud l_i en bits de cada elemento i dependerá de la clase o_i a la que pertenezca dicho elemento, y será exactamente:

$$l_i = L[c_i] = |o_i| = \lceil \log_2 \binom{b}{c_i} \rceil$$

Se puede ver que se utiliza un pequeño array $L[0,b]$ de enteros de tamaño fijo donde $l = \lceil\log_2(\lceil\log_2\binom{b}{b/2}\rceil + 1)\rceil$, de forma que se almacena en la posición $L[i]$ el número de bits necesarios para codificar en binario el desplazamiento de un bloque de clase c_i. Utilizando este array precalculado, se evita tener que calcular el número de bits necesarios para codificar el desplazamiento de un bloque en tiempo de ejecución.

Cabe mencionar que esta representación de los vectores de bits tendrá un parámetro k, que se utilizará en el array O de la misma forma que se explicó en la sección 2.3.2, sirviendo para hacer un balance entre el espacio ocupado y el tiempo de acceso a los datos de este array. Posteriormente veremos que este parámetro k también se utilizará para añadir el soporte de $rank$ a esta estructura.

Codificación y decodificación

Hasta ahora, no hemos visto cómo se asocia un desplazamiento a un elemento que pertenezca a una clase determinada. Para ello, asignaremos los desplazamientos en orden numérico creciente de los valores de los bloques, interpretados como números enteros. Es decir, al bloque con valor numérico más pequeño se le asignará el desplazamiento 0, al siguiente el desplazamiento 1, y así sucesivamente hasta llegar al último bloque, al que se le asignará el desplazamiento $\binom{b}{c_i} - 1$. Se puede ver de forma sencilla que hay $\binom{b-1}{c}$ bloques que empiezan (en su representación binaria) con un 0, que preceden a $\binom{b-1}{c-1}$ bloques que empiezan con un 1. De esta forma, se puede asignar un desplazamiento o_i al contenido de un bloque B_i de la clase c_i, y recuperar el bloque B_i a partir de su desplazamiento o_i y su clase c_i.

El algoritmo para asignar el contenido de un bloque B_i de clase c_i a un desplazamiento o_i utilizando un tamaño de bloque b es el siguiente [1]: primero, establecemos $\mathcal{B} = B_i$, $c = c_i$ y $o_i = 0$ y a continuación comprobamos el primer bit de $\mathcal{B}$. Si $\mathcal{B}$ es de la forma $\mathcal{B} = 1.\mathcal{B}'$, quiere decir que $\mathcal{B}$ empieza por un 1, por lo que se situará por delante de los $\binom{b-1}{c}$ bloques que empiezan por un 0, estableceremos $o_i \leftarrow o_i + \binom{b-1}{c}$ y repetimos el proceso con $\mathcal{B} \leftarrow \mathcal{B}'$, $c \leftarrow c - 1$ y $b \leftarrow b - 1$. En cambio, si $\mathcal{B}$ es de la forma $\mathcal{B} = 0.\mathcal{B}'$, quiere decir que $\mathcal{B}$ empieza por un 0, por lo que no se modifica ni o_i ni c y repetimos el proceso con $\mathcal{B} \leftarrow \mathcal{B}'$ y $b \leftarrow b - 1$. El proceso finaliza cuando se cumpla que $b = 0$ o $c = 0$, teniendo calculado en o_i el desplazamiento correspondiente al bloque B_i. Llamaremos a este algoritmo $encode(B_i)$, el cual nos devolverá la codificación del bloque B_i, es decir, el par (c_i,o_i) que lo identifica.

De forma análoga, para decodificar un bloque B_i de clase b a partir de su codificación (c_i,o_i) realizamos el algoritmo de forma inversa [1], empezando a decodificar por el bit más significativo, es decir, de izquierda a derecha. Primero, empezamos con $c = c_i$, $o = o_i$ y $\mathcal{B} = \epsilon$ (cadena vacía), si se cumple que $o < \binom{b-1}{c}$, entonces en este punto $\mathcal{B}$ termina por un 0, por lo que continuamos con $b \leftarrow b - 1$ y $\mathcal{B} \leftarrow \mathcal{B}.0$. En cambio, si $o \geq \binom{b-1}{c}$, entonces en este punto $\mathcal{B}$ termina por un 1, por lo que continuamos con $o \leftarrow o - \binom{b-1}{c}$, $c \leftarrow c - 1$, $b \leftarrow b - 1$

y $\mathcal{B} \leftarrow \mathcal{B}.1$. Este algoritmo termina cuando se alcanza el final del bloque, es decir, cuando $b = 0$. Llamaremos a este algoritmo $decode(c_i,o_i)$, el cual nos devolverá el bloque B_i a partir de su codificación (c_i,o_i).

2.5.1 Access

Una vez descrita la representación de esta estructura, la podremos ver como un vector de bits $B_{H_0}[1,n]$ en el que la operación de lectura ($access$) se puede realizar de la siguiente forma:

$$B[j] = access(B,j) = \frac{decode(C[\lceil \frac{j}{b} \rceil],O[\lceil \frac{j}{b} \rceil])}{2^{b-(j \bmod b)}} \bmod 2$$

Se puede ver que el elemento j del vector B_{H_0} se corresponde con el bit $j \bmod b$ del bloque $B_{\lceil \frac{j}{b} \rceil}$ (empezando a contar de izquierda a derecha), por lo que una vez decodificado dicho bloque, se tiene que hacer un desplazamiento de $b-(i \bmod b)$ bits hacia la derecha del mismo y tomar el bit menos significativo.

Finalmente, se puede determinar que la lectura es rápida y eficiente [49, 50], puesto que la operación de $decode$ se puede realizar en tiempo $\mathcal{O}(b)$, el acceso a los elementos del array C (sección 2.3.1) se realiza en tiempo constante $\mathcal{O}(1)$ y el acceso a los elementos del array O (sección 2.3.2) se realiza en tiempo $\mathcal{O}(k)$, por lo que la operación de lectura en este vector de bits se puede realizar con una complejidad computacional de $\mathcal{O}(b + k)$. En la práctica, el parámetro b está limitado por la capacidad del sistema para realizar la operación de *popcount* de forma eficiente, por lo que habitualmente suele tomar valores como 63, 31 o 15 bits, y el parámetro k se puede ajustar para conseguir una compensación entre el espacio ocupado y el tiempo de acceso a los datos.

2.5.2 Rank y Select

La estructura auxiliar que utilizaremos en este vector de bits para soportar la operación de $rank$ es la estructura de $rank$ con un nivel de directorio, vista en la sección 2.4.1. Utilizaremos el mismo parámetro k que se utiliza en el array O para el array R de esta estructura auxiliar.

La interpretación del array R para soportar las operaciones de $rank$ y $select$ se hará de forma muy similar, pero, en este caso, en lugar de obtener los elementos empleando el array W, tendremos que decodificar los bloques. Además, ya no estableceremos $s = k \cdot w$, puesto que los bloques en los que dividimos el vector B ya no son de w bits como en el caso de los vectores planos, sino que utilizaremos bloques de tamaño b bits. Por lo tanto, estableceremos $s = k \cdot b$ y se construirá el array R de forma que se guarde el valor de $rank$ cada k bloques de b bits del vector B_{H_0}, leyendo el valor de $C[i]$ para cada bloque en lugar de decodificar el bloque completamente y emplear la operación de $popcount$.

Entonces, la operación de $rank$ se puede definir de la siguiente forma:

$$rank_1(B_{H_0},i) = R[\lfloor \frac{i}{s} \rfloor] + \left(\sum_{t=\lfloor \frac{i}{s} \rfloor \cdot k+1}^{\lceil \frac{i}{b} \rceil -1} C[t] \right) + popcount\left(\left\lfloor \frac{decode(C[\lceil \frac{i}{b} \rfloor], O[\lceil \frac{i}{b} \rceil])}{2^{b-(i \bmod b)}} \right\rfloor \right)$$

La implementación de $select$ se hace de forma análoga, combinando las ideas expuestas en esta sección sobre la decodificación del bloque junto con lo ya descrito sobre el $select$ para esta estructura de $rank$ con un nivel de directorio.

Por último, debido a que esta representación tiene como principal objetivo comprimir el vector de bits original, y no busca tanto la velocidad en las operaciones de $rank$ y $select$, se puede utilizar la optimización en espacio mencionada en la sección 2.4.1, donde utilizaremos para representar el array R un array compacto de enteros, donde la longitud de cada elemento será de $l = \lceil \log_2(m+1) \rceil$ bits, siendo m el número de unos que hay en el vector B_{H_0}. Esta optimización en espacio no afecta a la complejidad computacional de las operaciones, pero sí que puede afectar al tiempo de acceso a los datos, debido a que utilizar un array compacto de enteros implica que se tengan que realizar varias operaciones aritméticas antes de acceso a los datos. Esta optimización, por lo tanto, hace que esta representación utilice un menor espacio, a expensas de que las operaciones de $rank$ y $select$ sean más lentas.

2.6 Wavelet Tree

Las operaciones de $rank$, $select$ y $access$ se pueden generalizar a secuencias arbitrarias de símbolos S con un alfabeto Σ de tamaño σ. Entonces, dada una secuencia de símbolos $S = s_1 s_2 \ldots s_\sigma$ y cualquier símbolo $s \in \Sigma$:

- $rank_s(S,i)$: Devuelve el número de ocurrencias del símbolo s en la secuencia S hasta la i-ésima posición, es decir, en el intervalo $S[1,i]$.
- $select_s(S,j)$: Devuelve la posición de la j-ésima ocurrencia del símbolo s en la secuencia S.
- $access(S,i)$: Devuelve el i-ésimo símbolo de la secuencia S, s_i.

Una solución sencilla para poder soportar estas operaciones es construir σ vectores de bits (uno por cada símbolo del alfabeto), cada uno de longitud $|S|$, cumpliéndose que $B_i[j] = 1$ si $S[j] = s_i$ y $B_i[j] = 0$ en caso contrario. De esta forma, podremos construir las estructuras de soporte de $rank$ y $select$ para cada uno los n vectores de bits, y definimos

$rank_s(S,i) = rank_1(B_i,i)$ y $select_s(S,j) = select_1(B_i,j)$. Sin embargo, esta solución tiene un coste en espacio del orden de $\mathcal{O}(\sigma \cdot |S|)$, lo que hace que no sea viable en la práctica. Veremos que utilizando un *Wavelet Tree* podremos realizar estas operaciones utilizando un espacio del orden de $\mathcal{O}(|S| \cdot \log \sigma)$, lo que hace que sea una solución mucho más eficiente.

Un *Wavelet Tree* [10, 51, 52] es un árbol binario balanceado que se construye de forma recursiva, dividiendo en cada nivel el alfabeto Σ en dos partes, de forma que cada símbolo de un alfabeto $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ estará asociado a un nodo hoja del árbol.

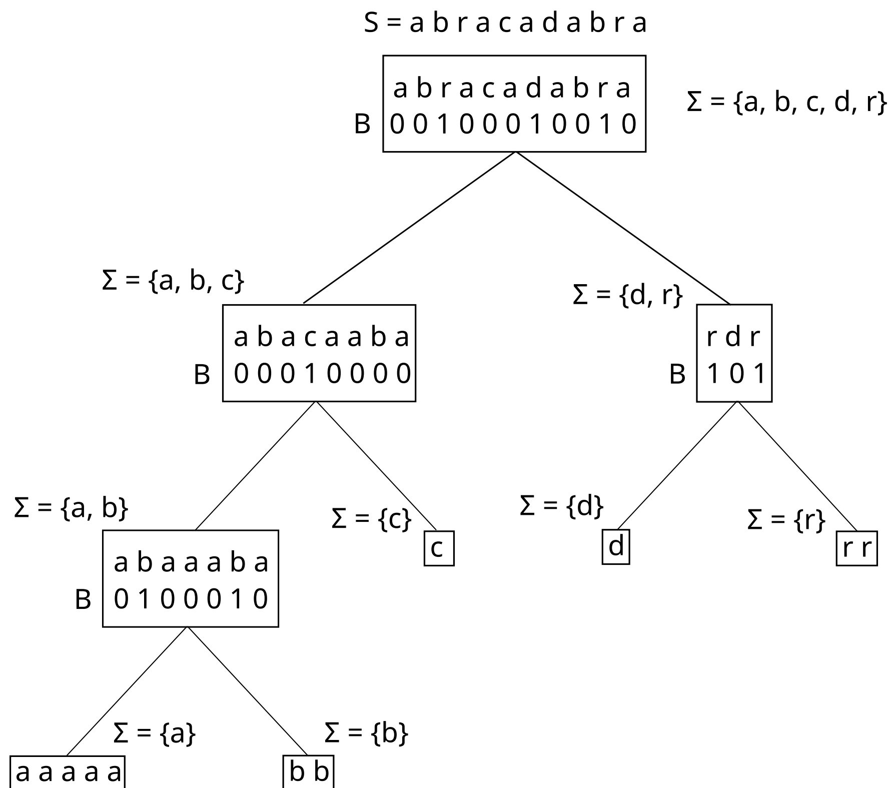

Figura 2.4: Ejemplo de un *Wavelet Tree* construido a partir de la secuencia $S = abracadabra$ y el alfabeto $\Sigma = \{a,b,c,d,r\}$. Se muestra la subcadena asociada a cada nodo por claridad, pero esta no se almacena. Tampoco se almacena en los nodos su alfabeto asociado, puesto que esto está implícito en la forma del árbol

Dada una secuencia S formada por símbolos de un alfabeto Σ, el *Wavelet Tree* se construye de la siguiente forma. Primero comenzamos con el nodo raíz del árbol y utilizaremos la secuencia inicial S y su alfabeto Σ. Posteriormente, se realizará una partición del alfabeto Σ en dos partes, $\Sigma_L = \{s_1, s_2, \ldots, s_{\sigma/2}\}$ y $\Sigma_R = \{s_{\sigma/2+1}, s_{\sigma/2+2}, \ldots, s_\sigma\}$, de forma que se asignará al nodo actual un vector de bits B de tamaño $|S|$, donde $B[i] = 0$ si $S[i] \in \Sigma_L$ y $B[i] = 1$ si $S[i] \in \Sigma_R$. Sobre este vector de bits, se podrán construir las estructuras auxiliares que se deseen, para posteriormente realizar las operaciones de $rank$ y $select$ de forma efi-

ciente (por ejemplo, utilizando la estructura de $rank$ con un nivel de directorio, la estructura de $rank$ en tiempo constante). A continuación, se dividirá la secuencia S en dos partes, las subsecuencias S_L y S_R, de forma que S_L contendrá los símbolos de S que pertenezcan a Σ_L y S_R contendrá los símbolos de S que pertenezcan a Σ_R, pero manteniendo dentro de cada subsecuencia el orden original que tenían los símbolos en S. El siguiente paso es realizar una llamada recursiva para construir el subárbol izquierdo del nodo actual con el alfabeto Σ_L y la secuencia S_L, y otra llamada recursiva para construir el subárbol derecho con el alfabeto Σ_R y la secuencia S_R, realizando el mismo proceso que se hizo para el nodo raíz. El proceso se repite recursivamente hasta que se alcancen los nodos hoja del árbol, donde la secuencia es una repetición de un mismo símbolo.

Los nodos internos únicamente almacenarán su vector de bits correspondiente y sus nodos hijo izquierdo y derecho, no siendo necesario que se almacene el alfabeto Σ asociado a dicho nodo interno, puesto esto está implícito dado un alfabeto inicial y la forma del árbol. En los nodos hoja se almacenará únicamente la longitud de la subsecuencia (la cual estará formada por un único símbolo) que representa dicho nodo hoja. Finalmente, se debe guardar el alfabeto Σ asociado con la secuencia S inicial junto con la raíz del árbol, de manera que combinando el alfabeto inicial junto con la topología del árbol, se pueda calcular el alfabeto Σ correspondiente a cada nodo. En la figura 2.4 se puede ver un ejemplo de un *Wavelet Tree* construido a partir de la secuencia $S = abracadabra$.

2.6.1 Access

El *Wavelet Tree* es una representación autosuficiente de la secuencia S, por lo que los elementos de la secuencia original siguen siendo accesibles. Para obtener el símbolo i de la secuencia S, empezaremos leyendo el bit $b_i = B[i]$ del vector de bits B del nodo raíz. El bit b_i nos indicará si el símbolo a buscar se encuentra en el subárbol izquierdo (si $b_i = 0$) o en el subárbol derecho (si $b_i = 1$). Además, $rank_{b_i}(B,i)$ nos indicará en qué posición del vector de bits del subárbol correspondiente se encuentra el símbolo i que estamos buscando. A continuación se realiza una llamada recursiva para repetir este proceso, estableciendo como nodo actual el nodo hijo correspondiente y la nueva posición $i \leftarrow rank_{b_i}(B,i)$. Esto se repite hasta que se alcance un nodo hoja, donde se devolverá finalmente el símbolo que representa dicho nodo para dar respuesta a la consulta $access(S,i)$.

La complejidad de esta operación es muy dependiente de estructura de soporte que se utilice para realizar la operación de $rank$, puesto que realizaremos $\mathcal{O}(\log \sigma)$ llamadas recursivas, y en cada una de ellas se realizará una operación de $rank$. Por ejemplo, si utilizamos la estructura de $rank$ con un nivel de directorio, la complejidad computacional de esta operación será $\mathcal{O}(k \log \sigma)$ y si utilizamos la estructura de $rank$ en tiempo constante, la complejidad computacional será $\mathcal{O}(\log \sigma)$.

2.6.2 Rank

La operación de $rank$ se resuelve de forma similar a la de $access$. Para responder a la consulta $rank_s(S,i)$, se comienza en el nodo raíz y si $s \in \Sigma_L$, se realiza una llamada recursiva por el subárbol izquierdo para repetir este proceso, estableciendo una nueva posición $i \leftarrow rank_0(B,i)$ utilizando el vector de bits B del nodo actual. De forma similar, si $s \in \Sigma_R$, se realiza una llamada recursiva por el subárbol derecho, estableciendo $i \leftarrow rank_1(B,i)$. Esto se repite hasta que se alcance un nodo hoja, donde el símbolo que representa dicho nodo será el símbolo s que estamos buscando y la posición final i en este nivel de la recursividad será el valor de respuesta a la consulta $rank_s(S,i)$.

2.6.3 Select

La consulta de $select_s(S,j)$ se resuelve siguiendo el proceso inverso de los anteriores. Primero, empezaremos en el nodo hoja que representa al símbolo s, donde sabemos que en este punto la j-ésima ocurrencia del símbolo s ocurre en la posición j de la subcadena asociada a este nodo (puesto que la secuencia es una repetición de un mismo símbolo). A continuación, se realiza una llamada recursiva a su nodo padre, en el cual si el símbolo s se encuentra en el subárbol izquierdo (es decir, $s \in \Sigma_L$), se establece $j \leftarrow select_0(B,j)$, y si el símbolo s se encuentra en el subárbol derecho (es decir, $s \in \Sigma_R$), se establece $j \leftarrow select_1(B,j)$. Esto se repite de forma ascendente hasta que se alcanza el nodo raíz donde se vuelve a calcular el valor j de la misma forma, pero en lugar de seguir con la llamada recursiva, se devuelva j como respuesta a la consulta $select_s(S,j)$.

2.6.4 Wavelet Tree comprimido con H_0

Si se utilizan los vectores de bits comprimidos con H_0 (sección 2.5) para los vectores de bits de los nodos internos del *Wavelet Tree* se puede conseguir una compresión de esta estructura cercana a la entropía empírica de orden cero de la secuencia S, haciendo que el espacio ocupado por el *Wavelet Tree* sea del orden de $nH_0(S) + o(n \log \sigma)$ [10, 51, 53]. Esta variante de los *Wavelet Trees* es práctica y muy competitiva en ocasiones, puesto que el rendimiento de sus consultas depende principalmente del rendimiento de las consultas sobre los vectores de bits comprimidos con H_0.

Capítulo 3

Fundamentos tecnológicos

En este capítulo se hablará de los fundamentos tecnológicos que se han utilizado para la realización de este proyecto, como pueden ser los conceptos más destacables sobre el lenguaje de programación Rust o el estado del arte de las librerías de estructuras de datos compactas existentes en la actualidad.

3.1 Estado del arte

En esta sección se hablará del estado del arte y de las distintas iniciativas existentes de librerías de estructuras de datos compactas en distintos lenguajes de programación, así mismo de cuáles son sus puntos fuertes y débiles.

3.1.1 Análisis de soluciones existentes

En lo referente al estado del arte, se han analizado las distintas soluciones existentes en la actualidad que ofrecen implementaciones de estructuras de datos compactas, las cuales son:

- ***Pizza&Chili*** [54, 55]: Es una de las primera iniciativas de este tipo, datando sus inicios en el año 2007. Consiste en una colección de corpus de texto e índices de texto comprimidos que surge debido a la idea de que los índices comprimidos serán una herramienta clave para el diseño de soluciones software sofisticadas y eficientes, pero la tecnología algorítmica necesaria para implementar dichos índices no se encuentra al nivel de un graduado universitario. Debido a que la implementación de cualquier índice comprimido es un trabajo que requiere un gran esfuerzo de ingeniería y un fuerte trasfondo en algoritmia, surge la necesidad de ofrecer una colección de índices comprimidos que puedan ser utilizados de forma sencilla. El corpus *Pizza&Chili* provee de un conjunto de implementaciones de índices comprimidos, además de una colección de textos sobre los que se pueden realizar pruebas y experimentos. Pese a que la colección de textos sigue siendo muy útil para realizar experimentos en la actualidad, el código disponible es

complicado de integrar en proyectos modernos, puesto que no se provee de una librería ni de un sistema de versiones, únicamente se proporcionan los archivos fuente individuales. Además, las implementaciones de estructuras de datos se limitan únicamente a índices de texto comprimidos.

