Directly Addressable Codes: Implementando Operaciones de Modificación en Secuencias Comprimidas de Enteros

Facultad de Ciencias Empresariales
Universidad del Bío-Bío

José Benavente
jose.benavente2001@alumnos.ubiobio.cl

Profesores Guía
Rodrigo Torres-Avilés Luis Cabrera-Crot

Resumen

Los Directly Addressable Codes (DACs) proveen un esquema eficiente de codificación de longitud variable para secuencias de enteros con capacidades de acceso directo. Si bien los DACs logran excelentes ratios de compresión y acceso aleatorio rápido, son fundamentalmente estáticos, requiriendo reconstrucción completa ante cualquier modificación de la secuencia. En este trabajo presentamos una modificación de la implementación original de DACs que incorpora operaciones dinámicas directamente sobre la representación comprimida. Las operaciones de adición al final y eliminación del final se implementan en tiempo polilogarítmico respecto al valor máximo, sin desplazamiento de bits, mientras que las operaciones de inserción, eliminación y reemplazo en posiciones arbitrarias operan en tiempo $O(n/W)$, donde n es el número de elementos y W es el tamaño de palabra de la máquina. De manera crucial, nuestra modificación preserva todas las propiedades de espacio y tiempo de acceso de los DACs originales sin requerir reconstrucción completa. Esto significa que las ventajas inherentes de los DACs sobre sus competidores dinámicos se conservan completamente. Experimentos sobre cinco datasets lo confirman: los DACs modificados consumen 2–3× menos memoria y logran acceso aleatorio 2–11× más rápido que la estructura dinámica del estado del arte SPSI, como consecuencia directa de la representación DACs en sí. En operaciones dinámicas, la adición y eliminación al final superan adicionalmente a SPSI por uno o más órdenes de magnitud, mientras que la inserción, eliminación y reemplazo en posiciones arbitrarias son más lentos debido al desplazamiento a nivel de bits, representando el único trade-off introducido por nuestra modificación.

Keywords: Estructuras de datos compactas, Códigos de longitud variable, Operaciones dinámicas, Directly addressable codes, Secuencias compactas

Índice

1. Definición del Problema de Estudio y Oportunidad **5**

2. Estado del Arte **5**
2.1. Preliminares 5
2.1.1. Códigos de Longitud Variable Fundamentales 5
2.1.2. Codificación Vbyte 6
2.1.3. Operaciones Rank y Select 6
2.1.4. Técnicas de Acceso Directo Relacionadas 6
2.1.5. Aplicaciones que Requieren Dinamismo 8
2.2. Los Directly Addressable Codes 9
2.2.1. Construcción de los DACs 9
2.2.2. Algoritmo de Acceso 11
2.2.3. Tamaños de Chunk Óptimos 12
2.2.4. Complejidad Espacial y Temporal 12
2.3. SPSI: Sumas Parciales Buscables con Inserción 12
2.4. Comparación con Codificaciones Alternativas 13
2.5. Relación con el Aporte del Trabajo Propuesto 14

3. Hipótesis y Objetivos **14**
3.1. Hipótesis 14
3.2. Objetivo General 15
3.3. Objetivos Específicos 15

4. Artículo **15**

5. Conclusiones Generales y Recomendaciones **49**
5.1. Desarrollo de Algoritmos Dinámicos sobre la Estructura Comprimida 49
5.2. Optimización del Manejo de Memoria 49
5.3. Implementación de Estructuras Auxiliares 49
5.4. Análisis Teórico de Complejidad 50
5.5. Evaluación Experimental 50
5.6. Comparación con Soluciones Alternativas 50
5.7. Reflexiones Finales sobre el Proyecto de Investigación 51

Índice de figuras

1. Ejemplo de Wavelet Tree 8
2. Estructura DACs 10

Índice de cuadros

1. Muestras de sparse sampling con $h = 3$ 7

1. Definición del Problema de Estudio y Oportunidad

Los códigos de longitud variable son fundamentales en compresión de datos, asignando representaciones más cortas a elementos más frecuentes para reducir el espacio de almacenamiento. Sin embargo, presentan una limitación crítica: la imposibilidad de acceder directamente al i-ésimo elemento de una secuencia codificada sin decodificar elementos previos. Esta limitación es problemática en aplicaciones que requieren tanto compresión eficiente como acceso directo rápido.