- **libcds** [56]: Es una librería para C++ que provee implementaciones de bajo nivel de estructuras de datos compactas, como pueden ser los vectores de bits, vectores de bit comprimidos con H_0 [46] o los *wavelet trees* [47]. No es una librería muy conocida y carece de pruebas y de buena documentación, además que la instalación e integración con otros proyectos es complicada. Por último, esta librería no funciona con sistemas de 64 bits (puesto que en sus inicios, la librería estaba pensada para sistemas de 32 bits) y su desarrollo está descontinuado desde el año 2022.
- **libcds2** [57]: Es la recodificación y segunda versión de la librería libcds, la cual tiene como objetivos principales el soporte de sistemas de 64 bits (principal problema de la librería libcds), mejora de velocidad de las estructuras de datos compactas ya disponibles en la anterior versión, además de incorporar pruebas unitarias. Esta librería, al igual que su versión anterior, no es muy conocida y su instalación e integración con otros proyectos es complicada. Su desarrollo también está descontinuado desde el año 2022, por lo que se considera que es ya un proyecto abandonado.
- **Sux** [58, 59]: Este proyecto provee de implementaciones de varias estructuras de datos compactas, en las que podemos destacar estructuras que permiten realizar las operaciones de $rank$ y $select$ sobre vectores de bits. Este proyecto proporciona librerías para los lenguajes C++ y Java, altamente probadas y con *benchmarks*, además de una documentación decente, pero carece de instrucciones de instalación. Como punto negativo, no ofrece una implementación de vectores de bits que se puedan utilizar de una forma cómoda, sino que queda a cargo del usuario proporcionar el array de enteros que se interpretará internamente como un vector de bits, dificultando en gran medida el uso de esta librería. Actualmente en 2023, este proyecto está siendo bastante activo, e incluso está intentando implementar una versión de la librería en el lenguaje Rust [60], pero el desarrollo de esta última está todavía en una fase muy temprana y carece de funcionalidades importantes, por lo que no es usable.
- **SDSL** [29, 61]: La Succinct Data Structures Library (SDSL) es, sin lugar a duda, la librería de estructuras de datos compactas más conocida, potente y utilizada en la actualidad. Esta librería está escrita en C++ y provee de implementaciones de tanto las estructuras de datos más fundamentales (vectores de bits, vectores de enteros compactos, estructuras de $rank$ y $select$, *wavelet trees*, etc.) como de estructuras de datos más complejas

como pueden ser los *Compressed Suffix Arrays* [9] o los *Compressed Suffix Trees* [62]. Esta librería contiene las aportaciones más destacables de más de 40 artículos científicos, proporcionando implementaciones de código libre de gran calidad y muy eficientes tanto en espacio utilizado como en tiempo de ejecución, además de estar ampliamente probadas para multitud de parámetros de las estructuras. A su vez, esta librería presenta muchos y muy buenos ejemplos de uso junto con tutoriales guiados, pero carece de buena documentación del código y de las clases y métodos que se utilizan. También es muy notable que permite utilizar la memoria secundaria cuando una estructura de datos puede no caber en memoria principal, además de funcionar correctamente en sistemas de 64 bits. Por último, es destacable que esta librería tiene funcionalidades muy útiles e importantes: las estructuras de datos implementadas son altamente configurables, mediante el uso de *templates* del lenguaje C++ para establecer sus parámetros (una limitación de este sistema de configuración es que únicamente se puede configurar la estructura de datos en tiempo de compilación, no en tiempo de ejecución); se puede visualizar el espacio ocupado por una estructura de datos de forma interactiva, descomponiéndolo en el espacio ocupado por cada una de las partes que la componen para así poder analizarlo de forma muy sencilla; todas las estructuras de datos disponibles son serializables y deserializables, lo que permite guardarlas en un fichero de disco y volver a construirlas en memoria posteriormente, a partir de dicho fichero; y por último, esta librería es fácil de utilizar e intuitiva, puesto que proporciona muchas de estas funcionalidades de una forma de muy alto nivel.

- **sucds** [63]: Es uno de los pocos intentos actuales de implementar una librería de estructuras de datos compactas en el lenguaje Rust. La librería provee de las estructuras fundamentales más necesarias, como pueden ser los vectores compactos de enteros de tamaño fijo, vectores de bits, estructuras auxiliares que permiten realizar las operaciones de $rank$ y $select$ de forma eficiente sobre estos últimos, o los *wavelet trees*. Como principal punto negativo, las estructuras de datos que provee esta librería no son configurables, es decir, no permiten establecer parámetros para obtener un balance entre el espacio ocupado y el tiempo de ejecución, lo que hace que no sea una librería muy flexible. Esta librería presenta una buena documentación y también pruebas unitarias, aunque estas últimas de forma no muy exhaustiva. Por último, mencionar que esta solución es fácilmente integrable en otros proyectos en Rust, utilizando el gestor de paquetes Cargo, puesto que se encuentra disponible en el repositorio oficial de paquetes de Rust [64] y la instalación es tan sencilla como añadir una línea al fichero de dependencias del proyecto en el que será utilizada.

- **rust-bio** [65, 66]: Esta es una librería escrita en Rust muy utilizada y conocida en el ám-

bito de la bioinformática en este lenguaje, que proporciona implementaciones de algoritmos y estructuras de datos muy útiles en el campo de la bioinformática. En concreto, provee de algoritmos muy complejos como puede ser el alineamiento de secuencias de ADN, la búsqueda de patrones en texto, la Burrows-Wheeler Transform (BWT), el FM-index, los *suffix arrays*, etc. Pese a todo esto, carece de implementaciones de calidad de las estructuras de datos compactas más fundamentales, puesto que únicamente proporciona una sola implementación de estructura auxiliar de $rank$ y $select$ sobre vectores de bits, que además no es configurable y presenta graves problemas del uso del espacio, dado que utiliza la representación de $rank$ con un nivel de directorio de forma muy ineficiente, llegando a duplicar el espacio utilizado. Esta librería presenta una buena documentación y extensas pruebas unitarias, además de ser fácilmente integrable en otros proyectos en Rust, lo que justifica su amplio uso. El gran problema de esta solución es lo mencionado anteriormente, puesto que debido a las débiles bases sobre las que se construyen los algoritmos complejos que proporciona, estos tendrán un rendimiento en espacio y tiempo muy por debajo de lo que podrían tener si se utilizasen estructuras de datos compactas de calidad.

3.1.2 Conclusión de las soluciones analizadas

Tras analizar las distintas soluciones existentes, se puede concluir que la librería SDSL es la más completa y la principal candidata a utilizar en este campo. La principal limitación de esta es que está escrita en el lenguaje C++, que pese a ser un lenguaje muy establecido y potente, no es un lenguaje moderno que se adecúe a las necesidades de desarrollo de software actuales como sí lo es el lenguaje Rust. Por ejemplo, C++ no dispone de un gestor de paquetes y de proyecto de forma oficial, ni de un repositorio de paquetes oficial, lo que hace que la integración de otras librerías en proyectos que utilicen C++ sea una tarea muy complicada y que dificulta en gran medida los sistemas de construcción de código.

El lenguaje Rust, en cambio, se adecúa mucho mejor a las necesidades de desarrollo de software actuales, puesto que su gestor de paquetes y proyecto oficial, Cargo [67], junto con el repositorio oficial de paquetes, *crates.io* [64], hace que integrar una librería sea tan sencillo como añadir una línea al fichero de dependencias del proyecto. Como ya hemos visto en la sección anterior, las librerías de estructuras de datos compactas existentes en Rust están en una fase muy poco madura y no proporcionan implementaciones eficientes y de calidad como sí lo hace la librería SDSL en C++, por lo que se considera que es necesario implementar una librería de estructuras de datos compactas en Rust que proporcione estructuras eficientes, altamente probadas, documentadas y configurables, de manera que se potencie el uso de estructuras de datos compactas en proyectos de software modernos escritos en este lenguaje.

Además, cabe destacar la gran importancia de proveer de unas implementaciones amplias

y de gran calidad de las estructuras de datos compactas más fundamentales, que sirvan para sentar las bases sobre las que construir estructuras de datos compactas más complejas. Este último es el camino que ha seguido la librería SDSL, frente a la ineficiente implementación de estas estructuras que se puede encontrar en la librería *rust-bio*, que hace que los esfuerzos empleados en implementar algoritmos complejos se vean mermados por el uso de estructuras de datos fundamentales de baja calidad.

3.2 Lenguaje de programación Rust

Rust [11, 68] es un lenguaje de programación moderno y seguro que se destaca por su énfasis en la prevención de errores de memoria y su alto rendimiento. Este lenguaje tiene una multitud de conceptos y funcionalidades cuya comprensión es necesaria para poder desarrollar software en él. Sin embargo, en esta sección nos centraremos únicamente en los conceptos indispensables para poder entender la memoria del proyecto realizado.

En Rust, los *structs* son estructuras de datos que se asemejan a las clases en los lenguajes de programación orientados a objetos (como pueden ser Java o Python) y se utilizan para poder definir tipos de datos compuestos, a la vez que se permite definir métodos que operen sobre estos tipos de datos.

Los *traits*, por otro lado, son similares a las interfaces en otros lenguajes de programación y permiten definir comportamientos comunes que pueden ser implementados por distintos tipos de datos como pueden ser los *structs*, pudiendo agrupar así tipos de datos diferentes que compartan una cierta funcionalidad.

Uno de los conceptos más distintivos de Rust es su sistema de *ownership* (propiedad). En Rust, cada valor tiene un propietario único y las reglas de *ownership* [11] aseguran que no haya fugas de memoria ni problemas de acceso concurrente. Esto hace que Rust sea altamente seguro en términos de manejo de la memoria y se pueda lograr una gestión automática de memoria por parte del compilador, liberando de esta manera la memoria reservada cuando se deje de utilizar, sin necesidad de disponer de un recolector de basura.

Sin embargo, en situaciones excepcionales (por ejemplo, para realizar optimizaciones y obtener un mejor rendimiento) en las que sea necesario desactivar algunas de las restricciones de seguridad del sistema de *ownership*, se puede utilizar la funcionalidad *unsafe* de este lenguaje. Esto permite al programador eludir las comprobaciones de seguridad del compilador, pero conlleva la responsabilidad de garantizar por parte del desarrollador que el código sea completamente seguro y libre de errores, puesto que de otra manera se podría producir un comportamiento indefinido (UB) en tiempo de ejecución.

Capítulo 4

Metodología y Planificación

En este capítulo se explicará la metodología de desarrollo que se ha seguido en el proyecto, así como la planificación y seguimiento del mismo.

4.1 Metodología de desarrollo

Como metodología de desarrollo, se ha seguido una metodología ágil, iterativa e incremental, de forma que se favorezca la flexibilidad y la rápida adaptación al ritmo de trabajo además de obtener resultados intermedios de forma rápida. El desarrollo del proyecto se dividió en *sprints* de dos o tres semanas de duración, en los que se realizarían las tareas planificadas para dicho período y, al final del *sprint*, se tendría una reunión para revisar el trabajo realizado y planificar el siguiente *sprint*.

Además, es importante mencionar que durante el desarrollo se empleó en gran medida la metodología *Test-Driven Development (TDD)* [69, 70], la cual es una práctica de desarrollo de software que se basa en escribir las pruebas unitarias antes de escribir el código de la funcionalidad a implementar, de forma que la implementación esté guiada por las pruebas y se asegure que el código pueda ser automáticamente probado y que cumpla con los requisitos funcionales. Mediante el uso de esta metodología se favorece en gran medida la calidad del software, puesto que todas las funcionalidades tendrán sus pruebas unitarias asociadas y se facilita la refactorización del código. De esta manera, dado un código inicial que pase correctamente las pruebas, podemos modificarlo sustancialmente por limpieza u optimizaciones y asegurarnos de que sigue pasando las pruebas correctamente. Empleando esta metodología se consigue un desarrollo en el que el programador se siente más seguro de que el código que está escribiendo es correcto y sin necesidad de comprobar de forma manual que todos los requisitos funcionales se siguen verificando.

La primera parte del proyecto se dedicó a realizar un estudio preliminar de los conceptos previos al desarrollo, como puede ser el estudio de los fundamentos teóricos de las estructuras

de datos compactas para poder escoger posteriormente las estructuras a implementar, o el estudio del lenguaje de programación utilizado.

En lo referente a los siguientes *sprint*, las tareas realizadas fueron, principalmente, para cada una de las estructuras escogidas a implementar:

- **Análisis**: Análisis de los requisitos funcionales y no funcionales que se deben cumplir en la implementación de la estructura de datos.
- **Diseño**: Diseño de los diferentes módulos que compondrían la estructura de datos escogida en este *sprint*, de forma que se pudiese realizar una implementación modular y reutilizable, utilizando lo diseñado en anteriores *sprint* y facilitando el uso en *sprints* posteriores.
- **Implementación**: Implementación de la estructura de datos en el lenguaje Rust, siguiendo el diseño realizado en el paso anterior y buscando cumplir con los requisitos funcionales y no funcionales descritos en el análisis.
- **Prueba**: Programación de pruebas unitarias sobre la implementación desarrollada en el paso anterior, de forma que sirvan para verificar que se cumplen los requisitos funcionales descritos en el análisis.
- **Documentación**: Documentación de lo implementado utilizando estándares de documentación de código.

4.1.1 Herramientas utilizadas

A lo largo del desarrollo proyecto, se utilizaron las siguientes herramientas como apoyo:

- **Latex** [71]: Es una herramienta de composición de textos, orientada a la creación de documentos científicos y técnicos. Se utilizó para la redacción de la memoria del proyecto.
- **Cargo** [67]: Es el gestor de paquetes y de proyectos de Rust. Se utilizó para la gestión de dependencias y compilación del proyecto.
- **Git** [72]: Es un sistema de control de versiones distribuido, que permite el trabajo colaborativo en un proyecto. Se utilizó para el control de versiones del proyecto.
- **GitHub** [73]: Es una plataforma de desarrollo colaborativo, que permite el alojamiento de proyectos utilizando el sistema de control de versiones *Git*. Se utilizó para la distribución y alojamiento del proyecto.

- **Visual Studio Code** [74]: Es un entorno integrado de desarrollo, que permite la integración con herramientas externas mediante extensiones. En el proyecto se utilizó como entorno de desarrollo para la implementación de la librería.
- **Draw.io** [75]: Es una herramienta en línea para la creación de diagramas. Se utilizó para la creación de los diagramas de diseño.

4.2 Planificación

La planificación del proyecto se dividió en un total de siete *sprint*, que son los siguientes:

- **Sprint 1**: Estudio de conceptos teóricos previos sobre estructuras de datos compactas, junto con una recopilación de la literatura más destacable. Además, en este *sprint* se realizó una selección inicial de las estructuras destacadas como más importantes en una implementación inicial.
- **Sprint 2**: Aprendizaje avanzado del lenguaje de programación Rust, junto con un análisis de su viabilidad para la implementación de las estructuras de datos compactas. Además, preparación del entorno de trabajo junto con las herramientas a utilizar e investigación sobre el ecosistema de Rust y las librerías de estructuras de datos compactas disponibles tanto en Rust como en otros lenguajes de programación.
- **Sprint 3**: Análisis, diseño, implementación, prueba y documentación de los vectores bits planos (sección 2.4) y vectores de enteros compactos de tamaño fijo y variable (secciones 2.3.1 y 2.3.2), además de la definición de los elementos básicos y reutilizables de la librería que servirán como base para lo implementado posteriormente.
- **Sprint 4**: Realización de las mismas tareas que en el *sprint* anterior, pero en este caso para dar soporte a las estructuras de $rank$ en un nivel y en dos niveles de directorio (sección 2.4.1) sobre vectores de bits planos, junto con las operaciones de $rank$ y $select$ asociadas a estas estructuras.
- **Sprint 5**: De la misma forma, realización del análisis, diseño, implementación, prueba y documentación para el vector de bits comprimido con H_0 (sección 2.5), además de las operaciones de $rank$, $select$ y $access$ asociadas a esta estructura.
- **Sprint 6**: Una vez más, de igual manera que en los *sprint* anteriores, se realizó ;a implementación del *Wavelet Tree* (sección 2.6), junto con las operaciones de $rank$, $select$ y $access$ a nivel de símbolos sobre esta estructura.

- **Sprint 7**: Experimentación y análisis de los resultados obtenidos en los *benchmarks* de las estructuras de datos implementadas, realizando una comparativa con las soluciones implementadas en otras librerías.

Finalmente, después de la realización de los *sprint* anteriores, se realizó la tarea final de la redacción de la memoria del proyecto.

Se puede ver en la tabla 4.1 la planificación del proyecto, donde se muestran las fechas de inicio y fin estimadas de cada *sprint*. Posteriormente, podremos ver en la tabla 4.3 las fechas reales de inicio y fin correspondientes.

Tarea	Fecha de inicio estimada	Fecha de fin estimada
Sprint 1	01/03/2023	15/03/2023
Sprint 2	16/03/2023	29/03/2023
Sprint 3	30/03/2023	12/04/2023
Sprint 4	13/04/2023	26/04/2023
Sprint 5	27/04/2023	10/05/2023
Sprint 6	11/05/2023	24/05/2023
Sprint 7	25/05/2023	07/06/2023
Memoria	08/06/2023	28/07/2023

Tabla 4.1: Planificación estimada de las tareas del proyecto

4.2.1 Recursos

En esta sección se describen los recursos necesarios para la realización del proyecto, los cuales se dividen en recursos materiales y recursos humanos.

Recursos Materiales

El único recurso material que fue necesario para la realización del proyecto fue un ordenador portátil personal con las siguientes características:

- **Procesador**: AMD Ryzen 5 3500U
- **Memoria RAM**: 2x4GB @ 2667MHz DDR4
- **Almacenamiento**: 256GB NVMe SSD
- **Sistema Operativo**: Arch Linux x86_64 (Kernel 6.4.12)

Recursos Humanos

- **Analista programador**: Este es el rol asumido por el autor del proyecto, el cual estaba a cargo del diseño e implementación de la librería, además de la realización de los experimentos y la redacción de la memoria del proyecto.
- ***Project Managers***: Los tutores del proyecto asumieron el rol de dirigir, gestionar y revisar el proyecto, ya que fueron los que se encargaron de la planificación y supervisión del proyecto y de los *sprint*. Para favorecer la comunicación entre el autor y los tutores, se realizaron reuniones al finalizar cada *sprint*, de forma que se revisaba el trabajo realizado y se planificaba el siguiente *sprint*.

4.2.2 Costes

En lo relacionado a la estimación de los costes del proyecto, tendremos en cuenta únicamente los costes de los recursos humanos. Para calcular dichos costes, primero calcularemos las horas por persona que se dedicaron al proyecto, y a continuación, multiplicaremos dichas horas por el coste por hora de cada persona. El autor del proyecto dedicó **4 horas de trabajo al día** por cada día laborable de la semana, lo que equivale a **20 horas de trabajo semanales**, y teniendo en cuenta que se estimó una duración de **21 semanas** para el proyecto, se obtiene un total de **420 horas** de trabajo estimadas. Por otro lado, los tutores del proyecto dedicaron aproximadamente unas **30 horas** de trabajo entre las reuniones y la revisión de la memoria.

Para calcular el coste por hora de cada persona, se han utilizados datos de distintas fuentes, como pueden ser las páginas Levels [76] o Indeed [77], de las cuales se han extraído los salarios medianos de los puestos de Analista Programador Junior y *Project Manager*, dando lugar a un salario mediano bruto anual de **25.479 €** para el puesto de Analista Programador Junior y de **49.568 €** para el puesto de *Project Manager*. A partir de estos datos, tomando en cuenta que aproximadamente el número de horas laborables anuales es de **1.800 horas**, se obtiene un coste por hora bruto de **14,15 €/h** para el puesto de Analista Programador Junior y de **27,54 €/h** para el puesto de *Project Manager*.

Finalmente, podemos ver estos datos reflejados en la tabla 4.2, donde se muestran los costes por hora brutos de cada persona, junto con el número de horas que dedicaron al proyecto de forma estimada y, finalmente, el coste total bruto (antes de impuestos) estimado para cada persona. Posteriormente, podremos ver los costes reales del proyecto en la tabla 4.4. El autor, Jorge Hermo, fue quien asumió el rol de Analista Programador Junior, mientras que los tutores del proyecto, Diego Seco y Susana Ladra, asumieron el rol de *Project Manager*.

Persona	Coste/h	Horas	Coste total
Jorge Hermo	14,15 €/h	420	5.943,00 €
Diego Seco	27,54 €/h	30	826,20 €
Susana Ladra	27,54 €/h	30	826,20 €
Total			7.595,40 €

Tabla 4.2: Costes del proyecto

4.3 Resultado del seguimiento

En lo referente a la monitorización y seguimiento del proyecto, se ha cumplido en gran parte con la planificación inicial, gracias a que no han existido modificaciones sustanciales en los requisitos y la mayor parte de las tareas se estimaron correctamente.

A pesar de esto, sí que se han producido algunos retrasos en la realización de algunas tareas. En concreto, el **Sprint 1** se retrasó una semana, debido a que se subestimó el tiempo necesario para investigar y aprender sobre las estructuras de datos compactas, las cuales acabaron siendo mucho más complejas de lo esperado. Por otro lado, el **Sprint 2** se retrasó también una semana, puesto que el aprendizaje avanzado del lenguaje Rust llevó más tiempo del esperado. Por último, el **Sprint 3** se retrasó una semana más, debido a que el ecosistema de Rust no estaba lo suficientemente maduro en esta temática.

Teniendo todo esto en cuenta, se puede ver en la tabla 4.3 las fechas de inicio y fin reales de cada tarea. Además, se muestra en la tabla 4.4 el coste real del proyecto, teniendo en cuenta los retrasos mencionados anteriormente.

Tarea	Fecha de inicio real	Fecha de fin real
Sprint 1	01/03/2023	22/03/2023
Sprint 2	23/03/2023	12/04/2023
Sprint 3	13/04/2023	03/05/2023
Sprint 4	04/05/2023	17/05/2023
Sprint 5	18/05/2023	31/05/2023
Sprint 6	01/06/2023	14/06/2023
Sprint 7	15/06/2023	28/06/2023
Memoria	29/06/2023	17/08/2023

Tabla 4.3: Fechas reales de las tareas del proyecto

Persona	**Coste/h**	**Horas**	**Coste total**
Jorge Hermo	14,15 €/h	480	6.792,00 €
Diego Seco	27,54 €/h	30	826,20 €
Susana Ladra	27,54 €/h	30	826,20 €
Total			8.444,40 €

Tabla 4.4: Costes reales del proyecto

Capítulo 5

Análisis

En este capítulo se expondrá el análisis realizado de la literatura para la selección de las estructuras de datos compactas que se implementarán, así como los requisitos funcionales y no funcionales que esta tiene que cumplir.

5.1 Selección de las estructuras de datos

El primer paso a realizar es el de hacer una selección de las estructuras de datos consideradas como las más fundamentales. Esta selección se realizó basándose en múltiples artículos científicos de la literatura y en estudios sobre las estructuras de datos compactas que son útiles en la práctica [1]. La metodología ágil que se utilizó en este proyecto, descrita en la sección 4.1, nos permite adaptarnos a los cambios que se puedan producir en el desarrollo, puesto que quizá una estructura, que en un principio no se consideraba necesaria o importante, al final se puede considerar que sí lo es y se puede implementar sin afectar de forma muy negativa a la duración del proyecto.

Las estructuras de datos que se destacaron de cara a ser implementadas son principalmente las que se explicaron en la sección 2.2, y son las siguientes:

- Vector compacto de enteros, tanto de tamaño fijo como de tamaño variable (sección 2.3).
- Vector de bits en su representación plana, así como las estructuras de soporte de $rank$ como puede ser el $rank$ con un nivel o con dos niveles de directorio (sección 2.4).
- Vector de bits comprimido con H_0 (sección 2.5)
- *Wavelet tree*, tanto en su forma clásica como su variante comprimida con H_0 (sección 2.6).

5.2 Análisis de requisitos

En esta sección se detallarán los requisitos funcionales y no funcionales que debe cumplir la librería que se va a desarrollar.

Para la definición de estos requisitos, se han tomado en cuenta varios de los aspectos positivos de las librerías de estructuras de datos compactas existentes en la actualidad, expuestas en la sección 3.1.1, además de que se han intentado mejorar los aspectos negativos y las desventajas que presentan estas librerías.

5.2.1 Requisitos funcionales

Los requisitos funcionales que debe cumplir la librería son los siguientes:

- **RF1**: La librería debe permitir la creación de vectores de enteros de tamaño fijo y de tamaño variable, junto con sus operaciones básicas.
- **RF2**: La librería debe posibilitar la creación de vectores de bits en su representación plana, junto con las estructuras de soporte de $rank$ en un nivel y en dos niveles de directorio, a la vez que las operaciones de $rank$ y $select$ con estas estructuras.
- **RF3**: Se debe proporcionar la creación de vectores de bits comprimidos con H_0, además de las operaciones de $rank$, $select$ y $access$ sobre esta estructura.
- **RF4**: La librería debe permitir la creación de *wavelet trees*, tanto su variante normal como la comprimida con H_0, junto con las operaciones de $rank$, $select$ y $access$ sobre alfabetos de símbolos arbitrarios.
- **RF5**: Se debe soportar la serialización y deserialización de las estructuras de datos en los formatos de intercambio de datos más utilizados en la actualidad, los cuales son el formato binario, JSON y Protobuf.
- **RF6**: Las estructuras de datos implementadas deben reportar información sobre el tamaño que ocupan en memoria.
- **RF7**: Las estructuras de datos deben ser altamente configurables y parametrizables, tanto parámetros sencillos como es el caso del $rank$ con un nivel de directorio, como parámetros más complejos como puede ser el tipo de vector de bits que se utiliza en el *wavelet tree*. Además, estos parámetros deben poder ser configurados en tiempo de ejecución y no únicamente en tiempo de compilación.

5.2.2 Requisitos no funcionales

Los requisitos no funcionales que se han definido son los siguientes:

- **RNF1**: Esta librería debe ser eficiente, tanto en tiempo de ejecución como en uso del espacio para cada una de las estructuras implementadas, por lo que dichas medidas deben estar en el mismo orden de magnitud que las de las implementaciones más potentes en la actualidad, que son las de la librería SDSL.
- **RNF2**: La librería debe ser fácil de utilizar y entender, además que deben proveerse de abstracciones que faciliten su uso.
- **RNF3**: Es necesario que la librería sea segura y que no contenga errores de memoria ni de ningún otro tipo.
- **RNF4**: Las estructuras de datos propuestas por esta librería deben estar altamente probadas, de forma que se demuestre su correcto funcionamiento.
- **RNF5**: La librería tiene que ser altamente extensible, de forma que se puedan añadir nuevas estructuras y funcionalidades en un futuro sin tener que modificar las bases sobre las que se ha construido la librería.
- **RNF6**: Es necesario que esta solución sea compatible y eficiente tanto en sistemas de 64 bits como de 32 bits (siendo ejemplos de estos últimos dispositivos *edge* como las *Raspberry Pi* y derivados, que se benefician ampliamente de estas estructuras de datos).
- **RNF7**: Se debe proveer de una buena documentación, tanto de las estructuras de datos como de las funcionalidades que se proporcionan, así como de ejemplos de uso de la librería.
- **RNF8**: La librería tiene que ser fácilmente utilizable en otros proyectos de Rust, de forma que se integre correctamente en su ecosistema y se adecúe al flujo de trabajo habitual en este lenguaje.
- **RNF9**: La serialización y deserialización debe poder ser extendible de forma sencilla a nuevos formatos de intercambio de datos.
- **RNF10**: La librería debe ser de código libre, utilizando una licencia de software libre apropiada, de forma que se pueda utilizar tanto en proyectos libres como privativos.
- **RNF11**: El código desarrollado debe ser limpio, entendible y fácil de mantener, tanto en su diseño, como en su organización y nomenclatura, siguiendo los estándares de calidad de código de la industria [78]. Estas directrices son muy necesarias para lograr una comunicación científica de buena calidad [79].

Capítulo 6

Diseño

En este capítulo se detallará el diseño de la librería, así como las decisiones que se tomaron durante el proceso de diseño, de forma que se cumplan los requisitos especificados.

Además, se han tenido en cuenta en todo momento los conceptos de ***Clean Code*** [78], de forma que esta librería presente un código limpio, fácil de entender y mantener, favoreciendo de esta manera el cumplimiento del requisito no funcional RNF11.

A continuación se mostrarán varios diagramas UML de clases y se utilizarán los términos de clases e interfaces, pero cabe tener en cuenta que su equivalente en el lenguaje de programación Rust son los *structs* y los *traits*, respectivamente.

6.1 Vector de bits

Se puede ver el diagrama UML de clases del vector de bits plano en la figura 6.1, en el cual se puede ver que esta estructura está representada por la clase ***BitVec***. Esta clase tiene el objetivo de cumplir con el requisito funcional RF2.

El diseño de esta estructura se realizó siguiendo los fundamentos teóricos de la sección 2.4, la cual estará compuesta por un vector de enteros de tipo *usize* (*Vec<usize>*) y una longitud. En Rust, el tipo *usize* es un entero cuyo tamaño depende de la arquitectura del sistema, siendo de 32 bits en sistemas de 32 bits y de 64 bits en sistemas de 64 bits. Esto hace que el vector de enteros sea más eficiente, ya que se utiliza el tamaño de entero más adecuado para cada sistema, pero además también tiene el objetivo de cumplir el requisito no funcional RNF6, de forma que esta estructura funcione en sistemas de 32 bits y de 64 bits. En general, una importante decisión de diseño que se ha tomado en el contexto global de la librería es la utilización del tipo de dato *usize* siempre que sea posible, de forma que se cumpla el requisito no funcional RNF6 correctamente en toda la librería.

En dicho diagrama se pueden ver también los métodos de los que dispone el vector de bits, los cuales conforman el API de esta estructura. En concreto, los métodos *push*, *push_bits*,

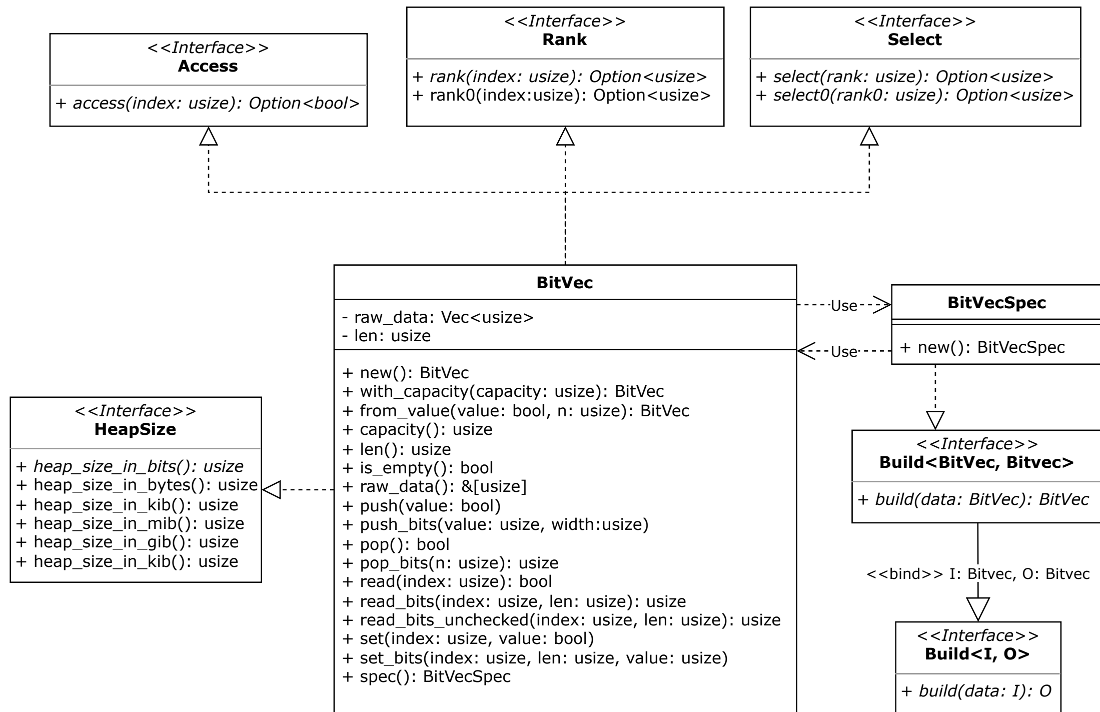

Figura 6.1: Diagrama de clases del vector de bits plano

read y *read_bits* serán los más interesante y los que se utilizarán más, de forma que se puedan añadir elementos al vector de bits y se puedan leer elementos del mismo.

En esta figura, a su vez, aparecen por primera vez las interfaces *Access*, *Rank* y *Select*, las cuales se corresponden con la definición de las operaciones correspondientes en la sección 2.4 y que veremos utilizadas en el resto de estructuras relacionadas con los vectores de bits. Cada una de estas interfaces provee de los métodos necesarios para realizar las operaciones de $access$, $rank$ y $select$, pero en concreto se destaca el uso del **patrón de diseño método plantilla** [80] en la interfaz $Rank$, puesto que clases que implementen dicha interfaz únicamente tendrán que implementar el método *rank* (el cual equivale a la operación $rank_1$) y el método *rank0* tendrá una implementación plantilla que utilizará el método *rank* para calcular su valor de retorno, puesto que se sabe que $rank_0(B,i) = i - rank_1(B,i)$. Como se puede ver en el diagrama, la clase *BitVec* implementa estas tres interfaces.

En lo referente a los demás elementos básicos de la librería, también se puede ver que la clase *BitVec* implementa la interfaz *HeapSize*, la cual tiene como objetivo cumplir con el requisito funcional RF6, de forma que se pueda inspeccionar el tamaño que ocupa la estructura en memoria. Esta interfaz utiliza de nuevo el **patrón método plantilla**, de forma que únicamente se tiene que implementar el método *heap_size_in_bits* y se soporten automáticamente otras unidades de medida como pueden ser los bytes, kibibytes, etc.

De la misma manera, podemos ver por primera vez a lo que llamaremos a partir de ahora como sistema de *specs*, el cual servirá para poder configurar las estructuras de datos en tiempo de ejecución, de forma que se cumpla el requisito funcional RF7. El **sistema de *specs*** es una de las aportaciones más potentes y destacables de esta librería, basándose principalmente en el uso de la interfaz *Build<I, O>*, la cual es una interfaz genérica que está parametrizada por los tipos *I*, que representa el tipo de dato que se utilizará como entrada, y *O*, que representa el tipo de dato que se producirá como salida. Como se puede intuir, esta interfaz permite que a partir de un método *build*, se pueda construir una estructura de datos tomando otra como entrada. Lo que especificará la forma (en configuración y parámetros) en la que se construirá el tipo de salida será lo que llamaremos *specs*, que se pueden interpretar como planos de construcción, los cuales contendrán todos los parámetros necesarios para configurar y crear la estructura de datos de salida deseada y además implementarán la interfaz *Build*. Este sistema se basa principalmente en el **patrón de diseño factoría abstracta** [80], de forma que se delegue en la interfaz *Build* y en los *specs* la responsabilidad de creación de los objetos, lo que favorecerá, además de la configuración en tiempo de ejecución, la extensibilidad de la librería (veremos un ejemplo de esto en la sección 6.5), cumpliendo el requisito no funcional RNF5.

En este caso, se puede ver el uso del *spec* ***BitVecSpec***, el cual implementa la interfaz *Build< BitVec, BitVec>*, de forma que se pueda construir un *BitVec* a partir de otro *BitVec*, y que además no permite ningún tipo de configuración. La implementación de este *spec* será trivial, ya que únicamente se tiene que devolver el *BitVec* que se recibe como entrada, pero esto servirá para la configuración de estructuras más complejas como puede ser el *wavelet tree* que se detallará en la sección 6.5.

Por último, mencionar que la clase *BitVec* proporciona en su API un método *read_bits _unchecked*, el cual utiliza la funcionalidad *unsafe* del lenguaje Rust para proveer de una forma de leer un rango de bits sin realizar comprobaciones de límites de los índices (también conocido como *bounds checking*). De esta forma, si estamos completamente seguros de que las posiciones que vamos a leer están dentro de los límites del vector, la lectura se hará de una forma mucho más rápida y eficiente. Proveer de un API *unsafe* en muchas de las estructuras de esta librería fue una decisión de diseño importante y que se tomó para poder exprimir al máximo el rendimiento de las mismas, de forma que se favorezca el cumplimiento del requisito no funcional RNF1.

6.2 Vector de enteros compacto

En esta sección veremos el diseño de las clases que implementan los vectores de enteros compactos, tanto de tamaño fijo como de tamaño variable. En concreto, se verán las clases *CompactIntVec* y *VariableSizeIntVec*, las cuales implementan el requisito funcional RF1.

6.2.1 Vector de enteros compacto de tamaño fijo

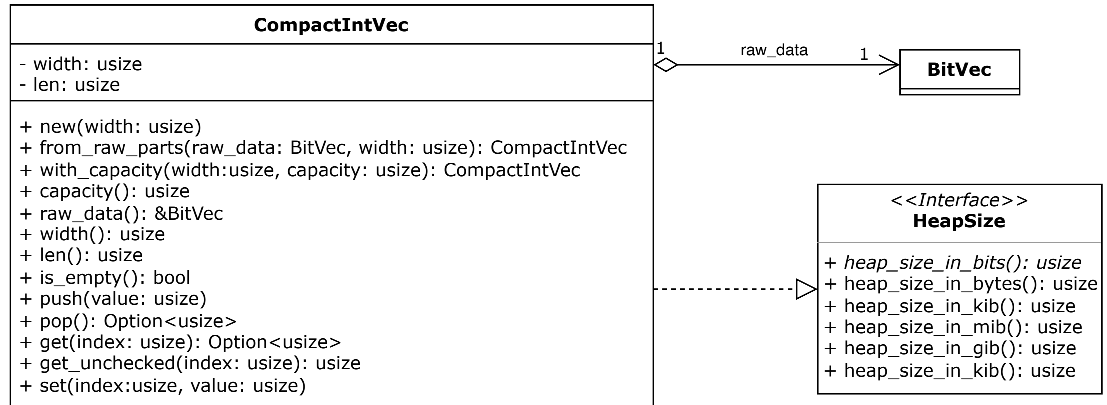

Figura 6.2: Diagrama de clases del vector compacto de enteros de tamaño fijo

El vector compacto de enteros de tamaño fijo se corresponde en la librería con la clase ***CompactIntVec***, y se puede ver el diagrama de clases correspondiente en la figura 6.2. En dicha figura podemos ver que el diseño, en relación a los elementos que componen esta estructura y las operaciones que permite, es muy similar a lo descrito en los aspectos teóricos de esta estructura en la sección 2.3.

En el diagrama se puede destacar la relación de composición con la clase *BitVec* a través del atributo *raw_data*, que es el lugar donde se guardarán los elementos del vector de enteros a nivel de bits, de una forma compacta, agrupando los bits del vector de bits de forma lógica en grupos de $width$ bits, para posteriormente ser leídos como enteros a través del método *read _int* de la clase *BitVec*. A su vez, podemos ver cómo se puede especificar el tamaño de los enteros en la creación de esta estructura a través de su método *new* (y que dicho tamaño no podrá verse modificado posteriormente) y que, además, se proveen de métodos $push$ y pop para poder modificar el número de elementos que conforman el vector de enteros, operaciones que no se contemplaban en los aspectos teóricos, pero que en la práctica son de suma importancia para poder utilizar esta estructura de forma cómoda. De igual forma que en el caso de la clase *BitVec*, la clase *CompactIntVec* también presenta un API *unsafe* para poder leer elementos del vector a través del método *get_unchecked*, de forma que no se realicen comprobaciones sobre el índice a leer y la lectura sea más rápida y eficiente, a expensas de la posibilidad de que ocurra comportamiento indefinido, o también conocido como Undefined Behavior (UB).

6.2.2 Vector de enteros compacto de tamaño variable

Se puede ver el diagrama de clases del vector de enteros compacto de tamaño variable en la figura 6.3. En este caso, esta estructura estará implementada en la clase ***VariableSizeIntVec***,

la cual presenta un diseño muy similar al de la clase *CompactIntVec*, pero con ciertas diferencias. En concreto, en el método *new* se especifica el parámetro k que determinará el ratio de muestreo de las posiciones de los elementos, a la vez que también se tendrá que proveer de una función que determine el tamaño de cada entero. Esta función, a partir de un índice de un elemento del vector, deberá devolver el tamaño del entero correspondiente. De esta forma se consigue modelar lo descrito en la sección 2.3.2 en la que se delega la lógica de cálculo del tamaño de cada elemento en una función, dotando al usuario de la librería de una gran flexibilidad, lo que favorece el cumplimiento del requisito no funcional RNF5. De la misma manera que en el caso de la estructura anterior, esta estructura también utiliza la clase *BitVec* como almacenamiento de los bits de sus elementos, de forma que dicho almacenamiento se realice de forma compacta.

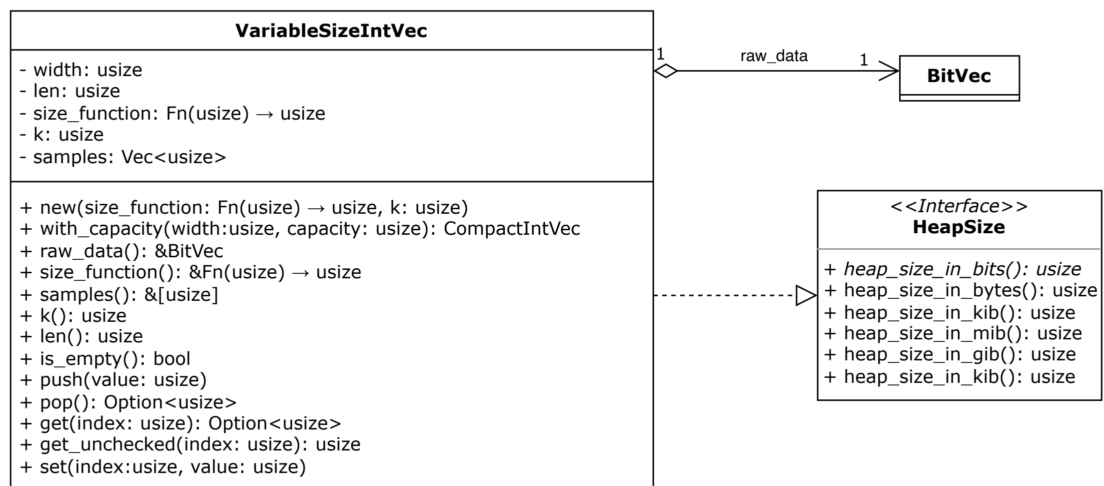

Figura 6.3: Diagrama de clases del vector compacto de enteros de tamaño variable

6.3 Estructuras auxiliares

Consideraremos como estructuras auxiliares a aquellas que se utilizan para dar soporte a operaciones como puede ser el $rank$, pero que por sí mismas no pueden dar la respuesta completa a las consultas y en última instancia tienen que acceder a los datos de la estructura a la que están dando soporte (o estructura principal). Un ejemplo de este tipo de estructuras pueden ser las estructuras de $rank$ en un nivel y dos niveles de directorio, vistas en la sección 2.4.1.

Entonces, estas estructuras deben tener alguna manera de poder acceder a la estructura a la que dan soporte. Esto se puede realizar de dos maneras; la primera de ellas es que la estructura de soporte guarde una referencia a la estructura principal, y la segunda es que en el momento de realizar la llamada a la función que resuelve la consulta (por ejemplo, para

el $rank$), siempre se le pase como parámetro una referencia a dicha estructura principal, de forma que no se tenga que guardar la referencia como atributo de la estructura de soporte.

El problema que existe con la primera opción anteriormente mencionada, es que la serialización y deserialización de la estructura de soporte se vuelve mucho más compleja y no se puede conseguir una solución elegante y que a la vez cumpla con la seguridad en memoria. En cambio, el uso de esta estructura es mucho más fácil y sin riesgo de cometer errores por parte del usuario de la librería. Por otro lado, la segunda opción funciona mucho mejor con la serialización y deserialización, haciendo que ambas funcionalidades se puedan implementar de una forma trivial, pero presenta problemáticas con la ergonomía de uso de dicha estructura, puesto que en cada llamada habría que pasar como referencia la estructura principal, pudiendo ocurrir errores por parte del usuario, como pasar una referencia a una estructura distinta a la que se utilizó para crear la estructura de soporte. En esta librería se ha optado por utilizar la segunda opción, pero intentando buscar una solución a la problemática de esta.

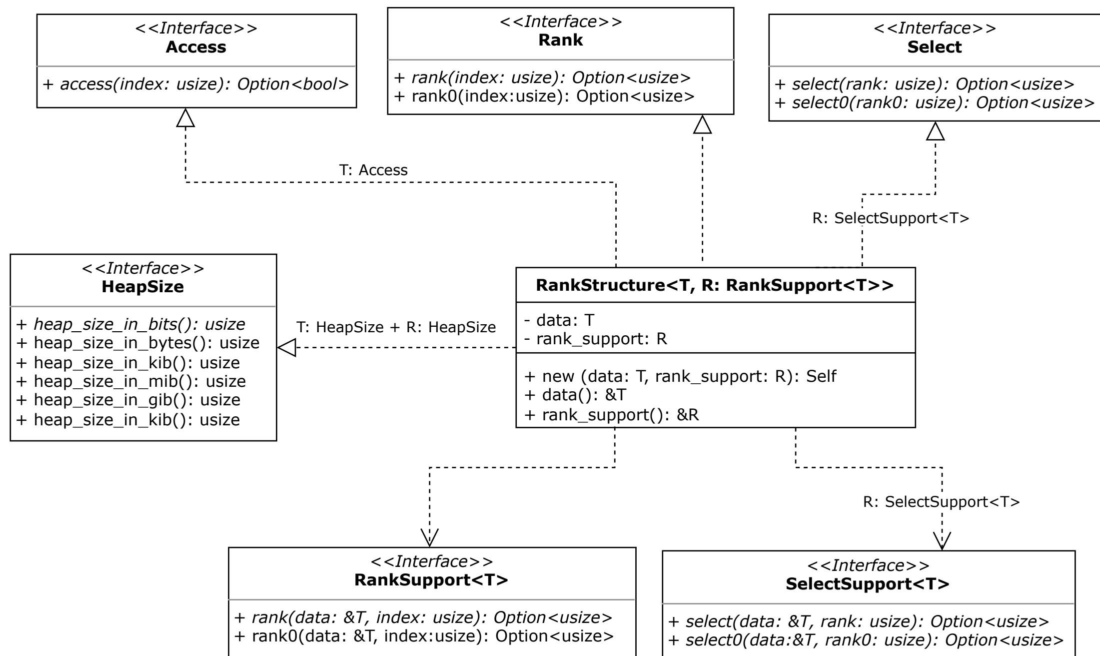

Figura 6.4: Diagrama de clases de los elementos genéricos para las estructuras auxiliares

Entonces, como solución a dicho problema, se ha utilizado la clase ***RankStructure***, de la que se puede ver su diagrama de clases en la figura 6.4. Esta clase sirve como un contenedor que agrupa en un mismo lugar la estructura de soporte y la estructura principal, de forma que esta clase sea la encargada de pasar las referencias de la estructura principal a la estructura de soporte en el momento de realizar alguna consulta en la que la clase de soporte necesite acceder a la estructura principal. La clase *RankStructure* utiliza el concepto de *ownership* de Rust, de forma que la estructura principal y la estructura de soporte se almacenen como atri-

butos de dicha clase, teniendo esta clase el *ownership* de ambas estructuras. De esta manera, la serialización de la clase *RankStructure* se hace de forma muy sencilla (puesto que contiene a las dos estructuras como valor y no como referencia), además de que se sigue manteniendo la facilidad de uso de cara al usuario, cumpliendo con los requisitos no funcionales RNF2 y RNF3.