Las soluciones tradicionales para proporcionar acceso directo a códigos de longitud variable, como sparse sampling, introducen un compromiso entre espacio y tiempo que degrada significativamente el rendimiento. Estas técnicas requieren muestreo regular y almacenamiento de punteros absolutos, lo que incrementa el uso de memoria y los tiempos de acceso.

Los Directly Addressable Codes (DACs) [1] resuelven este problema fundamental mediante una reorganización inteligente de datos codificados que mantiene las propiedades de compresión mientras habilita acceso directo eficiente a cualquier posición en tiempo $O(\lceil \log(M)/b \rceil)$ en el peor caso, donde M es el entero máximo de la secuencia y b es un parámetro de configuración.

Sin embargo, los DACs son estructuras esencialmente estáticas, lo que limita su aplicabilidad en escenarios modernos que requieren modificaciones dinámicas sobre datos comprimidos. Este trabajo propone una extensión de los DACs con operaciones dinámicas (DACs modificados), que incorpora operaciones eficientes de inserción, eliminación y modificación sin comprometer sus propiedades fundamentales de compresión y acceso directo.

2. Estado del Arte

2.1. Preliminares

A continuación se presentan los conceptos y estructuras fundamentales necesarios para comprender los DACs y el contexto de este trabajo.

2.1.1. Códigos de Longitud Variable Fundamentales

Diversas familias de códigos de longitud variable han sido propuestas para codificar secuencias de enteros, cada una con distintos compromisos entre razón de compresión, velocidad de acceso y soporte para operaciones dinámicas [1].

Codificación Huffman: Un enfoque consiste en asignar codewords más cortas a los valores más frecuentes. La codificación de Huffman [12] es el código óptimo de este tipo, logrando la longitud total mínima que es decodificable unívocamente. Sin embargo, cuando el conjunto de valores distintos es grande, almacenar la tabla de Huffman introduce una sobrecarga prohibitiva [1].

Códigos Gamma (γ) y Delta (δ): Cuando se asume que los enteros más pequeños son más frecuentes, pueden usarse códigos prefijo que explotan directamente la magnitud. Los códigos γ y δ de Elias [7] son ejemplos conocidos. Los códigos γ representan un entero positivo n utilizando $1 + 2\lfloor \log_2 n \rfloor$ bits, escribiendo $\lfloor \log_2 n \rfloor$ ceros seguidos de un 1 y luego la representación binaria de n sin el bit más significativo. Los códigos δ mejoran a los γ para

números grandes, codificando $1+\lfloor \log_2 n \rfloor$ en γ seguido de los $\lfloor \log_2 n \rfloor$ bits menos significativos de n. Estos códigos no requieren modelo almacenado y funcionan bien cuando la distribución está sesgada hacia valores pequeños.

Códigos Rice: Los códigos Rice [15] son una familia de códigos parametrizados por un valor r. Para codificar un entero n, se representa $\lfloor n/2^r \rfloor$ en unario, seguido de los r bits menos significativos de n en binario. Son especialmente eficientes cuando los datos siguen una distribución geométrica.

2.1.2. Codificación Vbyte

Introducida por Williams y Zobel [14], la codificación Vbyte divide la representación binaria de un entero en segmentos de tamaño fijo llamados **chunks**. Un *chunk* es una unidad básica de b bits de datos que, junto con un bit adicional de control, forma un bloque de $b+1$ bits. El bit más alto de cada chunk actúa como indicador de continuación: 0 marca el chunk más significativo (final), mientras que 1 indica que continúan más chunks [1].

Por ejemplo, codificando $x_i = 70$ con $b = 3$: la representación binaria 1000110_2 produce tres chunks 001, 000, 110 (de más a menos significativo), dando la forma codificada 0001 1000 1110. Aunque Vbyte sacrifica un bit por cada b bits respecto a una codificación óptima, es excepcionalmente rápido de decodificar [1], lo que lo convierte en la base de los DACs.

La codificación Vbyte es fundamental para los DACs porque proporciona la estructura de chunks necesaria para la reorganización por niveles. Mientras que en Vbyte tradicional los chunks de cada número se almacenan consecutivamente, los DACs aprovechan esta división natural para reorganizarlos por posición (primer chunk de todos los números, segundo chunk de todos los números, etc.), lo que permite acceso directo sin sacrificar las propiedades de compresión [1].