En la figura 6.4 también se puede ver la inclusión de nuevas interfaces genéricas como son ***RankSupport<T>*** y ***SelectSupport<T>***, que modelan el concepto anteriormente expuesto sobre las estructuras de soporte, de forma que estas ayuden a realizar las operaciones de $rank$ y $select$ sobre una estructura de datos principal de tipo *T*. De esta forma, se diseñó la clase *RankStructure* de forma muy genérica y expansible, utilizando el potente sistema de tipos **genéricos** de Rust. Como se puede ver en dicho diagrama, la clase *RankStructure* está parametrizada por un tipo *T*, el cual representa a la estructura de datos principal, y un tipo *R*, el cual representa a la estructura de soporte, la cual tiene que implementar la interfaz *RankSupport<T>*. Es importante destacar que el diseño de la clase *RankStructure* está basado en gran parte en el **patrón de diseño adaptador** [80], de forma que se adapta una estructura de soporte para que se pueda interpretar, en última instancia, como si fuese una clase que implemente la interfaz *Rank*. Además, de forma que esta solución sea muy expandible, si dicha estructura de soporte también implementa la interfaz *SelectSupport<T>* (es decir, que permita realizar sobre ellas consultas de tipo $select$, como es el caso de las estructuras vistas en la sección 2.4.2), la estructura contenedora *RankStructure* también implementará la interfaz *Select*, de forma que también se soporte la operación de $select$. En el diagrama se puede ver que ocurre algo muy similar a lo anterior con las interfaces de $Access$ y $HeapSize$.

Utilizando esta abstracción, el usuario ya no tendrá por separado (en dos variables distintas) la estructura principal y una estructura de soporte que amplíe o mejore la eficiencia de sus funcionalidades, sino que tendrá una única estructura (por lo tanto, en una única variable del programa) que encapsulará a ambas. Esto facilita en gran medida el trabajo del desarrollador, puesto que evita que se cometan errores por confusión de variables y además produce un código mucho más limpio y expresivo. Esta funcionalidad surge con motivo de solucionar el problema de ergonomía de la librería SDSL en esto mismo, en la cual sí que se tiene en variables separadas la estructura principal y todas sus estructuras de soporte, de forma que la confusión de variables es muy fácil de cometer y el código resultante es mucho más verboso y difícil de entender.

Mediante el uso de esta potente abstracción, daremos soporte a las estructuras de $rank$ en un nivel y en dos niveles de directorio sobre vectores de bits (*BitVec*), pudiéndose ampliar fácilmente su uso en un futuro con nueva estructuras de soporte o nuevas estructuras principales. Todo lo expuesto sobre esta abstracción ayuda a cumplir con los requisitos no funcionales RNF2, RNF3 y RNF5.

6.3.1 Rank con un nivel de directorio

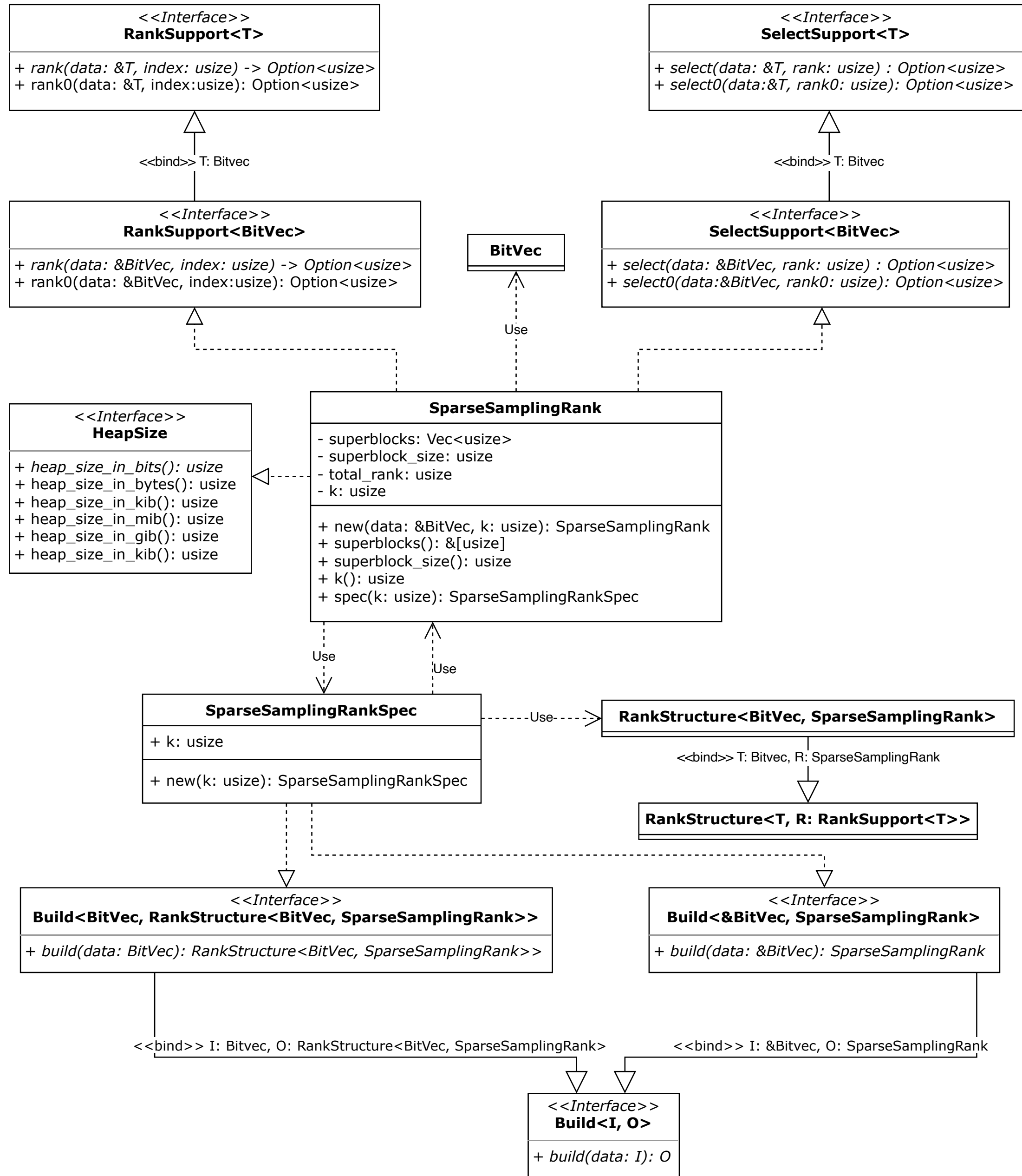

Figura 6.5: Diagrama de clases de la estructura de $rank$ con un nivel de directorio

Se puede ver el diagrama de clases de la estructura auxiliar de $rank$ con un nivel de directorio, descrita en la sección 2.4.1, en la figura 6.5. En este caso, esta estructura de soporte estará representada por la clase ***SparseSamplingRank***, y su diseño es muy similar a lo descrito en los aspectos teóricos de esta estructura, donde se destaca la utilización del tipo *Vec<usize>*

para almacenar el muestreo del $rank$ en el atributo *superblocks* de dicha clase, modelando así el array R que conforma esta estructura. Además, cabe mencionar que en el método *new* se puede especificar el parámetro *k* propio de esta estructura, que indica el ratio de muestreo.

Podemos observar que esta clase implementa la interfaz *RankSupport<BitVec>*, de forma que se pueda utilizar como estructura de soporte para el $rank$ sobre un vector de bits en su representación plana, y que además implementa la interfaz *SelectSupport<BitVec>*, pudiendo también ser utilizada como estructura de soporte para el $select$ sobre el mismo vector de bits. De esta manera, los aspectos teóricos de esta estructura se ven perfectamente reflejados en el diseño de la clase *SparseSamplingRank*, buscando cumplir con el requisito funcional RF2.

Por último, también es destacable que se vuelve a ver el uso del **sistema de *specs***, siendo la clase *SparseSamplingRankSpec* el *spec* correspondiente a esta estructura, la cual almacena el parámetro *k* necesario para la construcción de la clase *SparseSamplingRank*. Este *spec* implementa la interfaz *Build<&BitVec, SparseSamplingRank>*, de forma que se pueda, utilizando el *spec*, construir una estructura de tipo *SparseSamplingRank* a partir de una referencia a un vector de bits. Además, también implementa la interfaz *Build<BitVec, RankStructure<BitVec, SparseSamplingRank>>*, de manera que a partir de un vector de bits (en este caso, no como una referencia, sino como valor) se pueda construir una estructura de tipo *RankStructure* que encapsule los detalles de las estructuras auxiliares, como se ha mencionado en la sección anterior.

6.3.2 Rank con dos niveles de directorio

El diseño de la estructura de $rank$ con dos niveles de directorio, que fue descrita de forma teórica en la sección 2.4.1, se puede ver en la figura 6.6. La clase ***DenseSamplingRank*** es la encargada de representar a esta estructura de soporte, la cual presenta un diseño muy fiel a lo expuesto en sus aspectos teóricos. Esta estructura presenta grandes similitudes con la estructura explicada en la sección anterior, por lo que no se entrará mucho más en detalle. Sin embargo, es notable destacar la relación de composición con la clase *CompactIntVec*, de forma que se guardará el muestreo de $rank$ a nivel de bloque en dicho vector de enteros compacto de tamaño fijo, representando al array R' según lo expuesto en la explicación teórica de esta estructura.

6.4 Vector de bits comprimido con H_0

Se puede ver en la figura 6.7 el diagrama de clases de la estructura de vector de bits comprimido con H_0, representado por la clase ***RRRBitVec***. Esta estructura de datos es la encargada cumplir con el requisito funcional RF3 y su diseño está basado en gran parte, una vez más, en los aspectos teóricos de esta estructura, descritos en la sección 2.5. El diseño de esta estructura

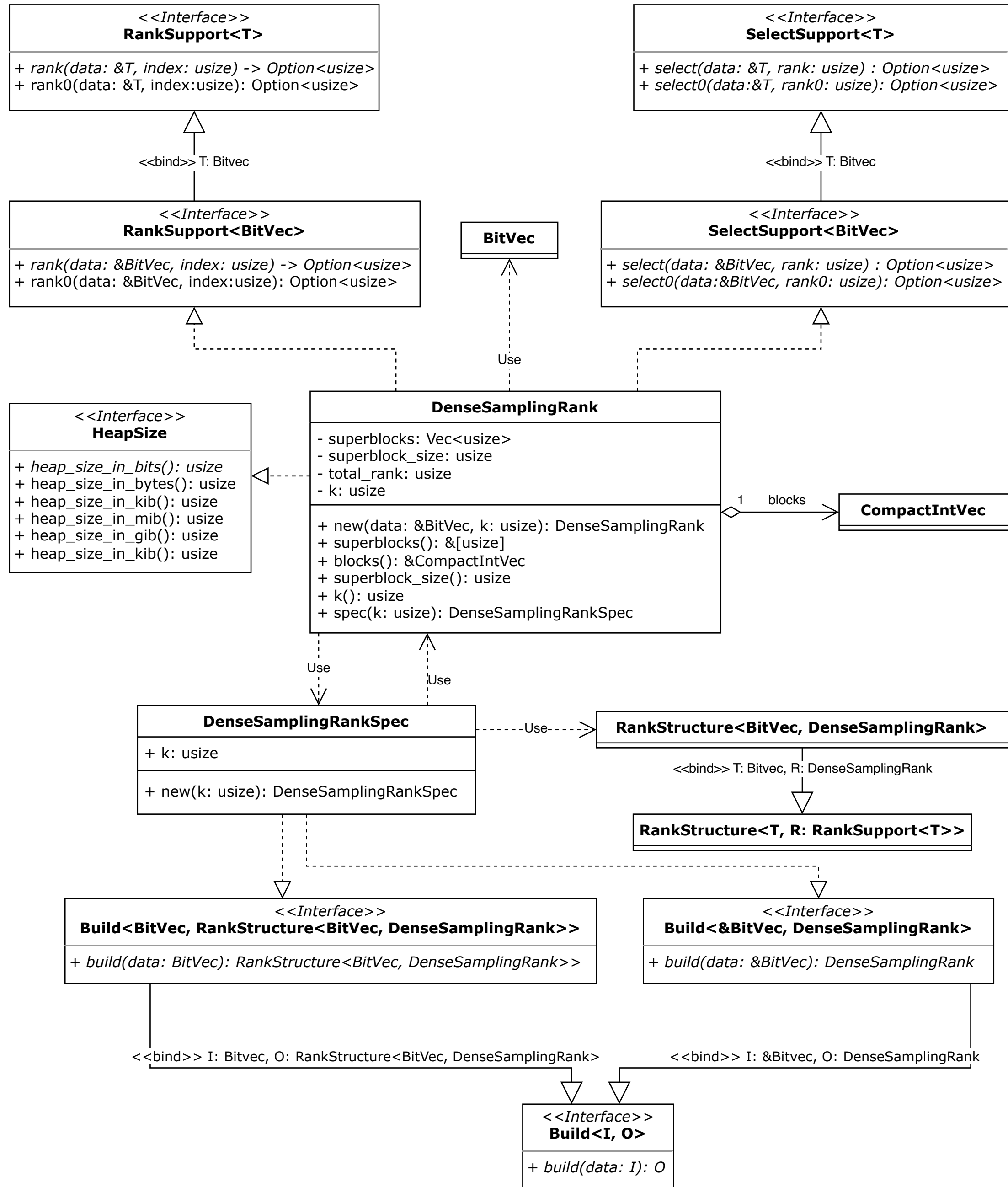

Figura 6.6: Diagrama de clases de la estructura de $rank$ con dos niveles de directorio

es muy similar al de la clase *BitVec*, puesto que esta representación comprimida también es autosuficiente para las consultas de $access$, $rank$ y $select$ y puede implementar las interfaces correspondientes de forma directa, sin necesidad de utilizar la solución empleada en la sección 6.3. Esta estructura permite, mediante su método *new*, crear una instancia de la misma a partir de un vector de bits plano (*BitVec*) recibido como valor (adquiriendo su *ownership*),

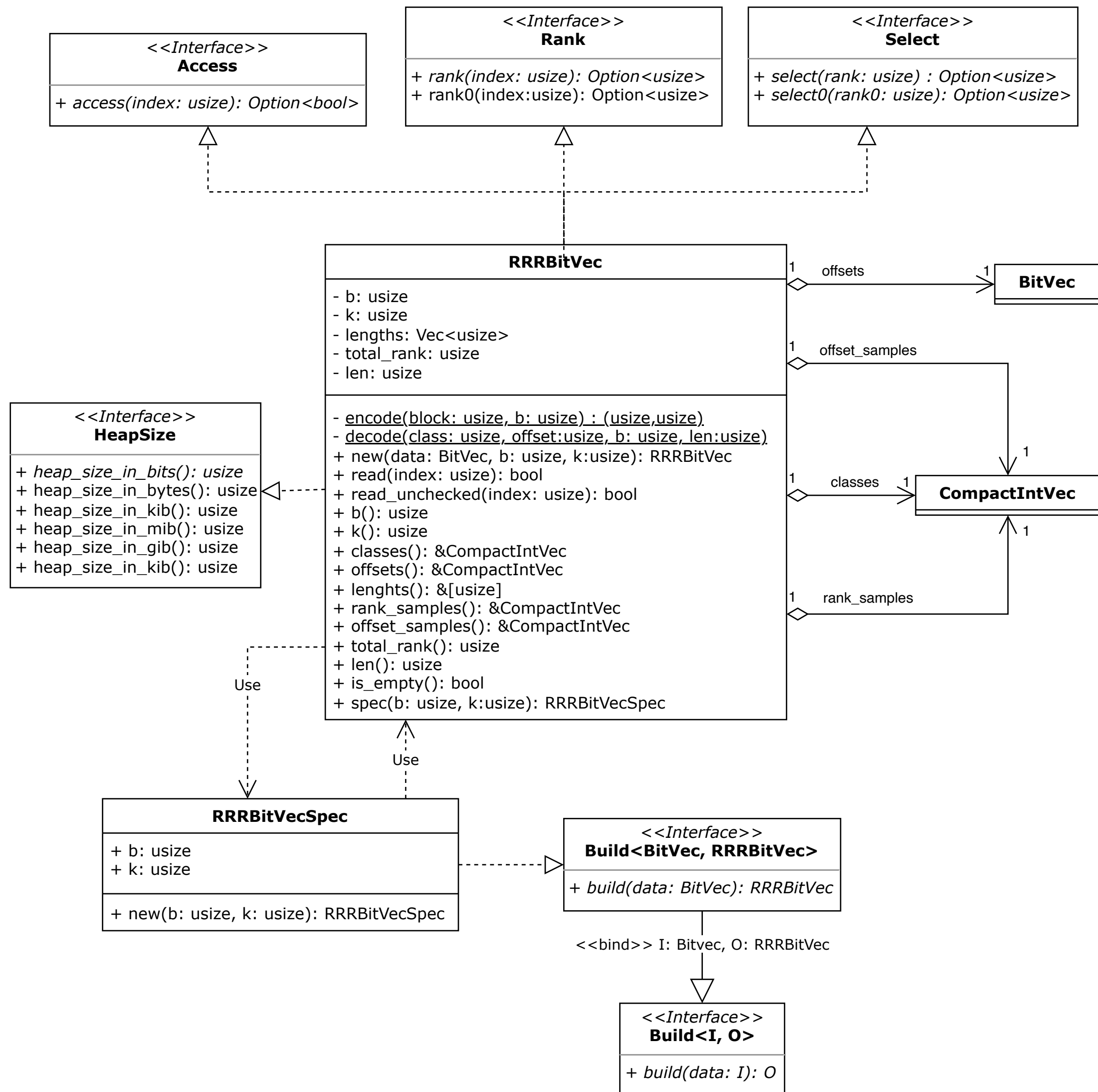

Figura 6.7: Diagrama de clases del vector de bits comprimido con H_0

además de especificar el tamaño de bloque b y el ratio de muestreo k, parámetros propios de esta estructura que fueron descritos en los aspectos teóricos.

A su vez, podemos ver la utilización del **sistema de *specs*** en la clase *RRRBitVecSpec*, de forma muy similar a lo visto para la clase *BitVecSpec*, pero en este caso se especifica el tamaño de bloque b y el ratio de muestreo k con el que se creará la estructura *RRRBitVec*. De esta manera, también se consigue que se pueda configurar esta estructura en tiempo de ejecución, cumpliendo con el requisito funcional RF7.

La principal diferencia a los aspectos teóricos que se puede destacar es que, en lugar de utilizar la clase *VariableSizeIntVec* para almacenar los desplazamientos, se utilizarán sus componentes internos por separado, es decir, se utilizará un vector de bits para almacenar los

desplazamientos (atributo *offsets*) y un array compacto de enteros de tamaño fijo para almacenar el muestreo de longitudes de los elementos (atributo *offset_samples*). La razón detrás de esto es que de esta manera se puede utilizar el mismo parámetro *k* para el muestreo de las longitudes (*offset_samples*) y para el muestreo del $rank$ (*rank_samples*), permitiendo de esta manera que se pueda realizar en un mismo bucle el cálculo del $rank$ y de las longitudes de los elementos que se encuentran entre dos muestras de k bloques consecutivas, favoreciendo así el rendimiento de la estructura, puesto que de otra manera se tendría que realizar esto último en dos bucles separados (puesto que la utilización de la clase *VariableSizeIntVec* no nos permitiría modificar su bucle interno). Además, el atributo *lengths* almacena el array L definido en los fundamentos teóricos de esta estructura, pero utilizando un vector de enteros no compacto (*Vec<usize>*), debido a que el tamaño de este array L será muy pequeño en tamaño y el balance entre la reducción del espacio utilizado y el aumento del tiempo de acceso que nos ofrece la alternativa utilizando un vector compacto de enteros no es favorable. De esta manera, utilizando ambos conceptos anteriormente mencionados, se consigue que la estructura sea eficiente tanto en tiempo de acceso como en espacio utilizado, favoreciendo el cumplimiento del requisito no funcional RNF1.

6.5 Wavelet Tree

El diagrama de clases para la estructura de datos *wavelet tree* se puede ver en la figura 6.8, cumpliendo de esta manera con el requisito funcional RF4. En dicho diagrama, la clase ***WaveletTree<T>*** es la encarga de implementar esta estructura, donde el tipo *T* denota el tipo de dato que se utilizará en cada nodo para almacenar la información del vector de bits característico de esta estructura. Se puede observar que el diseño se asemeja mucho a lo mencionado en los aspectos teóricos de las sección 2.6, donde es notable la utilización de nuevas interfaces que representarán a las operaciones de $access$, $rank$ y $select$ sobre símbolos pertenecientes a un alfabeto dado, las cuales son, respectivamente, *CharacterAccess*, *CharacterRank* y *CharacterSelect*. Estas interfaces se pueden ver como la generalización de las interfaces *Access*, *Rank* y *Select* sobre caracteres. El *wavelet tree* es una estructura autosuficiente para responder a las consultas anteriormente expuestas, no siendo una estructura auxiliar como las mencionadas en la sección 6.3.

En el diseño de esta estructura es destacable la relación de composición con la clase abstracta ***WaveletTreeNode<T>***, la cual tendrá las variantes *Internal* y *Leaf*, representando a los nodos internos y a las hojas del árbol, respectivamente. La clase *WaveletTree<T>* tendrá asociado un único nodo inicial, el cual será la raíz del árbol y, a partir del mismo, se podrá acceder a todos los nodos que lo conforman. El diseño de esta estructura está basado enormemente en el **patrón de diseño *composite*** [80], el cual nos permite crear relaciones en forma de árbol

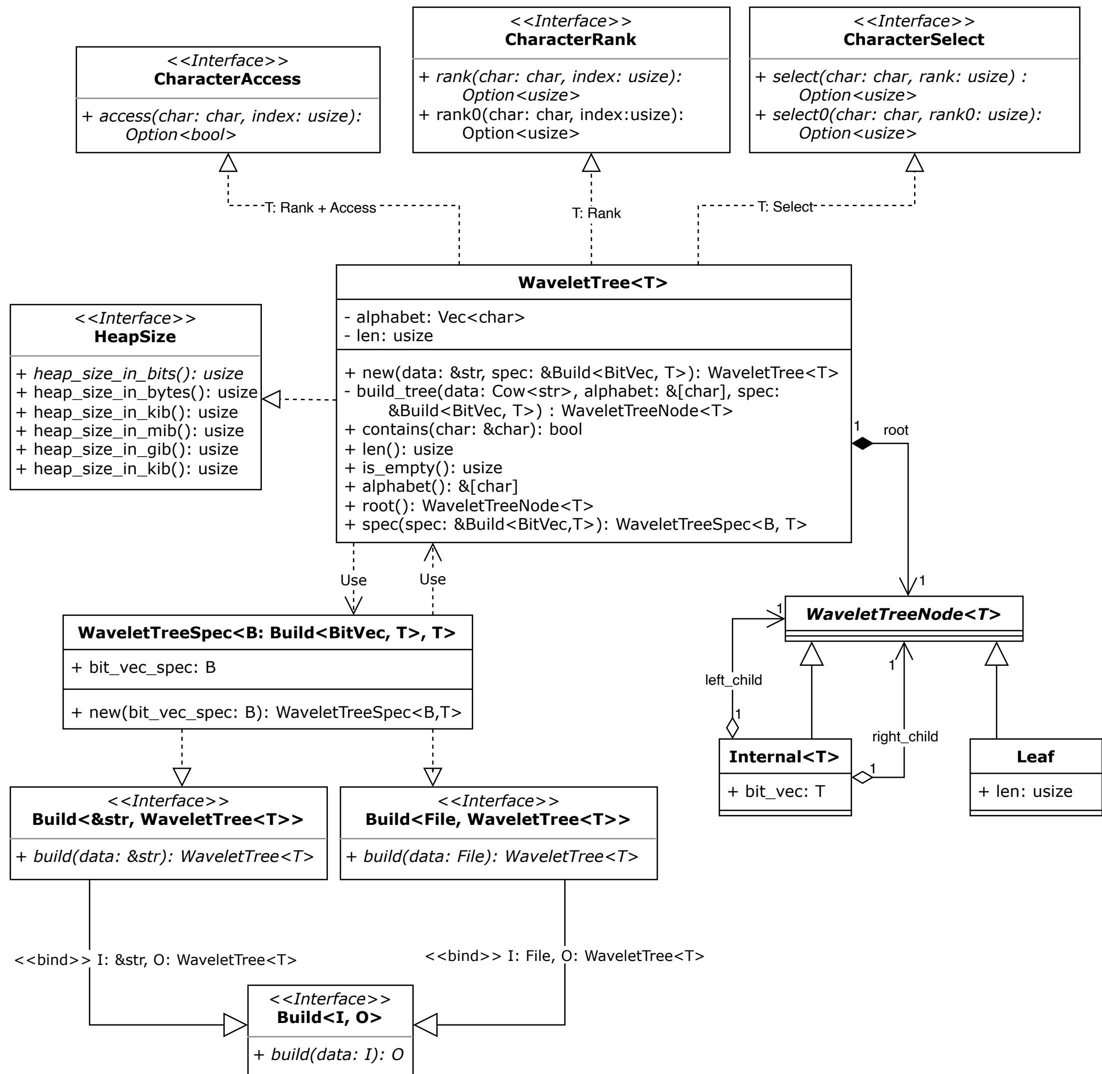

Figura 6.8: Diagrama de clases del *Wavelet Tree*

de manera muy sencilla y estructurada, consiguiendo así un diseño muy elegante y fácil de entender. Además, el uso de este patrón beneficiará enormemente la implementación, simplificando enormemente el recorrido y las búsquedas puesto que favorece la reutilización del código y la modularidad.

En la clase *WaveletTree* podemos ver el primer uso del **sistema de *specs***, puesto que en su método *new* se recibirá como parámetro un *spec* que implemente la interfaz *Build<BitVec, T>*, es decir, un *spec* que a partir de un vector de bits plano (*BitVec*) permita construir una estructura cualquier de tipo *T*. Este *spec* será utilizado para construir los nodos internos del árbol, dado que serán estos los que almacenen dicha estructura *T* creada. Las operaciones que podrá realizar la clase *WaveletTree* dependerán en gran medida de las operaciones que

pueda soportar la estructura de tipo T que se utilice, restricciones que son establecidas por los algoritmos de $rank$, $select$ y $access$ para el *wavelet tree* descritos en la sección 2.6. Estas restricciones que tiene que cumplir la estructura *T* están especificadas mediante las relaciones de implementación con las interfaces *CharacterAccess*, *CharacterRank* y *CharacterSelect* por la clase *WaveletTree<T>* en la figura 6.8.

Además, se puede ver que, de forma similar a lo ocurrido con otras estructuras, el *spec* correspondiente al *wavelet tree* es la clase *WaveletTreeSpec*, la cual implementa la interfaz *Build<&str, WaveletTree<T>>*, de forma que se pueda construir un *wavelet tree* a partir de una cadena de texto en memoria, y también implementa la interfaz *Build<File, WaveletTree<T>>*, permitiendo también la construcción a partir de un archivo en disco.

Mediante el sistema de *specs*, por lo tanto, podemos utilizar las estructuras de *BitVec*, *SparseSamplingRank*, *DenseSamplingRank* y *RRRBitVec* como estructuras internas para el *wavelet tree*, pudiendo conseguir de esta manera múltiples variantes prácticas muy interesantes de esta estructura, permitiendo además que en un futuro se puedan utilizar nuevas estructuras que implementen las operaciones de $rank$, $select$ y $access$ que se desarrollen. Además, cabe destacar que la variante de *wavelet tree* que utiliza la clase *RRRBitVec* como estructura interna es la que se corresponde con el *wavelet tree* comprimido con H_0, mencionado en la sección 2.6.4.

Con todo lo anteriormente mencionado se consigue que el *wavelet tree* sea altamente configurable en tiempo de ejecución y extensible, cumpliendo finalmente con los requisitos RF7 y RNF5.

Capítulo 7

Implementación y pruebas

En este capítulo se hablará de los detalles de implementación en el proceso de desarrollo de la librería, siguiendo el diseño propuesto en el capítulo 6. A su vez, también se explicarán los aspectos relacionados con las pruebas que se realizaron.

7.1 Implementación

En esta sección se tratarán las particularidades de la implementación de la librería. Si bien se ha realizado la implementación de todo lo comentado en el capítulo de diseño (capítulo 6), únicamente se tratarán aquí las soluciones y algoritmos más complejos que merezcan una mención especial.

Durante todo el proceso de implementación se han tomado como referencia las recomendaciones oficiales del lenguaje Rust de implementación y diseño de API [81]. Este conjunto de recomendaciones establece una serie de pautas para que las librerías que se desarrollen en Rust sean de calidad y fáciles de utilizar por otros desarrolladores. Ejemplos de estas pautas pueden ser el nombrado de las funciones y *structs*, el aseguramiento de que los nuevos *structs* que se definan implementen los *traits* de la librería estándar de Rust, de forma que se favorezca la interoperabilidad con otras librerías, o cómo se tiene que realizar la documentación y las secciones que esta debe contener. El seguimiento de estas recomendaciones favoreció enormemente el cumplimiento de los requisitos no funcionales RNF2, RNF7 y RNF8.

En lo referente a la gestión del código de la librería, se hizo uso de la herramienta *Cargo* [67], la cual es el gestor de paquetes y proyecto oficial de Rust. Esta herramienta nos permite crear un proyecto de Rust, compilarlo, ejecutarlo, probarlo y crear documentación de forma muy sencilla. El uso de esta herramienta durante el proceso de desarrollo fue clave para cumplir lograr un desarrollo ágil y eficiente, ya que nos permite centrarnos en el desarrollo de la librería y facilitar la integración de la librería con otros proyectos, cumpliendo además con el requisito no funcional RNF8.

7.1.1 Optimizaciones de compilación

Rust es un lenguaje de programación compilado que utiliza la infraestructura de compilación de LLVM [82]. LLVM es un sistema de compilación muy potente que permite realizar optimizaciones a nivel de código intermedio y generar código máquina.

```
RUSTFLAGS="-C target-cpu=native -C opt-level=3 -C lto -C
panic=abort -C codegen-units=1" cargo build --release
```

Código 7.1: Instrucción de compilación de la librería

La eficiencia de esta librería dependerá en gran medida de cómo se realice la compilación de la misma. Por lo tanto, un aspecto que se tuvo muy en cuenta para lograr que la librería fuese rápida y eficiente fue el estudio y aplicación de las posibles configuraciones y optimizaciones que se podían realizar con el compilador de Rust. En concreto, se puede ver la instrucción de compilación que se utilizó para generar la librería en el código 7.1. En dichas instrucciones de compilación se indica que se compile en modo producción (*release*), de forma que se apliquen optimizaciones generales y se eliminen las comprobaciones de seguridad y símbolos de depuración que se realizan en modo desarrollo; que se compile para la arquitectura del procesador en el que se está compilando mediante el parámetro *target-cpu*, lo cual optimizará en gran medida las operaciones aritméticas que se utilicen, como puede ser la instrucción de *popcount* nativa de la CPU; que se apliquen optimizaciones de nivel tres mediante el parámetros *opt-level*, empleando así optimizaciones aún más agresivas como puede ser el desenrollado de bucles, *inlining* de funciones, etc; que se aplique Link Time Optimization (LTO), lo cual hará que se apliquen optimizaciones a nivel de enlazado, como puede ser la eliminación de código muerto; que se utilice el pánico abortivo mediante el parámetro *panic*, haciendo que el programa aborte en caso de que se produzca un evento de *panic*, en lugar de realizar un desenrollado de la pila de llamadas; y por último, que se utilice una única unidad de compilación (es decir, que no se compilen las dependencias en paralelo) mediante el parámetro *codegen-units*, permitiendo que se puedan aplicar ciertas optimizaciones que no se pueden lograr cuando se compilan las dependencias en paralelo, a expensas de incrementar el tiempo de compilación.

Además, se anotaron las funciones que se considerarían como candidatas a la optimización de *inlining* mediante el atributo *inline* de Rust. Esto permite que el compilador pueda realizar las llamadas optimizaciones de *inlining* de funciones, lo cual consiste en que el compilador sustituya la llamada a la función por el código de la misma, evitando de esta forma la sobrecarga de realizar una llamada a función. Esto es especialmente útil para funciones que se llamen muchas veces en un bucle y su tamaño en código máquina sea pequeño. La desventaja de esta optimización es que aumenta el tamaño de los binarios generados, ya que se

estaría duplicando el código de la función en cada lugar en el que se llame. Se puede ver un ejemplo de esto en el código 7.2, donde se define una pequeña función que realiza la suma de dos números que está anotada con el atributo *inline*, de forma que si el compilador considera que es conveniente, sustituirá la llamada a la función por el código del cuerpo de la misma.

```
#[inline]
fn sum(a: u32, b: u32) -> u32 {
    a + b
}
```

Código 7.2: Ejemplo de función que se puede optimizar con *inlining*

Finalmente, todos estos conceptos de optimización fueron clave para lograr cumplir finalmente con el requisito no funcional RNF1. Cabe destacar que se realizó un gran esfuerzo en investigación, prueba y comprensión sobre todas estas optimizaciones y configuraciones, ya que la documentación sobre esta temática no es muy extensa.

7.1.2 Utilización del API *unsafe* de Rust

Para poder lograr exprimir al máximo el rendimiento de la librería, se tuvo que hacer uso del API *unsafe* de Rust [83]. Esta funcionalidad del lenguaje permite evitar que se realicen ciertas comprobaciones de seguridad (por ejemplo, como se ha mencionado anteriormente, el bounds checking) a cambio de que el programador asuma la responsabilidad de que el código que se está ejecutando es seguro. Esto permite que se puedan realizar optimizaciones a nivel de compilación que de otra forma no serían posibles, ya que el compilador no podría garantizar que el código es seguro sin la información sobre las precondiciones que posee el desarrollador.

Por lo tanto, podemos ver el uso de esta funcionalidad en varias de las estructuras de datos que se han implementado, viendo un ejemplo de esto mismo para la operación de lectura de un entero sobre un vector de bits (*BitVec*) en el código 7.3. En este caso, se puede ver que la función *read_bits_unchecked* está anotada con el atributo *unsafe*, puesto que utiliza la función *get_unchecked* del vector de la librería estándar de *Rust*. Este último método nos permite obtener una referencia a un elemento del vector sin realizar comprobaciones de seguridad, como puede ser la comprobación de que el índice al que se está accediendo está dentro de los límites del vector, haciendo que de esta manera el código sea mucho más rápido.

```
struct BitVec{
    raw_data: Vec<usize>,
    len: usize,
}
impl BitVec{
    // In 64 bits systems, CONTAINER_WIDTH is 64 bits.
```

```
    // In 32 bits systems, CONTAINER_WIDTH is 32 bits.
    const CONTAINER_WIDTH: usize = std::mem::size_of::<usize>() * 8;

    #[inline]
    pub unsafe fn read_bits_unchecked(&self, index: usize, len:
    usize) -> usize {
        if len == 0 {
            return 0;
        }
        let offset = index % BitVec::CONTAINER_WIDTH;
        let index = index / BitVec::CONTAINER_WIDTH;

        let w1 = self.raw_data.get_unchecked(index) >> offset;
        if offset + len > BitVec::CONTAINER_WIDTH {
            let w2 = self.raw_data.get_unchecked(index + 1);
            let read_bits = BitVec::CONTAINER_WIDTH - offset;
            let rem_bits = len - read_bits;
            w1 | ((w2 & (usize::MAX >> (BitVec::CONTAINER_WIDTH -
    rem_bits))) << read_bits)
        } else {
            w1 & (usize::MAX >> (BitVec::CONTAINER_WIDTH - len))
        }
    }
}
```

Código 7.3: Ejemplo de uso del API *unsafe* de Rust

Podemos ver un ejemplo de uso del método *read_bits_unchecked* de la clase *BitVec* dentro de la función *get* de la clase *CompactIntVec*. Si sabemos que el índice al que se está accediendo en el vector de enteros compactos es correcto, no haría falta volver a realizar la comprobación de índices cuando queramos recuperar los datos de su vector de bits interno (siempre y cuando se haya programado correctamente la clase *CompactIntVec*). Por lo tanto, podemos utilizar la funcionalidad *unsafe* para poder acceder a los datos del vector compacto de enteros de forma mucho más rápida (lo que beneficia a muchas otras estructuras de datos de la librería), como se puede ver en el código 7.4.

```
struct CompactIntVec {
    raw_data: BitVec,
    width: usize,
    len: usize,
}
impl CompactIntVec {
    #[inline]
    pub fn get(&self, index: usize) -> Option<usize> {
        if index >= self.len() {
```

```
            return None;
        }
        Some(unsafe { self.get_unchecked(index) })
    }
}
```

Código 7.4: Ejemplo de uso del API *unsafe* de Rust

Se hizo uso de la funcionalidad *unsafe* de este lenguaje en muchas más ocasiones a lo largo de la implementación de las estructuras de datos de la librería, pese a que no se muestren más ejemplos de esto mismo. Cabe destacar la gran dificultad de escribir código que haga uso de esta funcionalidad, ya que se tiene que estar muy seguro de que el código no tendrá ningún tipo de error, además de que se han de tener en cuenta todos los posibles casos de uso límite que se puedan dar.

Finalmente, se considera que todo lo mencionado en esta sección ha sido un aspecto clave para lograr cumplir con el requisito no funcional RNF1.

7.1.3 Optimización del vector de bits comprimido con H_0

Primero se hizo una implementación inicial muy sencilla de esta estructura de datos, de una forma muy directa y siguiendo en gran medida la definición teórica de esta estructura (sección 2.5). Esta implementación simple resultó que no era competitiva en términos de rendimiento con otras implementaciones de la misma (por ejemplo, la de la librería SDSL), llegando a ser hasta un orden de magnitud más lenta. Para solucionar este problema, se realizaron una serie de optimizaciones que se detallarán a continuación, las cuales sirven para lograr cumplir con el requisito no funcional RNF1.

Codificación y decodificación de bloques de bits

La estructura *RRRBitVec* tiene dos métodos privados (*decode* y *encode*) que son los encargados de codificar y decodificar un bloque de b bits. El rendimiento de la estructura depende en gran medida de la eficiencia de estos dos métodos, por lo que se hizo un gran esfuerzo en optimizarlos. Se realizaron una serie de optimizaciones que se detallarán a continuación.

La primera optimización que se realizó fue la de precalcular los valores para los coeficientes binomiales. Calcular coeficientes binomiales de la forma $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ es una operación muy costosa, ya que requiere emplear el operador factorial. Además, si dicho cálculo no se hace con cuidado, es posible que se produzca un desbordamiento de enteros, ya que aunque el resultado de la operación sea un número menor que el entero máximo representable, el valor de $n!$ puede no serlo. Para poder realizar la codificación y decodificación *on-the-fly* [49] de un único bloque es necesario utilizar varios valores de coeficientes binomiales, los cuales se tendrían que calcular de forma repetida en muchas ocasiones.

Para evitar repetir cálculos, se utilizó una tabla de coeficientes binomiales precalculada **en tiempo de compilación**, de forma que se pudiesen utilizar los valores de los coeficientes binomiales de forma muy rápida, con un único acceso a memoria. El cálculo de esta tabla se realizó mediante la funcionalidad *const* del lenguaje Rust, la cual nos permite ejecutar funciones puras cuyo resultado es constante y se pueden ejecutar en tiempo de compilación. Esta tabla se calculó mediante un algoritmo de **programación dinámica**, basado en una definición recursiva de los coeficientes binomiales, la cual se puede ver en la Ecuación (7.1). Utilizando este algoritmo se consigue lograr un espacio ocupado de memoria en el orden de $\mathcal{O}(b^2)$ y una complejidad computacional de $\mathcal{O}(b^2)$, donde b es el parámetro utilizado por la estructura *RRRBitVec* que define el tamaño de los bloques de bits y que típicamente es un valor pequeño (menor a 64 bits), por lo que el espacio de memoria utilizado y el tiempo de ejecución del algoritmo es muy reducido. Además, este algoritmo tiene la ventaja de que no produce desbordamiento de enteros debido a cálculos intermedios (como sí ocurre utilizando la definición utilizando factoriales), por lo que si se cumple que el número $\binom{b}{b/2}$ es menor que el entero máximo representable, no se producirá ningún desbordamiento de enteros. En la práctica, el número $\binom{64}{32}$ es menor (en un orden de magnitud) que el entero sin signo máximo representable en una arquitectura de 64 bits (que es 2^{64}), por lo que no tendremos que preocuparnos por el desbordamiento en alguna posición de la tabla calculada para valores de b que cumplan que $0 < b \leq 64$

$$\binom{n}{k} = \begin{cases} 1 & \text{si}\, k = 0 \ \text{ o } \ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si}\, 0 < k < n \\ 0 & \text{en cualquier otro caso} \end{cases} \tag{7.1}$$

Esta optimización, que acelera en gran medida las operaciones de *encode* y *decode*, fue clave para incrementar enormemente el rendimiento de las operaciones de $rank$, $select$ y $access$, puesto que todos estos métodos hacen uso de la operación *decode*. Además, esta optimización también acelera la construcción de la estructura de datos, puesto que para ello se necesita la operación de *encode*.

Rank

Debido a que la operación de $rank$ es muy utilizada, y por lo tanto, es primordial que sea lo más rápida posible, se realizó una importante optimización sobre ella. Para realizar la operación $rank(B_{H_0},i)$, en última instancia habría que decodificar el bloque de b bits que se encuentra en la posición $\lfloor i/b \rfloor$ del vector de desplazamientos y, posteriormente, contabilizar el número de 1s que hay en los primeros i mod b bits de dicho bloque. Aunque no sea muy obvio, de esta manera estaríamos realizando trabajo repetido; primero, decodificaríamos el

bloque completamente a partir de su clase y desplazamiento, de forma iterativa y decodificando primero sus bits más significativos, para después realizar un desplazamiento a nivel de bit para quedarnos únicamente con los bits que nos interesan. Esto es ineficiente, debido a que lo que se puede hacer es detener la decodificación una vez se hayan obtenido los primeros $i \bmod b$ bits del bloque, de forma que no se decodifiquen los bits que no nos interesan. Para ello, aprovechándonos de que la decodificación se hace de forma iterativa y el orden de decodificación de los bits de un bloque se corresponde con el orden de los bits en el vector de bits, se realizó una optimización del algoritmo de *decode* al que se le pasa por parámetro el número de bits que se quieren decodificar. Se puede ver dicho algoritmo en el código 7.5.

```
#[inline]
fn decode(class: usize, offset: usize, b: usize, len: usize) ->
usize {
    if len == 0 {
        return 0;
    }
    // OPTIMIZATION: There is only one possible combination if
    // the class is `b` or `0`
    if class == b {
        // The block is all ones
        return usize::MAX >> (usize::BITS - len as u32);
    } else if class == 0 {
        // The block is all zeros
        return 0;
    }
    let mut block = 0;
    let mut current_class = class;
    let mut current_offset = offset;
    let mut i = 0;
    while current_class > 1 {

        // OPTIMIZATION: decode only `len` bits
        if i >= len {
            return block;
        }
        let num_previous_offsets = BINOMIALS[b - i -
1][current_class];
        if current_offset >= num_previous_offsets {
            // If the bit at ith position (starting from the
right) is set,
            // there are BINOMIALS[b-i-1][current_class]
combinations that
            // have a 0 in that position and precede that offset
            block |= 1 << i;
```

```
                current_offset -= num_previous_offsets;
                current_class -= 1;
            }
            i += 1;
        }
        // OPTIMIZATION:
        // In case the class is 1, the bit at position
        // b - offset - 1 is set
        // since there are only b possible combinations
        if current_class > 0 {
            let bit_offset = b - current_offset - 1;
            if bit_offset < len {
                block |= 1 << bit_offset;
            }
        }
        block
    }
```

Código 7.5: Algoritmo optimizado de decodificación de un bloque de bits

Además de la optimización anteriormente mencionada, también se puede ver en el algoritmo del código 7.5 un par de optimizaciones más, las cuales se pueden ver en los comentarios del código. La primera de ellas es que si la clase del bloque es b o 0, no es necesario realizar ninguna decodificación, ya que el bloque es todo 1s o todo 0s, respectivamente. La otra optimización es que si la clase del bloque tiene exactamente el valor 1, entonces entre los índices *current_offset* y b del bloque, hay únicamente un bit que está a 1, el cual se encuentra exactamente en la posición $b - current_offset - 1$ del bloque (empezando a contar en el índice 0), debido a que hay únicamente b posibles combinaciones para el bloque restante y por definición, estas combinaciones tienen un desplazamiento asignado acorde a su orden dentro de la clase. Estas dos optimizaciones se aprovechan de varias de las propiedades de los números combinatorios y de la definición que se hizo en la sección 2.5 sobre la codificación de los bloques de bits.

7.1.4 Serialización y deserialización

La serialización se refiere al proceso de convertir una estructura de datos en un formato que pueda ser almacenado o transmitido a través de una red, mientras que la deserialización es el proceso inverso; utilizar los datos serializados como entrada y reconstruir la estructura de datos original. *Serde* es una librería de Rust que proporciona un conjunto de macros y *traits* para implementar la serialización y deserialización de forma sencilla, genérica y eficiente. La utilización de esta librería es un estándar de facto en el ecosistema de Rust, según lo declaran las recomendaciones oficiales [81].

Para poder soportar las funcionalidades de serialización y deserialización y cumplir con el requisito funcional RF5 se utilizó la librería *serde*. Para ello, se anotaron todas las estructuras de datos de la librería con el atributo *derive(Serialize, Deserialize)*, de forma que se implementase mediante un tipo de macros llamadas *derive macros* [84] las interfaces *Serialize* y *Deserialize* de *serde*, las cuales indican cómo se debe serializar y deserializar una determinada estructura de datos. Se proporciona un ejemplo de esta anotación en el código 7.6, donde se puede ver esto implementado para la estructura de datos de $rank$ con dos niveles de directorio.

```
#[derive(Serialize, Deserialize)]
struct DenseSamplingRank {
    superblocks: Vec<usize>,
    superblock_size: usize,
    blocks: CompactIntVec,
    k: usize,
    total_rank: usize,
}
```

Código 7.6: Ejemplo de anotación de una estructura de datos con *serde*

La manera de funcionar del atributo *derive(Serialize,Deserialize)* es que, si todos los campos de una estructura dada implementan también las interfaces de *Serialize* y *Deserialize* (estas están implementadas por defecto para los tipos básicos de la librería estándar de Rust), entonces se puede implementar de forma automática la serialización y deserialización de dicha estructura de datos. Esto es muy útil, ya con muy poca cantidad de código se puede lograr implementar funcionalidades muy complejas como son la serialización y deserialización.