2.1.3. Operaciones Rank y Select

Las operaciones rank y select son primitivas fundamentales sobre bitmaps. Dado un bitmap B de longitud m, $\text{rank}_1(B, i)$ cuenta el número de bits 1 en $B[1..i]$, mientras que $\text{select}_1(B, j)$ retorna la posición del j-ésimo bit 1 en B. Estas operaciones pueden implementarse para ejecutarse en tiempo constante $O(1)$ usando estructuras auxiliares de tamaño $o(m)$ [3]. En los DACs, rank permite determinar la posición correspondiente de un elemento en el siguiente nivel de la jerarquía [1].

2.1.4. Técnicas de Acceso Directo Relacionadas

Sparse Sampling. Sparse sampling es una técnica clásica para proporcionar acceso directo a secuencias codificadas con códigos de longitud variable [1]. La idea consiste en muestrear regularmente la secuencia y almacenar punteros absolutos a elementos específicos, permitiendo acceso directo mediante decodificación parcial desde la muestra más cercana.

Consideremos la secuencia $[2, 5, 1, 8, 3, 6, 1, 4]$ codificada usando códigos γ. Al concatenar todos los códigos, obtenemos una secuencia codificada de 29 bits:

```
10|00101|1|0001000|011|00110|1|00100
```

donde cada elemento comienza en las posiciones de bit 0, 2, 7, 8, 15, 18, 23 y 24, respectivamente. Aplicando sparse sampling con $h = 3$, se almacenan muestras cada $h = 3$ elementos de la secuencia original:

Muestra	**Posición en secuencia**	**Puntero (bit)**
0	0	0
1	3	8
2	6	23

Cuadro 1: Muestras de sparse sampling con $h = 3$

Para acceder al elemento en posición i se identifica la muestra más cercana $\lfloor i/h \rfloor$, se utiliza su puntero para posicionarse en la secuencia codificada, y se decodifican secuencialmente $d = i - (\text{muestra} \times h)$ elementos. Esta técnica establece un compromiso fundamental entre espacio y tiempo que motiva la búsqueda de alternativas más eficientes como los DACs [1].

Elias-Fano. La representación Elias-Fano [18] está específicamente diseñada para secuencias de enteros estrictamente crecientes $y_1 < y_2 < \ldots < y_n$ que terminan en $u = y_n$ [1]. Esta representación utiliza $n \log(u/n) + O(n)$ bits y proporciona acceso en tiempo constante a cualquier y_i.

La técnica separa cada elemento y_i en sus $s = \lceil \log(u/n) \rceil$ bits inferiores y sus bits superiores restantes. Los bits inferiores se almacenan contiguamente, mientras que los bits superiores se representan mediante un bitmap donde se establecen bits en posiciones específicas.

La limitación fundamental de Elias-Fano es su restricción a secuencias estrictamente crecientes, lo que reduce significativamente su aplicabilidad general. Estas limitaciones motivaron el desarrollo de los DACs [1], que mantienen las propiedades de acceso directo eficiente y compresión espacial, pero operan sobre secuencias arbitrarias de enteros sin restricciones de orden o unicidad.

Wavelet Tree. El Wavelet Tree [8] es una estructura de datos versátil que representa una secuencia de símbolos y proporciona acceso directo eficiente a cualquier elemento. A diferencia de las técnicas anteriores, los Wavelet Trees no están restringidos a secuencias estrictamente crecientes y pueden manejar alfabetos arbitrarios.

Un Wavelet Tree para una secuencia $S[1, n]$ sobre un alfabeto $[1..\sigma]$ se construye como un árbol binario balanceado con σ hojas. En cada nodo interno se almacena un bitmap que codifica una partición recursiva del alfabeto. Para un nodo que maneja el rango de alfabeto $[a..b]$, el bitmap $B[1, n]$ se define como: $B[i] = 0$ si $S[i] \leq \lfloor (a + b)/2 \rfloor$, y $B[i] = 1$ en caso contrario.