La utilización de *serde* únicamente nos indica la manera en la que se debe serializar y deserializar una estructura de datos, pero no especifica el formato en el que se debe serializar, evitando así el acoplamiento a un formato concreto. Para lograr esto, *serde* se basa enormemente en el **patrón de diseño adaptador**. Entonces, para poder serializar a un formato determinado, se deben utilizar otras librerías (o proveer de un serializador personalizado) como pueden ser *serde_json* para serializar a formato JSON, *serde_protobuf* para serializar a formato Protobuf o *bincode* para serializar a formato binario. De esta forma, se cumplen con los requisitos RF5 y RNF9.

Cabe destacar que la implementación de esto fue muy sencilla, pero se debe a que se tuvo en cuenta desde un principio todo esto, además de orientar el diseño mostrado en el capítulo 6 a que la serialización y deserialización fuese muy sencilla. Por ejemplo, la decisión de diseño de que las estructuras auxiliares (*SparseSamplingRank* y *DenseSamplingRank*) no tuviesen referencias al vector de bits sobre el que están construidas se debe en gran medida a esto, puesto que las referencias a otras estructuras complican mucho la serialización y deserialización, pero si todas la estructuras tienen campos de los que son propietarias (es decir, sin referencias),

el soporte de esta funcionalidad es trivial.

La serialización y deserialización de las estructuras de datos que se provee en esta librería desbloquea muchas nuevas posibilidades, como por ejemplo la de poder enviar una estructura de datos a otro servicio a través de una red, la de poder almacenar una estructura de datos en un fichero para poder recuperarla posteriormente, o gestionar las estructuras de datos construidas de una forma más estructurada mediante una base de datos (por ejemplo, utilizando el formato BSON y la base de datos *MongoDB* [85]).

7.1.5 Documentación

En Rust, la documentación se crea utilizando comentarios especiales llamados *doc comments*. Estos comentarios se parecen a los comentarios regulares de código, pero comienzan con tres barras (*///*).

Los *doc comments* se utilizan para generar la documentación de forma automática a partir del código fuente (por poner un ejemplo similar, parecido a lo que ocurre con *JavaDoc* en el lenguaje Java). Para generar la documentación, se utiliza la herramienta *rustdoc*, la cual viene incluida con el compilador de Rust. La principal ventaja de escribir así la documentación es que se facilita la creación de documentación precisa y actualizada, puesto que esta está localizada cerca del código relevante a la misma. Se puede ver en el código 7.7 un ejemplo de un *doc comment* utilizado en la librería.

```
/// Trait that defines structures that themselves full support rank.
trait Rank {
    /// Returns the number of 1s in the bit vector in the range [0,
    index).
    /// If index is out of bounds, returns None.
    /// By definition, rank(0) == 0
    fn rank(&self, index: usize) -> Option<usize>;

    /// Returns the number of 0s in the bit vector in the range [0,
    index).
    /// If index is out of bounds, returns None.
    /// By definition, rank0(0) == 0
    fn rank0(&self, index: usize) -> Option<usize> {
        self.rank(index).map(|rank| index - rank)
    }
}
```

Código 7.7: Ejemplo de *doc comment* para el *trait Rank*

Se ha intentado documentar la librería implementada utilizando *doc comments*, de forma que se logre cumplir con el requisito no funcional RNF7 de forma satisfactoria. Finalmente,

se dispuso dicha documentación en una página web[1] mediante las herramientas *rustdoc* y *cargo*, además del repositorio *crates.io*, facilitando de esta manera el acceso y distribución de la misma. Se puede ver un ejemplo de la página web mencionada en la figura 7.1.

Figura 7.1: Página web de la documentación de la librería

7.2 Pruebas

Un aspecto clave para el correcto desarrollo del proyecto fue el empleo de la metodología TDD, la cual se explicó en la sección 4.1. Se implementaron numerosas pruebas unitarias antes de realizar la implementación de cada una de las estructuras de datos, de forma que dichas pruebas sirviesen para validar el correcto funcionamiento de las mismas, logrando así cumplir con los requisitos no funcionales RNF3 y RNF4 de forma satisfactoria.

Se utilizaron dos tipos principales de pruebas unitarias; **pruebas clásicas basadas en ejemplos** y **pruebas parametrizadas**.

[1] https://docs.rs/faex

Las **pruebas unitarias clásicas basadas en ejemplos** son el tipo de prueba más sencillo, las cuales se basaban en probar que las operaciones sobre las estructuras de datos se realizaban correctamente, utilizando ejemplos concretos de entrada y configuraciones concretas de las estructuras, verificando que se cumplían ciertas aserciones sobre los resultados de las operaciones. Este tipo de pruebas no tienen necesidad de especial mención más allá de la dificultad de realizar pruebas exhaustivas que cubriesen todos los casos posibles, tanto casos normales como casos límite, de manera que se consiga verificar el comportamiento en el mayor número de situaciones posible.

En cambio, las **pruebas unitarias parametrizadas** [86, 87] son un tipo de pruebas muy similares a las anteriores, pero que se basan en la generación automática de casos de prueba a partir de una serie de parámetros de entrada. La principal característica de estas pruebas es que podemos emplear, de igual manera que en las pruebas clásicas, una serie de ejemplos que verifiquen el comportamiento de ciertas estructuras, pero variando la configuración de las estructuras de datos a probar. De esta forma, se pueden probar muchas más configuraciones de las estructuras de datos utilizando pruebas genéricas como si fuese una plantilla. La ventaja que nos proporciona este tipo de pruebas es que nos permite verificar el comportamiento externo de una estructura sin necesidad de conocer su implementación interna, por ejemplo, la operación de $rank$ sobre vectores de bits debería exhibir el mismo comportamiento en las clases *BitVec*, *RRRBitVec*, *SparseSamplingRank* y *DenseSamplingRank*, independientemente de la configuración de cada una, por lo que podríamos utilizar una prueba genérica que defina cómo se debe comportar la operación de $rank$ y que se generen de forma automática casos de prueba para cada una de las clases mencionadas. Además, este tipo de pruebas permite la reutilización de casos de prueba y, por lo tanto, de código, puesto que la instanciación de un caso de prueba parametrizado con una configuración de parámetros concreta genera como resultado una prueba unitaria clásica, logrando así probar múltiples configuraciones de las estructuras de datos, logrando así cumplir con el requisito no funcional RNF5.

La implementación de las pruebas parametrizadas se hizo mediante el sistema de *macros* del lenguaje Rust, el cual nos permite emplear técnicas de metaprogramación, es decir, programar sobre el propio código fuente a nivel de *token*. Mediante las *macros* se definirán las pruebas parametrizadas de forma genérica, de manera que cuando se llame a la *macro* utilizando como parámetro una configuración concreta, se generen automáticamente las pruebas unitarias clásicas correspondientes. Se puede ver un ejemplo de esto en el código 7.8.

```
macro_rules! test_rank_for {
    ($t:ty, $($args:tt)*) => {
        mod rank{
            const BIG_BITVEC_SIZE: usize = 10_000;
            #[test]
            fn rank_when_all_ones() {
```

```
                let bv = BitVec::from_value(true, BIG_BITVEC_SIZE);
                let spec = <$t>::spec($($args)*);
                let rs = spec.build(bv);

                for i in 0..BIG_BITVEC_SIZE {
                    assert_eq!(rs.rank(i), Some(i));
                }
            }

        }
    };
}
macro_rules! test_constant_time_for_k{
    ($( $k: expr ),*) => {
        $(
            mod when_k_is_$k{
                test_rank_for!(DenseSamplingRank, $k);
            }
         }
        )*
    }
}
test_constant_time_for_k!(1, 2, 4, 5, 8, 16, 20, 32);
```

Código 7.8: Ejemplo de *macro* para generar pruebas parametrizadas

En este ejemplo se puede ver que la *macro test_rank_for* se encarga de generar una prueba que verifique el comportamiento de la función de $rank$ para una estructura de datos y parámetros de configuración para dicha estructura que se pasen por argumento (esta *macro* es, en esencia, una función variádica que admite un número variable de argumentos, útil en caso de que la estructura a probar admita también más de un parámetro de configuración). En la realidad, esta *macro* define muchos más tests que verifican el comportamiento de la función de $rank$ en otras muchas situaciones, pero esto se ha omitido en el ejemplo por razones de simplicidad. De una forma similar, la *macro test_constant_time_for_k* admite como parámetro el valor k de la estructura de datos *DenseSamplingRank*, de forma que se generen automáticamente pruebas de $rank$ para un valor de k indicado (en realidad, en esta *macro* también se generarían otras pruebas, como por ejemplo comprobar la correcta construcción de la estructura con un parámetro k determinado), utilizando una llamada a la macro *test_rank_for*. Por último, este proceso se cierra con la llamada en la última línea del ejemplo a la *macro test _constant_time_for_k*, con varios parámetros de k distintos (1, 2, 4, 5...), instanciando así las pruebas definidas de forma genérica con valores concretos de los parámetros de la estructura y generando de forma automática las pruebas unitarias clásicas correspondientes.

El uso de las pruebas parametrizadas fue muy útil en el desarrollo de la librería, puesto

que, por ejemplo, una vez definidas las pruebas parametrizadas (siendo genéricas tanto sobre la estructura concreta como sobre sus parámetros de configuración) del comportamiento que deberían exhibir las operaciones de $rank$ y $select$, se pudo validar de forma muy rápida y sencilla el correcto funcionamiento de las estructuras de datos *BitVec*, *RRRBitVec*, *SparseSamplingRank* y *DenseSamplingRank*, facilitando enormemente el trabajo con la metodología TDD y favoreciendo en gran medida la reutilización del código de las pruebas. Se pueden ver más detalles sobre las pruebas unitarias en la sección A.5 del anexo.

7.3 Licencia y distribución

De manera que la librería desarrollada pueda ser utilizada por otras personas pertenecientes tanto a la comunidad científica como a la comunidad de desarrolladores de Rust, se ha liberado el código fuente de la misma como código libre bajo la licencia MIT [88]. Esta licencia es una licencia de software libre muy permisiva que se adecúa perfectamente a los requisitos de este proyecto, posibilitando de esta manera a los desarrolladores hacer prácticamente cualquier cosa que deseen con el código fuente. Esto incluye su uso, modificación, distribución y utilización en proyectos privados o comerciales, favoreciendo así su adopción y popularidad. Además, esta licencia es muy sencilla y la responsabilidad del autor es muy limitada, puesto que no se ofrece ninguna garantía ni responsabilidad sobre el código fuente en caso de que ocurra algún problema.

Para la distribución del código fuente se ha utilizado *GitHub* [73], una plataforma de desarrollo colaborativo de software, alojando allí el código fuente de la librería[2] y facilitando la colaboración de otros desarrolladores en el proyecto. Además, para favorecer y simplificar el uso de la librería en otros proyectos del lenguaje Rust, se ha publicado la misma[3] en el repositorio oficial de librerías (también llamadas *crates*) de Rust, *crates.io* [64], de forma que la distribución de esta librería se integre perfectamente en el ecosistema del lenguaje. Por último, cabe destacar que se ha utilizado una herramienta de *GitHub* llamada *GitHub Actions* para establecer un sistema de integración continua en el repositorio del proyecto, de forma que se ejecuten automáticamente ciertas comprobaciones sobre el código fuente cada vez que se realice un *push* al mismo, mejorando así la experiencia de desarrollo colaborativa. Estas comprobaciones son el correcto formateo y estilo del código fuente, la correcta compilación del proyecto y la ejecución satisfactoria de todas las pruebas unitarias definidas.

Con todo esto, se logra cumplir con los requisitos no funcionales RNF8 y RNF10 de forma satisfactoria.

[2] https://github.com/jorgehermo9/faex
[3] https://crates.io/crates/faex

Capítulo 8

Experimentos y comparativa

En este capítulo se presentará la evaluación experimental que se llevó a cabo para poder realizar una comparativa de rendimiento en espacio y tiempo entre las estructuras de datos implementadas en este trabajo y las estructuras de datos de la librería SDSL. Únicamente se realizará una comparativa con las estructuras de datos de la librería SDSL debido a que es la mejor solución existente en el estado del arte de este campo, tanto en términos de rendimiento como en funcionalidades disponibles.

8.1 Marco experimental

En la evaluación experimental se ha utilizado el ordenador personal mencionado en la sección 4.2.1. Para la compilación de la librería desarrollada en este proyecto se ha utilizado el compilador *rustc* versión *1.71.0* con las opciones de compilación mencionadas en la sección 7.1.1. Así mismo, el compilador utilizado para compilar la librería SDSL ha sido *gcc* versión *13.2.1* con las opciones de compilación *-O3 -DNDEBUG -march=native*, de forma que la comparativa entre estas dos soluciones fuese lo más justa posible.

Para realizar los experimentos de la sección 8.2.1 se utilizaron como entrada vectores de bits con un tamaño de 10^9 elementos. De manera que se pudiese ilustrar el comportamiento de las distintas estructuras, se utilizaron varios vectores de bits con una densidad de bits (definida como el ratio de bits a uno) distinta, concretamente se emplearon densidades del 5 %, 50 % y 90 %. Además, las posiciones de los bits con valor uno de estos vectores de bits se generaron siguiendo una distribución aleatoria uniforme, de forma que se cumplan las densidades especificadas.

En los experimentos de la sección 8.2.2 se utilizaron dos corpus de texto de la colección *Pizza&Chili* [54], concretamente, los corpus *proteins*[1] y *english*[2], ocupando ambos un tamaño

[1] https://pizzachili.dcc.uchile.cl/texts/protein/
[2] https://pizzachili.dcc.uchile.cl/texts/nlang/english/

Corpus	Nº de caracteres	Tamaño del alfabeto	H_0
proteins	209.715.200	25	4,201
english	209.709.544	948	4,525

Tabla 8.1: Propiedades de los corpus de texto utilizados en los experimentos

de 200 MB (codificados en UTF-8). El corpus *proteins* consiste en una secuencia de proteínas separadas por saltos de línea, donde cada uno de los 20 aminoácidos que componen las proteínas se codifica con una letra mayúscula diferente. En cambio, el corpus *english* consiste en una concatenación de varios archivos de texto en lenguaje natural, en idioma inglés. Se puede ver en la tabla 8.1 ciertas propiedades de estos corpus de texto.

8.2 Evaluación experimental

En las próximas secciones se presentarán los resultados obtenidos en los experimentos realizados. Para favorecer la reproducibilidad de los mismos, se ha publicado el código fuente que se utilizó para realizar los experimentos en un repositorio de *GitHub* [3].

8.2.1 Vectores de bits

Los experimentos relacionados con las estructuras de datos relacionadas con los vectores de bits consisten principalmente en medir el tiempo de ejecución de las operaciones de $rank$ y $select$, a la vez que se mide el espacio en memoria que ocupa cada estructura de datos. En esta sección se mostrarán experimentos relacionados con las estructuras de tipo auxiliar (construidas sobre vectores de bits), los cuales se pueden ver en las figuras 8.1 y 8.2, y experimentos relacionados con las estructuras que son una representación alternativa de dicho vector de bits (las cuales son los vectores de bits comprimidos con H_0), ilustrados en las figuras 8.3 y 8.4.

Las estructuras de datos que se utilizarán en esta sección, junto con su identificador correspondiente, son las siguientes:

- *dense_sampling_rank*: Esta estructura se corresponde con el componente *DenseSamplingRank* de la librería desarrollada en este proyecto.

- *sparse_sampling_rank*: De la misma forma que en el caso anterior, esta estructura representa el componente *SparseSamplingRank*.

[3] https://github.com/jorgehermo9/faex-experiments

- *bit_vector_il*: Esta estructura forma parte de la librería SDSL y se corresponde con una estructura de $rank$ con un nivel de directorio, muy similar a la estructura *SparseSamplingRank*. De igual manera que su estructura análoga, esta estructura permite realizar las operaciones de $rank$ y $select$.

- *rank_support_v*: Esta estructura también forma parte de la librería SDSL y es una variante del $rank$ con dos niveles de directorio llamada *rank9* [58]. Además, es importante mencionar que esta estructura únicamente soporta la operación de $rank$, no permitiendo realizar la operación de $select$.

- *rank_support_v5*: Es una estructura muy similar a la anterior, perteneciente a su vez a la SDSL, pero consiste en una variante aún más compacta que logra utilizar menos espacio en memoria a costa de un rendimiento ligeramente inferior.

- *select_support_mcl*: Es la estructura de $select$ más rápida disponible en la librería SDSL, la cual permite realizar consultas de $select$ en tiempo constante. Es importante destacar que esta estructura no permite realizar la operación de $rank$.

- *rrr_faex_i*: Se corresponde con la estructura *RRRVec* de la librería desarrollada en el presente trabajo, con un tamaño de bloque $b = i$. Esta estructura es un vector de bits comprimido con H_0 que soporta las operaciones de $rank$ y $select$.

- *rrr_sdsl_i*: De igual manera que en el caso anterior, pero correspondiéndose con la clase *rrr_vector* de la librería SDSL. Esta estructura también soporta las consultas de $rank$ y $select$ y es un vector de bits comprimido con H_0.

En los experimentos se utilizaron varias configuraciones de los parámetros para obtener un balance entre el espacio ocupado y el tiempo de consulta de las operaciones, en la medida que las estructuras de datos utilizadas lo permitían. Para poder medir el tiempo que se emplea en las operaciones de consulta se computó el tiempo medio utilizado en realizar cada una de ellas. Más concretamente, se realizaron varias consultas de $rank$ sobre posiciones aleatorias de entre todas las posiciones del vector de bits. A su vez, las operaciones de $select$ se realizaron sobre valores de ocurrencia aleatorios comprendidos entre cero y el número total de unos del vector de bits. De esta manera se consiguió replicar un patrón de acceso a los datos que fuese lo más justo posible, en el sentido de que no se aprovechase de la localidad de los datos o propiedades similares que no tienen por qué ocurrir en todos los dominios.

Estructuras auxiliares

Las figuras 8.1 y 8.2 muestran los experimentos de las operaciones de $rank$ y $select$ respectivamente, para las estructuras auxiliares sobre vectores de bits. En dichas figuras se muestra

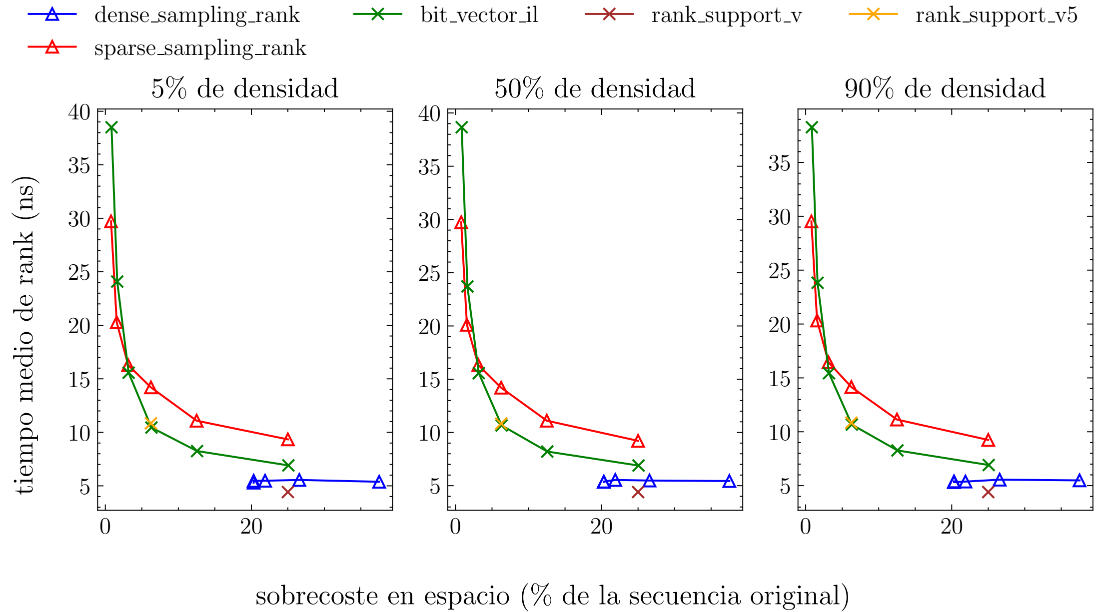

Figura 8.1: Balance entre el espacio ocupado y el tiempo de consulta para la operación de *rank* sobre posiciones aleatorias en las estructuras auxiliares construidas sobre vectores de bits. Se muestran experimentos para vectores de bits con un 5 %, 50 % y 90 % de densidad, que se corresponden con las gráficas de izquierda a derecha

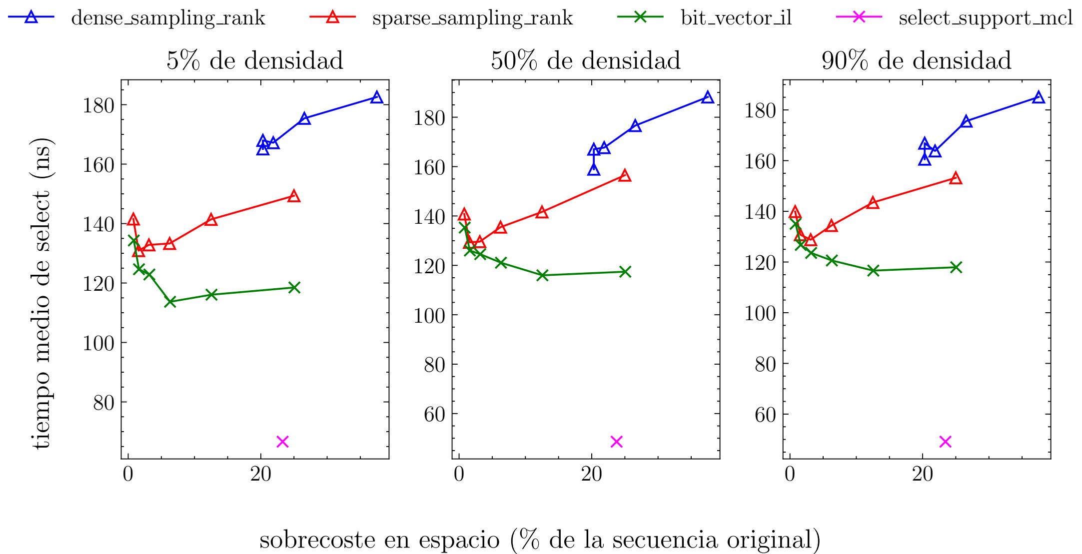

Figura 8.2: Balance entre el espacio ocupado y el tiempo de consulta para la operación de *select* sobre valores de ocurrencia aleatorios en las estructuras auxiliares construidas sobre vectores de bits. Se muestran experimentos para vectores de bits con un 5 %, 50 % y 90 % de densidad, que se corresponden con las gráficas de izquierda a derecha

el tiempo medio de consulta para cada una de las operaciones, así como el espacio ocupado por cada una de las estructuras. El espacio ocupado se muestra como el porcentaje de espacio ocupado por la estructura auxiliar respecto al tamaño del vector de bits original. Como las estructuras auxiliares no son una representación autosuficiente del vector de bits, se deben de utilizar junto con el vector de bits original, por lo que el espacio ocupado por la estructura auxiliar se considera como un espacio extra que se añade al espacio ocupado por el vector de bits original.

En los experimentos de la figura 8.1 se puede observar que, como era esperado, el rendimiento de la operación de $rank$ sobre estas estructuras se mantiene invariante según cambia la densidad del vector de bits. Además, se puede destacar que el rendimiento de las estructuras propuestas en este proyecto es muy similar al rendimiento de sus equivalentes en la librería SDSL, logrando conseguir una solución que sea competitiva en este aspecto con el estado del arte. En concreto, es notable que la estructura *dense_sampling_rank* tiene un rendimiento muy similar a la estructura de $rank$ más rápida disponible en la SDSL, que es la estructura *rank_support_v*, llegando lograr un espacio inferior que esta última mediante cambios en configuración de sus parámetros sin que su rendimiento se vea muy negativamente afectado.

Sobre los experimentos de la operación de $select$ ilustrados en la figura 8.2 se destaca que el rendimiento de la estructura *select_support_mcl* de la librería SDSL es muy superior al resto de estructuras, pero esta misma tiene la limitación de que únicamente permite realizar la operación de $select$. Además, se puede observar que para las estructuras *sparse_sampling_rank* y *dense_sampling_rank*, a pesar de que el rendimiento de la operación de $rank$ se viese incrementado a medida que se utiliza un mayor espacio por la estructura, el rendimiento de la operación de $select$ se ve degradado en una forma notable. Esto se debe a que las consultas de $select$ para encontrar una ocurrencia aleatoria produce un gran número de fallos caché en el algoritmo de búsqueda binaria utilizado para realizar esta consulta, lo que hace que la operación se vea especialmente perjudicada.

Finalmente, si se necesitasen soportar tanto las operaciones de $rank$ como las de $select$, la mejor solución por parte de la SDSL, en términos de velocidad de consulta, sería combinar las estructuras *rank_support_v* y *select_support_mcl*. Pero, una solución aún más eficiente sería utilizar únicamente la estructura *dense_sampling_rank* propuesta por este trabajo, logrando realizar ambas operaciones de una forma rápida y además empleando un menor espacio utilizado que la alternativa que se tendría que emplear en la SDSL. En este caso de uso, las soluciones propuestas por la librería desarrollada en este proyecto son superiores a las que provee la librería SDSL.

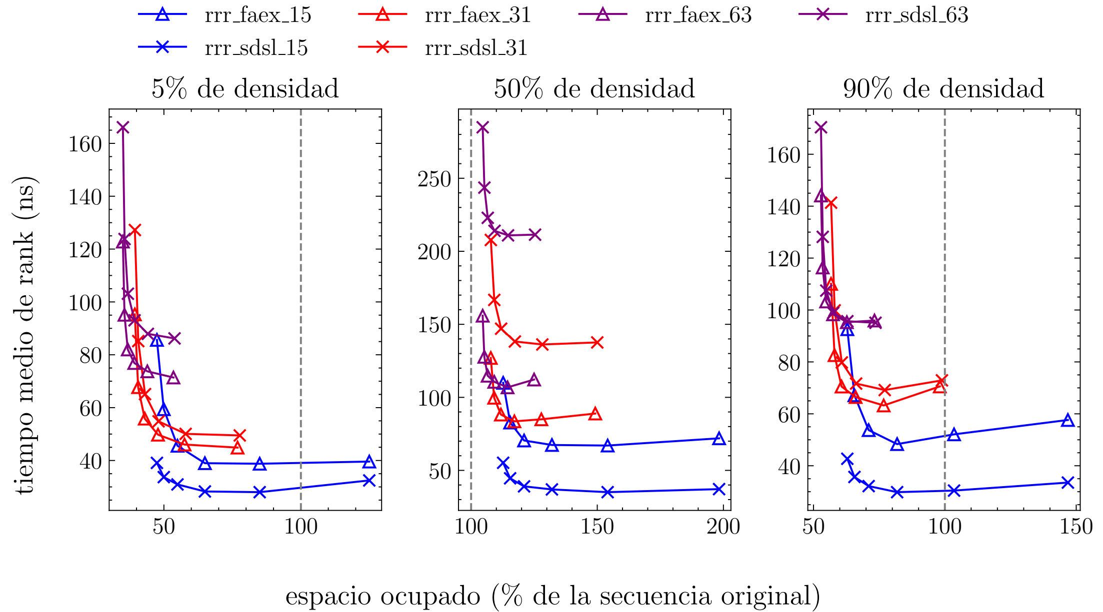

Figura 8.3: Balance entre el espacio ocupado y el tiempo de consulta para la operación de $rank$ sobre posiciones aleatorias en los vectores de bits comprimidos con H_0. Se muestran experimentos para vectores de bits con un 5 %, 50 % y 90 % de densidad, que se corresponden con las gráficas de izquierda a derecha. La línea discontinua vertical indica el tamaño del vector de bits original

Vectores de bits comprimidos con H_0

Los experimentos para los vectores de bits comprimidos con H_0 son muy similares a lo expuesto en la sección anterior. Se pueden ver los experimentos para la operación de $rank$ en la figura 8.3 y los experimentos para la operación de $select$ en la figura 8.4. En ambas figuras se muestra el tamaño utilizado por cada una de las estructuras como el porcentaje de espacio ocupado por la estructura respecto al tamaño del vector de bits original. Además, la línea discontinua vertical que se muestra en las gráficas indica el tamaño de dicho vector original.

En lo que respecta a la operación de $rank$, las soluciones de ambas librerías presentan un uso del espacio muy similar. Podemos observar que la solución propuesta por la SDSL funciona mejor en términos de tiempo en el caso de que se use un tamaño de bloque pequeño, de $b = 15$, pero en los otros dos casos, con tamaños de bloque $b = 31$ y $b = 63$, la alternativa propuesta en este proyecto es más rápida, siendo muy notable la diferencia en el caso de que la densidad del vector de bits sea del 50 %, aunque en este último caso no se consiga una compresión que reduzca el espacio ocupado de los datos. La configuración de tamaño de bloque más interesante en este caso es la de $b = 63$, puesto que es la configuración que alcanza una mayor compresión de los datos, siendo la solución propuesta por este proyecto

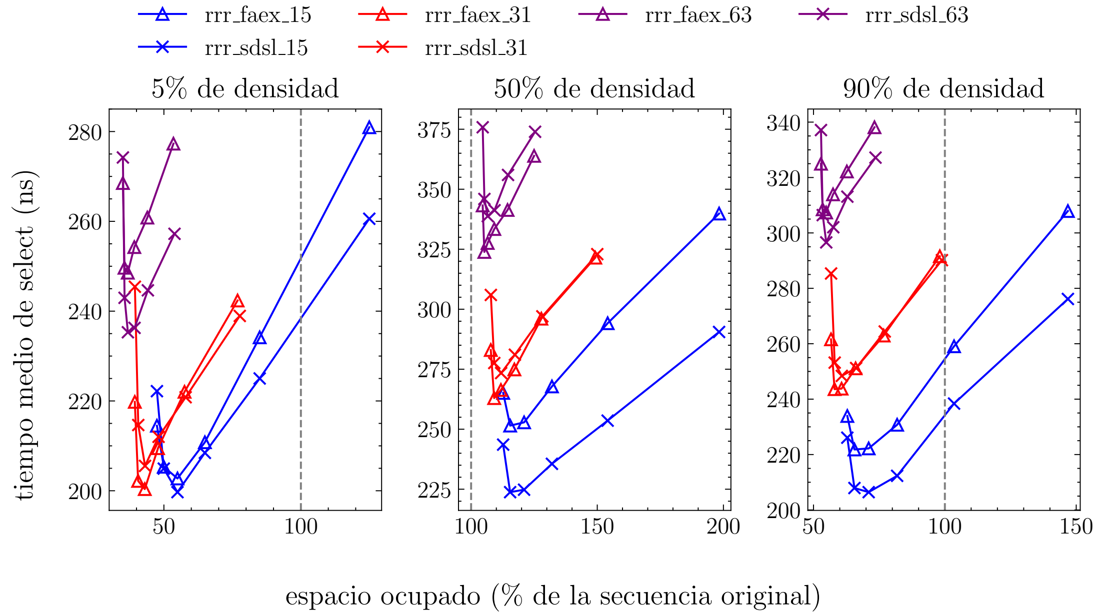

Figura 8.4: Balance entre el espacio ocupado y el tiempo de consulta para la operación de *select* sobre valores de ocurrencia aleatorios en los vectores de bits comprimidos con H_0. Se muestran experimentos para vectores de bits con un 5 %, 50 % y 90 % de densidad, que se corresponden con las gráficas de izquierda a derecha. La línea discontinua vertical indica el tamaño del vector de bits original

la que mejor funciona en términos de velocidad.

La operación de *select* muestra un comportamiento similar a lo mencionado en la sección anterior, exhibiendo resultados muy similares entre las alternativas de ambas librerías.

8.2.2 Wavelet Trees

Los experimentos relacionados con las estructuras de datos de tipo *wavelet tree* se pueden ver en las figuras 8.5 y 8.6, los cuales se corresponden con los corpus de texto *proteins* y *english* respectivamente.

Estos experimentos consisten en utilizar diferentes estructuras y configuraciones para la representación interna de los vectores de bits que utiliza el *wavelet tree*. En concreto, se utilizaron como estructuras de *wavelet tree* la solución *WaveletTree* propuesta por este proyecto y su análoga *wt_int* perteneciente a la SDSL. Como estructuras internas para representar los vectores de bits de los *wavelet trees* se utilizaron las descritas en la sección 8.2.1, empleando la misma convención de nombrado. Cabe destacar que únicamente se empleó la variante de vector de bits comprimido con H_0 con un tamaño de bloque $b = 63$, puesto que es la que presenta el comportamiento más interesante de entre las demás variantes de esta estructura, según los resultados descritos en la sección anterior.

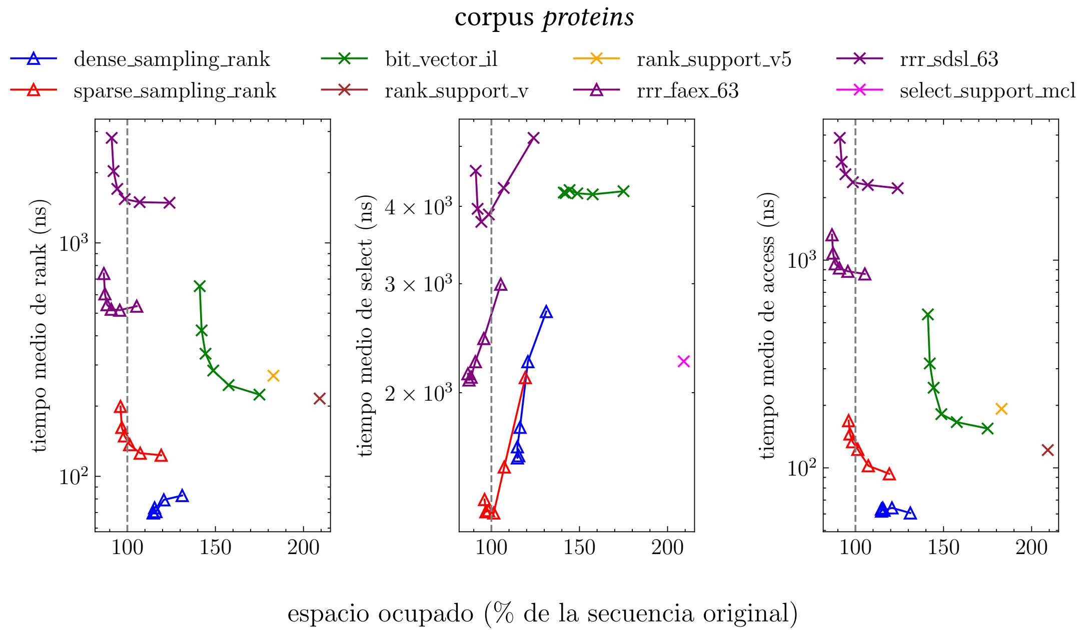

Figura 8.5: Balance entre el espacio ocupado y el tiempo de consulta para las operaciones de $rank$, $select$ y $access$ en la estructura *wavelet tree*, sobre el corpus de texto *proteins*, de izquierda a derecha. La línea discontinua vertical indica el tamaño óptimo en bits que ocuparía una secuencia de símbolos equiprobables de tamaño n, que es el valor $n \log_2 \sigma$, donde σ es el tamaño del alfabeto del corpus de texto

El tiempo de las operaciones de $rank$ fue medido realizando consultas de $rank$ sobre posiciones aleatorias y caracteres aleatorios del alfabeto del corpus de texto. De una forma similar, las consultas de $select$ fueron medidas utilizando caracteres aleatorios y valores de ocurrencia aleatorios comprendidos entre cero y el número total de ocurrencias del carácter que se utilizó para realizar la consulta. Así mismo, la operación de $access$ fue medida realizando consultas de $access$ sobre posiciones aleatorias del texto.

El espacio utilizado por las distintas estructuras se reporta como el porcentaje de espacio utilizado por la estructura respecto al tamaño óptimo en bits que ocuparía una secuencia de símbolos equiprobables de tamaño n, que se corresponde con el valor $n \log_2 \sigma$, donde σ es el tamaño del alfabeto del corpus de texto. Este valor también se puede interpretar como la entropía de orden cero para una secuencia de símbolos equiprobables. De esta manera, se consiguió establecer una línea base que sirviese para comparar el espacio utilizado por las diferentes soluciones.

Pese a que en los experimentos sobre los vectores de bits se puede apreciar un rendimiento muy similar entre las dos librerías, de los resultados mostrados para los dos corpus de texto en las figuras 8.5 y 8.6 se puede determinar que las estructuras propuestas por la librería desarrollada en este proyecto superan con creces a las estructuras de la librería SDSL, tanto

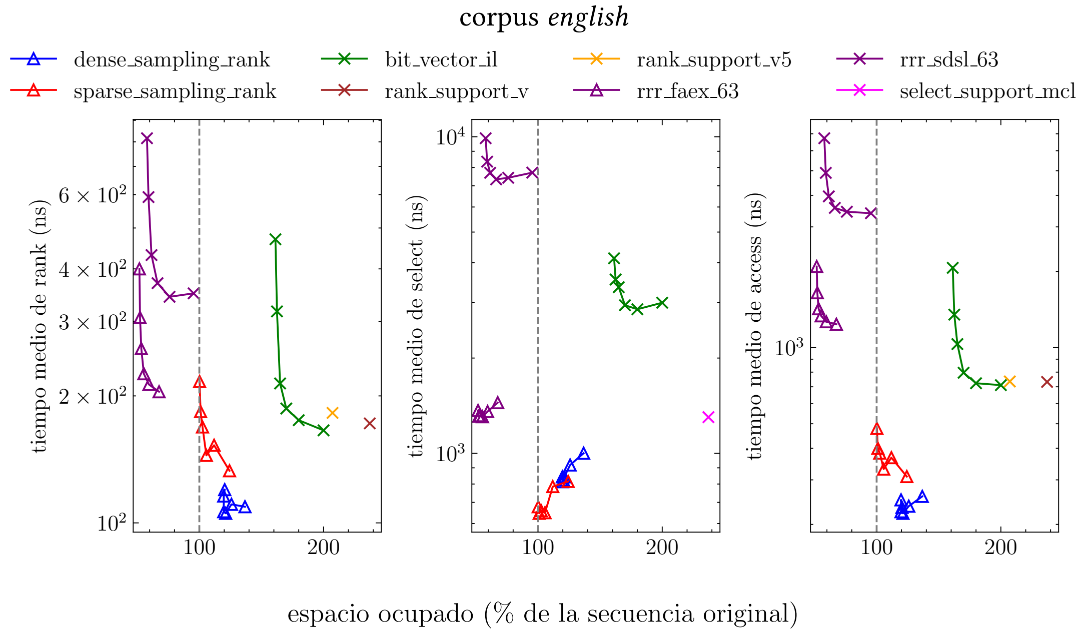

Figura 8.6: Balance entre el espacio ocupado y el tiempo de consulta para las operaciones de $rank$, *select* y $access$ en la estructura *wavelet tree*, sobre el corpus de texto *english*, de izquierda a derecha. La línea discontinua vertical indica el tamaño óptimo en bits que ocuparía una secuencia de símbolos equiprobables de tamaño n, que es el valor $n \log_2 \sigma$, donde σ es el tamaño del alfabeto del corpus de texto

en espacio como en tiempo, para todas las variantes de las estructuras de datos utilizadas.

La gran diferencia en tiempo que se puede apreciar entre la solución de la librería desarrollada en este proyecto y la solución de la librería SDSL se debe principalmente a que nuestra implementación es mucho más eficiente a nivel algorítmico, además de ajustarse mejor a los aspectos teóricos mostrados en la sección 2.6.

De la misma manera, la gran diferencia en espacio ocupado que existe, especialmente entre utilizar las estructuras *rank_support_v* y *select_support_mcl* de la librería SDSL y la estructura *dense_sampling_rank* propuesta por este proyecto, se debe también a una implementación mucho más eficiente y fiel a los aspectos teóricos mostrados en la sección 2.6 por parte de nuestra solución.

En último lugar, cabe destacar el comportamiento de la estructura interna *rrr_faex_63* en los experimentos sobre el corpus de texto *english*, de manera que se logra que la variante del *wavelet tree* comprimida con H_0 comprima el texto hasta un 50 % de su tamaño óptimo sin comprimir, soportando a su vez las operaciones de $rank$, *select* y $access$ de forma muy eficiente.

8.3 Comparativa cualitativa

Entre la librería desarrollada en este proyecto y la librería SDSL existen algunas diferencias que no se pueden cuantificar en los experimentos.

La principal ventaja y diferencia por parte de nuestra librería es la posibilidad de configurar, tanto los parámetros como el tipo de estructura de datos utilizada (esto último en el caso de los *wavelet trees*), en tiempo de ejecución. La librería SDSL únicamente permite configurar los parámetros en tiempo de compilación, mediante las *templates* del lenguaje de programación C++. La configuración en tiempo de ejecución tiene un gran número de ventajas, permitiendo que las estructuras se puedan adaptar a ciertas propiedades de los datos, además de que se elimina la necesidad de tener que recompilar el programa en caso de que se quieran cambiar los parámetros de las estructuras de datos. Además, en entornos en los que se requiera la interacción de un usuario para configurar los parámetros de las estructuras, permitir la configuración en tiempo de ejecución se convierte en una necesidad determinante. Por último, esta funcionalidad también es muy útil en el ámbito de las bases de datos, abriendo de esta manera un gran abanico de posibilidades en las que las estructuras de datos compactas pueden ser utilizadas.

También es notable destacar la facilidad de instalación de la librería desarrollada en este proyecto, puesto que mediante el gestor de paquetes *Cargo* y el repositorio de paquetes *crates.io*, se puede instalar la librería en cualquier proyecto de Rust con un sencillo comando. En cambio, la librería SDSL requiere de una instalación mucho más compleja y menos portable.

Por último, es importante mencionar que la librería desarrollada en este proyecto permite la serialización y deserialización de todas sus estructuras de datos en los formatos de intercambio de datos más utilizados, a diferencia de la librería SDSL, la cual únicamente soporta el formato binario. De esta manera se favorece en gran medida el uso de las estructuras de datos compactas en nuevos ámbitos en los que su uso no está tan explorado, como pueden ser los servicios web o las aplicaciones de escritorio.

Capítulo 9

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones obtenidas tras la realización de este proyecto, así como se proponen posibles líneas de trabajo futuro sobre la librería desarrollada.

9.1 Conclusiones

Una vez finalizado el desarrollo y la evaluación experimental del trabajo realizado, se puede concluir que se han cumplido los objetivos planteados en la sección 1.2, logrando una librería de estructuras de datos eficiente y competitiva con el estado del arte actual. Así mismo, como se ha podido ver en el capítulo 8, en varias situaciones se consigue un rendimiento significativamente mejor al de la librería SDSL.

Además, se consiguió proveer de nuevas funcionalidades no contempladas en el estado del arte, como puede ser la configuración en tiempo de ejecución de las estructuras de datos o la serialización y deserialización de las mismas a formatos de intercambio de datos, abriendo así un gran abanico de posibilidades para el uso de estructuras de datos compactas en muchos nuevos ámbitos de la informática en los que no son tan populares.

De los resultados obtenidos también se puede determinar que el haber empleado el lenguaje de programación Rust fue todo un acierto, puesto que es un lenguaje moderno que nos proporciona una gran ergonomía y ventajas en todo el ciclo de vida del desarrollo de software, desde el desarrollo hasta el mantenimiento y distribución. Además, se ha demostrado que este lenguaje puede ser utilizado perfectamente para el desarrollo de software científico que requiera de un gran rendimiento, por lo que su adopción por parte de la comunidad científica puede resultar muy positiva, proporcionando una alternativa a lenguajes como C++ o C.

Finalmente, es importante mencionar que con la realización de este proyecto se han conseguido poner en práctica muchos de los conceptos teóricos vistos en el grado, destacando principalmente las asignaturas de Algoritmos, Recuperación de la Información y Diseño Software, las cuales fueron fundamentales para lograr el éxito de este proyecto. Por último, con-

sidero que lo aprendido con este trabajo ha sido muy enriquecedor para mi futuro profesional y académico.

9.2 Trabajo futuro

Pese a que el alcance propuesto para este trabajo se ha cumplido satisfactoriamente, quedan abiertas muchas líneas de trabajo futuro que pueden ser muy interesantes tanto para esta librería como para el campo de las estructuras de datos compactas en su conjunto. Algunas de estas líneas de trabajo son:

- **Mejorar y optimizar las estructuras de datos** que se han desarrollado en este proyecto. Pese a que su rendimiento ya es muy competitivo, se considera que aún existe un cierto margen de optimización de las mismas, principalmente en tiempo de ejecución.
- **Implementar nuevas estructuras de datos más complejas** que no se han considerado para este proyecto. Las estructuras que se han implementado en esta librería son las consideradas como fundamentales, pero existen muchas otras estructuras de datos compactas más complejas que son muy interesantes y necesarias, como pueden ser los FM-index [8], los k^2*-trees* [4] o los *Suffix Trees* [62]. También puede ser interesante que las publicaciones de nuevos artículos científicos en el campo de las estructuras de datos compactas se implementen en esta librería de primera mano, de forma que se pueda mantener actualizada y al día con las últimas novedades, favoreciendo así la reproducibilidad y la reutilización de estas nuevas aportaciones por cualquier persona.
- **Emplear la librería desarrollada en otros entornos** como pueden ser los servicios web, aprovechando así el gran ecosistema web existente en el lenguaje Rust. Además, el uso del lenguaje Rust junto con esta librería puede ser muy útil para llevar a cabo proyectos en entornos de producción, yendo más allá de quedarse en una simple prueba de concepto y logrando así que las estructuras de datos compactas puedan ser utilizadas en la práctica por un mayor número de personas.

Apéndices

Apéndice A

Material adicional

A.1 Instalación y uso

En esta sección se verá cómo crear un nuevo proyecto en Rust y cómo incorporar la librería desarrollada en el mismo.