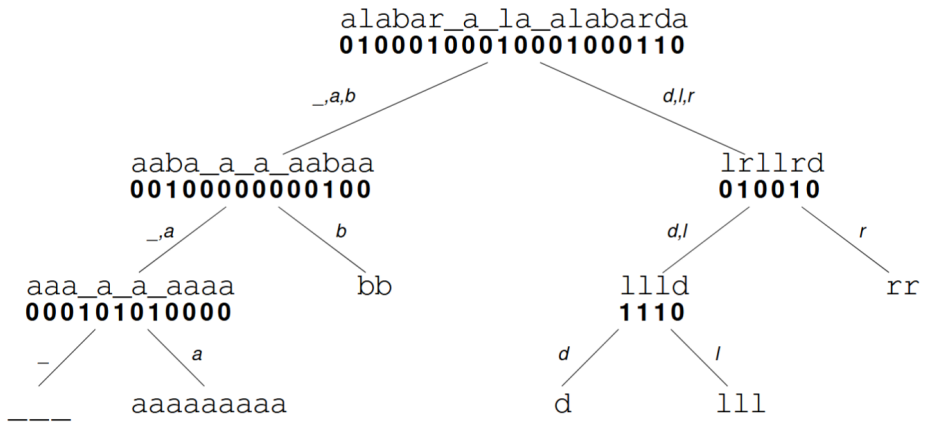

Figura 1: Wavelet Tree para la secuencia "alabar a la alabarda" [8]

Existe una conexión conceptual importante entre los Wavelet Trees y los DACs. Ambas estructuras abordan el problema de proporcionar acceso directo a códigos de longitud variable, pero desde enfoques diferentes. Mientras que los Wavelet Trees reorganizan la secuencia en una estructura de árbol basada en la partición recursiva del alfabeto, los DACs reorganizan los chunks de la codificación Vbyte para mantener la sincronización. Los DACs pueden verse como una especialización de los principios de los Wavelet Trees aplicada específicamente a la codificación Vbyte, ofreciendo una solución más simple y directa para secuencias de enteros, mientras que los Wavelet Trees proporcionan un marco más general para cualquier secuencia de símbolos y soportan una cantidad mayor de consultas.

2.1.5. Aplicaciones que Requieren Dinamismo

Los DACs han encontrado aplicación en numerosos contextos donde se requiere dinamismo, siendo particularmente relevantes en estructuras geoespaciales. Los DACs se utilizan como componente fundamental en la estructura k^2-raster, un esquema de compresión especializado para datos geoespaciales raster [4, 5], donde permiten realizar operaciones de álgebra de mapas directamente sobre los datos comprimidos sin descompresión completa [6, 16]. En estos contextos, las modificaciones dinámicas son esenciales para operaciones como actualización de mapas en tiempo real, análisis de cambios temporales en datos geoespaciales, y operaciones de álgebra de mapas que requieren modificar valores específicos sin reconstruir toda la estructura.

En el análisis de datos hiperespectrales, los DACs se han adaptado para codificación de enteros en estructuras k^2-raster [17], donde las modificaciones dinámicas son necesarias para algoritmos de procesamiento en tiempo real. Estos algoritmos requieren frecuentemente actualizar valores específicos durante el procesamiento, como corrección de valores anómalos, aplicación de filtros adaptativos, y actualización de clasificaciones en tiempo real.

Adicionalmente, como menciona el trabajo original [1], los DACs han demostrado relevancia en árboles de sufijos comprimidos, representación eficiente de gramáticas, indexación basada en Lempel-Ziv [19], y acceso directo a cadenas comprimidas por gramática. En todos

estos contextos, la capacidad de realizar modificaciones dinámicas ampliaría significativamente la aplicabilidad de estas estructuras.

2.2. Los Directly Addressable Codes

Los Directly Addressable Codes (DACs) [1] constituyen un esquema innovador de codificación de longitud variable para secuencias de enteros que resuelve una limitación fundamental de los códigos de longitud variable tradicionales: la imposibilidad de acceder directamente a cualquier elemento de la secuencia codificada sin decodificar elementos previos.

La innovación clave de los DACs [1] es una reorganización por niveles de los datos codificados con Vbyte. En lugar de almacenar todos los chunks de cada entero de forma consecutiva, los DACs agrupan los chunks por nivel: todos los primeros chunks juntos, todos los segundos chunks juntos, y así sucesivamente. Esta reorganización, combinada con operaciones rank eficientes sobre bitmaps, permite el acceso directo a cualquier elemento sin decodificación secuencial [1].

Formalmente, una estructura DACs $C = (A, B, p_A, p_B, b)$ consta de [1]:

- A: Array que almacena los chunks de datos de cada nivel, concatenados secuencialmente.
- B: Array de bitmap con un bit por chunk en A, indicando la continuación del chunk (0 = el valor termina en este nivel, 1 = continúa al siguiente nivel).
- p_A, p_B: Arrays de punteros que marcan los límites de nivel en A y B respectivamente.
- $b = \{b_0, b_1, \ldots, b_{L-1}\}$: Tamaños de chunk por nivel, donde L es el número de niveles.