Primero, para crear un nuevo proyecto, se hará uso de la herramienta *Cargo*, versión 1.71.0, y se ejecutará el comando del código A.1.

```
cargo new --lib my_project
```

Código A.1: Comando para crear un nuevo proyecto en Rust

En este punto, tendremos en el directorio *my_project* un fichero *Cargo.toml* y un directorio *src*, teniendo nuestro proyecto inicializado.

A continuación, se puede incorporar la librería en dicho proyecto empleando el comando del código A.2. Esta facilidad es posible gracias a que la librería se encuentra publicada en el repositorio de paquetes de Rust, *crates.io*, bajo el nombre de *faex*[1]. De esta manera, se modificará el fichero *Cargo.toml* para añadir la dependencia de la librería y que nuestro nuevo proyecto pueda utilizarla, compilando la misma en el momento de ejecutar el comando *build* de *Cargo*.

```
cargo add faex
```

Código A.2: Comando para incorporar la librería en un proyecto en Rust

[1] https://crates.io/crates/faex

A.2 Integración Continua

```
name: Rust

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]

env:
  CARGO_TERM_COLOR: always

jobs:
  ci:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Update stable and set as default
        run: rustup update stable && rustup default stable
      - name: Install clippy
        run: rustup component add clippy
      - name: Install rustfmt
        run: rustup component add rustfmt
      - name: Check formatting
        run: cargo fmt --all -- --check
      - name: Check clippy
        run: cargo clippy --lib --no-deps -- -D warnings
      - name: Build
        run: cargo build --verbose
      - name: Run tests
        run: cargo test --verbose
```

Código A.3: Fichero de configuración de GitHub Actions para el *pipeline* de integración continua

A.3 Algoritmo de cálculo de los coeficientes binomiales

La implementación del algoritmo de cálculo de los coeficientes binomiales utilizando programación dinámica se puede ver en el código A.4, donde podemos observar que primero se inicializa la diagonal y la primera columna de la tabla conforme a lo establecido en la Ecuación (7.1), para posteriormente calcular el resto de los valores de la tabla de una forma iterativa, utilizando los resultados de iteraciones anteriores para calcular los valores de la iteración ac-

tual.

```
// N equals the number of bits of the machine's architecture
const N: usize = std::mem::size_of::<usize>() * 8;
static BINOMIALS: [[usize; N + 1]; N + 1] = get_binomial_table();

// Precompute binomial table at compile time
const fn get_binomial_table() -> [[usize; N + 1]; N + 1] {
    let mut binomials = [[0; N + 1]; N + 1];
    let mut i = 0;
    // Initialize diagonal
    while i <= N {
        binomials[i][i] = 1;
        i += 1;
    }
    // Initialize first column
    i = 0;
    while i <= N {
        binomials[i][0] = 1;
        i += 1;
    }
    let mut n = 1;
    let mut k = 1;
    while n <= N {
        while k < n {
            binomials[n][k] = binomials[n - 1][k - 1] + binomials[n
    - 1][k];
            k += 1;
        }
        n += 1;
        k = 1;
    }
    binomials
}
```

Código A.4: Algoritmo de cálculo de coeficientes binomiales mediante programación dinámica

A.4 Operación de select en la estructura *DenseSamplingRank*

La operación de *select* sobre la estructura de *rank* con dos niveles de directorio se puede realizar con una doble búsqueda binaria (según lo mencionado en la sección 2.4.1), sobre el vector de *rank* de primer nivel y, posteriormente, sobre el vector de *rank* de segundo nivel. Esta operación es similar en la estructura de *rank* con un nivel de directorio, pero únicamente se comentará el caso de dos niveles de directorio, puesto que es el más complejo.

Podemos ver el código de este algoritmo en el código A.5. En él se puede observar que primero se realiza una búsqueda binaria de tipo *lower bound* sobre el vector de $rank$ de primer nivel (es decir, sobre los superbloques). Este tipo de búsqueda binaria tiene la particularidad de que si el elemento no se encuentra en el vector, devuelve el índice del mayor elemento que sea menor que el elemento buscado. Para que este algoritmo se realizase de forma más eficiente, se hizo la modificación de añadir siempre al final del vector de superbloques el número total de 1s que hay en el vector de bits (es decir, el valor *total_rank*), de forma que nunca se intente buscar un elemento que sea más grande que el mayor elemento del vector. De esta manera, si el elemento no se encuentra en el vector, se devolverá, a lo sumo, el índice del penúltimo elemento del vector. Esto último nos asegura que en la variable *left_superblock* siempre se encuentre el índice del mayor elemento que cumpla que es **menor o igual** que el elemento buscado y que en la variable *right_superblock* siempre se encuentre el índice de un elemento que sea **estrictamente mayor**. Si no utilizásemos esta modificación si quisiésemos buscar un elemento que fuese mayor que el mayor elemento del vector, en la variable *right_superblock* podría localizarse un elemento que sea menor que el elemento buscado, lo cual nos obligaría a realizar una comprobación adicional y que la búsqueda fuese más lenta. Todo lo mencionado anteriormente se basa en los principios de la programación *branchless*, la cual es una de las técnicas más clásicas y utilizadas en programación para lograr un código mucho más rápido y eficiente.

Posteriormente a la primera búsqueda binaria, se realiza otra más sobre los bloques del vector de $rank$ de segundo nivel, de forma que se localice el bloque concreto de w bits en el que se encuentra el elemento buscado (el cual estará en el índice indicado por la variable *left _block*), de forma muy similar a la primera búsqueda binaria. Una vez localizado el bloque, se realiza una búsqueda secuencial sobre los w bits de la palabra encontrada, utilizando las operaciones aritméticas de desplazamiento y enmascaramiento a nivel de bit.

```
impl SelectSupport<BitVec> for DenseSamplingRank {
    #[inline]
    unsafe fn select(&self, data: &BitVec, rank: usize) ->
    Option<usize> {
        if rank > self.total_rank {
            return None;
        }
        let mut left_superblock = 0;
        let mut right_superblock = self.superblocks.len() - 1;
        while right_superblock - left_superblock > 1 {
            let mid_superblock = (left_superblock +
    right_superblock) / 2;
            let mid_rank =
    *self.superblocks.get_unchecked(mid_superblock);
```

```
            if mid_rank < rank {
                left_superblock = mid_superblock;
            } else {
                right_superblock = mid_superblock;
            }
        }
        let superblock_rank =
    *self.superblocks.get_unchecked(left_superblock);
        let remaining_rank = rank - superblock_rank;
        let raw_data = data.raw_data();
        let mut left_block = left_superblock * self.k;
        let mut right_block = min(left_block + self.k - 1,
    raw_data.len() - 1);
        while right_block - left_block > 1 {
            let mid_block = (left_block + right_block) / 2;
            let mid_rank = self.blocks.get_unchecked(mid_block);
            if mid_rank < remaining_rank {
                left_block = mid_block;
            } else {
                right_block = mid_block;
            }
        }
        let right_rank = self.blocks.get_unchecked(right_block);
        let target_block_index = if right_rank < remaining_rank {
            right_block
        } else {
            left_block
        };
        let mut local_rank =
    self.blocks.get_unchecked(target_block_index);
        let mut block = *raw_data.get_unchecked(target_block_index);
        let mut bit_index = 0;
        while local_rank < remaining_rank {
            if block & 0b1 == 0b1 {
                local_rank += 1;
            }
            block >>= 1;
            bit_index += 1;
        }
        Some(target_block_index * BitVec::CONTAINER_WIDTH +
    bit_index)
    }
}
```

Código A.5: Algoritmo de $select$ sobre la estructura *DenseSamplingRank*

La operación de $select_0$ se haría de forma muy similar a la operación de $select$, pero

contando el número de 0s en lugar de los 1s. La modificación es trivial, por lo que no se entrará en detalle en esto.

A.5 Pruebas unitarias

Un importante aspecto a destacar de la librería es que, gracias en gran parte a las pruebas generadas automáticamente por las pruebas parametrizadas, se dispone de un total de 3.428 pruebas unitarias, las cuales representan una cobertura del código del 76,13 %.

Estas pruebas unitarias están repartidas entre las diferentes estructuras de datos implementadas y, a su vez, entre varias configuraciones posibles de una misma estructura (esto es, por ejemplo, pruebas con distintos valores de k para el *struct DenseSamplingRank*, o con distintos valores de b y k para el *struct RRRBitVec*). En la tabla A.1 se puede ver desglosado el número de pruebas unitarias que corresponde a cada estructura. En dicha tabla, es destacable mencionar el elevado número de pruebas unitarias que se corresponden con la estructura *RRRBitVec*, lo que se debe al elevado número de combinaciones que se generan utilizando los dos parámetros que nos permite configurar esta estructura junto con la manera en la que se implementaron las pruebas parametrizadas (es decir, se generan pruebas para todas las combinaciones posibles de los parámetros b y k que se especifican).

Estructura de datos	**Número de pruebas unitarias**
BitVec	120
CompactIntVec	405
VariableSizeIntVec	159
SparseSamplingRank	330
DenseSamplingRank	322
RRRBitVec	2040
WaveletTree	52
Total	**3428**

Tabla A.1: Número de pruebas unitarias de cada una de las estructura de datos implementadas

Lista de acrónimos

ADN Ácido Desoxirribonucleico. 29

API Application Programming Interface. 41, 43, 44, 55

BSON Binary JSON. 64

BWT Burrows-Wheeler Transform. 29

CPU Central Processing Unit. 56

FM-index Full-text index in Minute space. 2, 29, 80

JSON JavaScript Object Notation. 39, 63, 88

LLVM Low Level Virtual Machine. 56

LTO Link Time Optimization. 56

MIT Massachusetts Institute of Technology. 68

Protobuf Protocol Buffers. 39, 63

SDSL Succinct Data Structures Library. 2, 13, 27, 29, 30, 40, 47, 59, 69, 71, 73–79

TDD Test-Driven Development. 31, 65, 68

UB Undefined Behavior. 30, 44

UML Unified Modeling Language. 41

UTF-8 Unicode Transformation Format 8-bit. 70

Glosario

benchmark Medida estandarizada para evaluar y comparar el rendimiento de sistemas informáticos o componentes de dichos sistemas. Permite tomar decisiones informadas y realizar comparaciones objetivas contra otros sistemas o componentes. 27, 34

bounds checking Es una técnica de programación que consiste en comprobar si un índice o una referencia a un elemento de una estructura de datos está dentro de los límites válidos definidos para dicha estructura. Realizar esta comprobación sirve para evitar errores como el desbordamiento de búferes o violaciones de segmento. 43, 57

branchless Un algoritmo se dice que es *branchless* cuando no contiene sentencias condicionales. Esto es, no se bifurca el flujo de ejecución en función de una condición. 85

on-the-fly Un algoritmo se dice que es *on-the-fly* cuando se calcula el resultado en tiempo de ejecución, sin tener que precalcular los resultados de antemano. 59

population count O simplemente *popcount*, es una operación que cuenta el número de bits activos (bits con valor 1) en una palabra binaria. 15

sprint Un sprint es un periodo de tiempo en el que se desarrolla un conjunto de funcionalidades de un producto en una metodología ágil. 31

standalone Un software standalone (también conocido como software independiente o autónomo) se refiere a un programa que está diseñado para funcionar por sí mismo, sin requerir la instalación adicional de otras librerías o componentes externos para su funcionamiento. 2

token Un token es una secuencia de caracteres que sirve como unidad léxica para un lenguaje de programación. Los tokens pueden ser palabras clave, identificadores, literales, operadores, separadores, etc. 66

Bibliografía

[1] G. Navarro, *Compact data structures: A practical approach.* Cambridge University Press, 2016.

[2] R. D. Konow Krause, "Compact data structures for information retrieval on natural languages," 2016.

[3] S. Gog, R. Konow, and G. Navarro, "Practical Compact Indexes for Top-k Document Retrieval," *ACM J. Exp. Algorithmics*, vol. 22, mar 2017. [En línea a 11 de septiembre de 2023]. Disponible en: https://doi.org/10.1145/3043958

[4] N. R. Brisaboa, S. Ladra, and G. Navarro, "k2-Trees for Compact Web Graph Representation," in *String Processing and Information Retrieval*, J. Karlgren, J. Tarhio, and H. Hyyrö, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 18–30.

[5] R. Konow, G. Navarro, C. Clarke, and A. López-Ortíz, "Inverted treaps," *ACM Transactions on Information Systems*, vol. 35, pp. 1–45, 01 2017.

[6] T. Gagie, G. Navarro, and N. Prezza, "Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space," *J. ACM*, vol. 67, no. 1, jan 2020. [En línea a 11 de septiembre de 2023]. Disponible en: https://doi.org/10.1145/3375890

[7] B. Freire, S. Ladra, and J. R. Paramá, "Memory-Efficient Assembly Using Flye," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 19, no. 6, pp. 3564–3577, 2022.

[8] P. Ferragina and G. Manzini, "Indexing Compressed Text," *J. ACM*, vol. 52, no. 4, p. 552–581, jul 2005. [En línea a 11 de septiembre de 2023]. Disponible en: https://doi.org/10.1145/1082036.1082039

[9] R. Grossi, "A quick tour on suffix arrays and compressed suffix arrays," *Theoretical Computer Science*, vol. 412, no. 27, pp. 2964–2973, 2011, combinatorics on Words

(WORDS 2009). [En línea a 11 de septiembre de 2023]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0304397510007371

[10] G. Navarro, "Wavelet trees for all," *Journal of Discrete Algorithms*, vol. 25, pp. 2–20, 2014, 23rd Annual Symposium on Combinatorial Pattern Matching. [En línea a 11 de septiembre de 2023]. Disponible en: https://www.sciencedirect.com/science/article/pii/S1570866713000610

[11] S. Klabnik and C. Nichols, *The Rust programming language.* No Starch Press, 2023.

[12] I. Plauska, A. Liutkevičius, and A. Janavičiūtė, "Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller," *Electronics*, vol. 12, no. 1, 2023. [En línea a 11 de septiembre de 2023]. Disponible en: https://www.mdpi.com/2079-9292/12/1/143

[13] Y. Lin, S. M. Blackburn, A. L. Hosking, and M. Norrish, "Rust as a Language for High Performance GC Implementation," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 89–98. [En línea a 11 de septiembre de 2023]. Disponible en: https://doi.org/10.1145/2926697.2926707

[14] "Stack Overflow Developer Survey 2023." [En línea a 11 de septiembre de 2023]. Disponible en: https://survey.stackoverflow.co/2023/

[15] "Deno: A secure runtime for JavaScript and TypeScript." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/denoland/deno

[16] "SurrealDB: A scalable, distributed, collaborative, document-graph database, for the realtime web." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/surrealdb/surrealdb

[17] "Firecracker: Secure and fast microVMs for serverless computing." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/firecracker-microvm/firecracker

[18] "Tauri: Build smaller, faster, and more secure desktop applications with a web frontend." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/tauri-apps/tauri

[19] "Meilisearch: A lightning-fast search engine that fits effortlessly into your apps, websites, and workflow." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/meilisearch/meilisearch

[20] "Bevy: A refreshingly simple data-driven game engine built in Rust ." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/bevyengine/bevy

[21] "Rust in Azure." [En línea a 11 de septiembre de 2023]. Disponible en: https://azure.microsoft.com/en-us/blog/microsoft-azure-security-evolution-embrace-secure-multitenancy-confidential-compute-and-rust/

[22] "Rust in AWS." [En línea a 11 de septiembre de 2023]. Disponible en: https://aws.amazon.com/blogs/opensource/sustainability-with-rust/

[23] "Rust adoption in Google." [En línea a 11 de septiembre de 2023]. Disponible en: https://opensource.googleblog.com/2023/06/rust-fact-vs-fiction-5-insights-from-googles-rust-journey-2022.html

[24] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[25] J. Freire, N. Fuhr, and A. Rauber, "Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041)," *Dagstuhl Reports*, vol. 6, no. 1, pp. 108–159, 2016. [En línea a 11 de septiembre de 2023]. Disponible en: http://drops.dagstuhl.de/opus/volltexte/2016/5817

[26] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, no. 7604, 2016.

[27] J. R. F. Cacho and K. Taghva, "The state of reproducible research in computer science," in *17th International Conference on Information Technology–New Generations (ITNG 2020)*. Springer, 2020, pp. 519–524.

[28] O. B. Amaral and K. Neves, "Reproducibility: expect less of the scientific paper," *Nature*, vol. 597, no. 7876, pp. 329–331, 2021.

[29] S. Gog, T. Beller, A. Moffat, and M. Petri, "From Theory to Practice: Plug and Play with Succinct Data Structures," in *Experimental Algorithms*, J. Gudmundsson and J. Katajainen, Eds. Cham: Springer International Publishing, 2014, pp. 326–337.

[30] O. JE, "Why scientists are turning to Rust," *Nature*, vol. 588, p. 185, 2020.

[31] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, 2021, pp. 597–616.

[32] C. E. Shannon, "A Mathematical Theory of Communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.

[33] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text compression*. Prentice-Hall, Inc., 1990.

[34] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[35] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[36] ——, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[37] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[38] T. Gagie and G. Navarro, "Guest Editorial: Special Issue on Compact Data Structures," *Algorithmica*, vol. 80, pp. 1983–1985, 2018.

[39] T. Chilimbi, M. Hill, and J. Larus, "Making pointer-based data structures cache conscious," *Computer*, vol. 33, no. 12, pp. 67–74, 2000.

[40] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," in *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA).* CTI Press and Ellinika Grammata Greece, 2005, pp. 27–38.

[41] M. Zhang and Y. Zhang, "Rank and select operations on a word," *Information Processing Letters*, vol. 172, p. 106148, 2021. [En línea a 11 de septiembre de 2023]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0020019021000636

[42] I. Advanced Micro Devices, "AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions," 2023. [En línea a 11 de septiembre de 2023]. Disponible en: https://www.amd.com/system/files/TechDocs/24594.pdf

[43] G. Jacobson, "Space-efficient static trees and graphs," in *30th annual symposium on foundations of computer science.* IEEE Computer Society, 1989, pp. 549–554.

[44] Clark, David, "Compact PAT trees," Ph.D. dissertation, University of Waterloo, 1997. [En línea a 11 de septiembre de 2023]. Disponible en: http://hdl.handle.net/10012/64

[45] J. I. Munro, "Tables," in *International Conference on Foundations of Software Technology and Theoretical Computer Science.* Springer, 1996, pp. 37–42.

[46] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 4, pp. 43–es, 2007.

[47] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *International Symposium on String Processing and Information Retrieval.* Springer, 2008, pp. 176–187.

[48] R. Pagh, "Low redundancy in static dictionaries with O (1) worst case lookup time," in *Automata, Languages and Programming: 26th International Colloquium, ICALP'99 Prague, Czech Republic, July 11–15, 1999 Proceedings 26.* Springer, 1999, pp. 595–604.

[49] G. Navarro and E. Providel, "Fast, small, simple rank/select on bitmaps," in *International Symposium on Experimental Algorithms.* Springer, 2012, pp. 295–306.

[50] S. Gog and M. Petri, "Optimized succinct data structures for massive data," *Software: Practice and Experience*, vol. 44, no. 11, pp. 1287–1314, 2014.

[51] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," 2003.

[52] S. Ladra, "Algorithms and compressed data structures for information retrieval," 2011.

[53] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, pp. 2–es, 2007.

[54] P. Ferragina and G. Navarro, "Pizza & chili corpus," 2007. [En línea a 11 de septiembre de 2023]. Disponible en: http://pizzachili.dcc.uchile.cl/

[55] P. Ferragina, R. González, G. Navarro, and R. Venturini, "Compressed text indexes: From theory to practice," *Journal of Experimental Algorithmics (JEA)*, vol. 13, pp. 1–12, 2009.

[56] "libcds: Compact Data Structures Library." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/fclaude/libcds

[57] "libcds2: Compact Data Structures Library 2." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/fclaude/libcds2

[58] S. Vigna, "Broadword implementation of rank/select queries," in *International Workshop on Experimental and Efficient Algorithms.* Springer, 2008, pp. 154–168.

[59] "Homepage of Sux: Succinct data structures." [En línea a 11 de septiembre de 2023]. Disponible en: https://sux.di.unimi.it/

[60] "Github repository of the Rust implementation of Sux." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/vigna/sux-rs/

[61] "Github repository of the Succinct Data Structure Library." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/simongog/sdsl-lite

[62] L. M. Russo, G. Navarro, and A. L. Oliveira, "Fully compressed suffix trees," *ACM transactions on algorithms (TALG)*, vol. 7, no. 4, pp. 1–34, 2011.

[63] "sucds: Collection of succinct data structures in Rust." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/kampersanda/sucds

[64] "crates.io: The Rust community's crate registry." [En línea a 11 de septiembre de 2023]. Disponible en: https://crates.io/

[65] J. Köster, "Rust-Bio: a fast and safe bioinformatics library," *Bioinformatics*, vol. 32, no. 3, pp. 444–446, 2016.

[66] "rust-bio: A bioinformatics library for the Rust programming language." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/rust-bio/rust-bio

[67] "Cargo: The Rust package manager." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com/rust-lang/cargo

[68] W. Bugden and A. Alahmar, "Rust: The Programming Language for Safety and Performance," 06 2022.

[69] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[70] D. Astels, *Test driven development: A practical guide.* Prentice Hall Professional Technical Reference, 2003.

[71] L. Lamport, "Latex: A Document Preparation System." [En línea a 11 de septiembre de 2023]. Disponible en: http://www.latex-project.org/

[72] "Git: The fast version control system." [En línea a 11 de septiembre de 2023]. Disponible en: https://git-scm.com/

[73] "GitHub: Where the world builds software." [En línea a 11 de septiembre de 2023]. Disponible en: https://github.com

[74] "Visual Studio Code." [En línea a 11 de septiembre de 2023]. Disponible en: https://code.visualstudio.com/

[75] "Draw.io." [En línea a 11 de septiembre de 2023]. Disponible en: https://www.draw.io/

[76] "Levels.fyi: Compare career levels across companies." [En línea a 11 de septiembre de 2023]. Disponible en: https://www.levels.fyi/

[77] "Indeed: Job Search." [En línea a 11 de septiembre de 2023]. Disponible en: https://www.indeed.com/

[78] R. C. Martin, *The clean coder: a code of conduct for professional programmers.* Pearson Education, 2011.

[79] A. Filazzola and C. Lortie, "A call for clean code to effectively communicate science," *Methods in Ecology and Evolution*, vol. 13, no. 10, pp. 2119–2128, 2022.

[80] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Pearson Deutschland GmbH, 1995.

[81] "Rust API Guidelines." [En línea a 11 de septiembre de 2023]. Disponible en: https://rust-lang.github.io/api-guidelines/

[82] "LLVM: The LLVM Compiler Infrastructure." [En línea a 11 de septiembre de 2023]. Disponible en: https://llvm.org/

[83] "Unsafe Rust Official Documentation." [En línea a 11 de septiembre de 2023]. Disponible en: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

[84] "Rust Derive Macros." [En línea a 11 de septiembre de 2023]. Disponible en: https://doc.rust-lang.org/reference/procedural-macros.html#derive-macros

[85] "MongoDB: The most popular database for modern apps." [En línea a 11 de septiembre de 2023]. Disponible en: https://www.mongodb.com/

[86] N. Tillmann and W. Schulte, "Parameterized unit tests," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 253–262, 2005.

[87] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 364–374.

[88] "MIT License." [En línea a 11 de septiembre de 2023]. Disponible en: https://opensource.org/licenses/MIT