2.2.1. Construcción de los DACs

La construcción de los DACs sigue un proceso sistemático que transforma una secuencia de enteros en una estructura jerárquica accesible directamente [1]. A continuación ilustramos este proceso a través de un ejemplo detallado.

Ejemplo 1 (Construcción Completa de DACs)**.** Para comprender la construcción completa de los DACs, consideremos la secuencia $X = \{25, 2, 70, 10\}$ con tamaño de chunk $b = 3$.

Paso 1: Codificación Vbyte Generalizada

Dada una secuencia de enteros $X = x_1, x_2, \ldots, x_n$, cada entero x_i se codifica usando Vbyte generalizado con chunks de b bits. La representación binaria de x_i se divide en bloques de b bits, almacenando cada bloque en un chunk de $b + 1$ bits. El bit adicional actúa como indicador: 0 para el chunk más significativo (indicando fin de codeword), 1 para los demás chunks.

Valor	Binario	Chunks	Vbyte
25	11001_2	11|001	0011 1001
2	10_2	010	0010
70	1000110_2	1|000|110	0001 1000 1110
10	1010_2	1|010	0001 1010

Paso 2: Separación de Datos y Metadatos

Con $b = 3$ bits por chunk, los chunks se reorganizan separando los datos de los metadatos en cada nivel k. Aquí, n_k denota la cantidad de valores de la secuencia que requieren más de k niveles para su representación. Cada nivel C_k se descompone en dos componentes:

1. **Array de datos** A_k: Almacena contiguamente los b bits de datos de cada chunk en el nivel k, requiriendo $b \cdot n_k$ bits totales.
2. **Bitmap** B_k: Contiene n_k bits donde $B_k[i] = 1$ si el i-ésimo codeword en el nivel k continúa en el nivel $k+1$, y $B_k[i] = 0$ si termina en el nivel k.

Los chunks se reorganizan así:

- **Nivel 0** ($n_0 = 4$): $A_0 = \{001, 010, 110, 010\}$, $B_0 = \{1, 0, 1, 1\}$
- **Nivel 1** ($n_1 = 3$): $A_1 = \{011, 000, 001\}$, $B_1 = \{0, 1, 0\}$
- **Nivel 2** ($n_2 = 1$): $A_2 = \{001\}$, $B_2 = \{0\}$

Paso 3: Estructuras de Rank y Punteros de Nivel

Para navegar eficientemente entre niveles, se construyen estructuras de rank sobre cada bitmap B_k. Estas estructuras permiten calcular $\text{rank}(B_k, i)$ en tiempo constante usando $O(n_k \log\log N / \log N)$ bits extra, donde N es la longitud total de la secuencia codificada.

Adicionalmente, los arrays concatenados A y B requieren punteros de nivel para delimitar las secciones correspondientes a cada nivel k:

- **Array** p_A: $p_A[k]$ indica la posición de inicio (en bits) del nivel k dentro del array de datos A. Dado que el nivel k almacena n_k chunks de b bits cada uno, se tiene $p_A[k+1] = p_A[k] + n_k \cdot b$.
- **Array** p_B: $p_B[k]$ indica la posición de inicio (en bits) del nivel k dentro del bitmap B. Dado que el nivel k almacena n_k bits de continuación, se tiene $p_B[k+1] = p_B[k] + n_k$.

En el ejemplo, con $n_0 = 4$, $n_1 = 3$, $n_2 = 1$ y $b = 3$:

- $p_A = \{0,\ 4 \times 3,\ 4 \times 3 + 3 \times 3,\ 4 \times 3 + 3 \times 3 + 1 \times 3\} = \{0, 12, 21, 24\}$
- $p_B = \{0,\ 4,\ 4 + 3,\ 4 + 3 + 1\} = \{0, 4, 7, 8\}$

La Figura 2 muestra la estructura DACs resultante, donde A y B son los arrays concatenados de datos y bitmaps respectivamente, con p_A y p_B marcando los límites de cada nivel.

$$A = \{001010110010011000001001\}$$
$$B = \{10110100\}$$
$$pA = \{0, 12, 21, 24\}$$
$$pB = \{0, 4, 7, 8\}$$

Figura 2: Estructura DACs

Paso 4: Acceso Directo al tercer elemento (valor 70):

Aplicando el algoritmo de acceso descrito en la Sección 2.2.2, para acceder al tercer elemento comenzamos en posición $i_0 = 2$:

- Extraemos $A[2 \times 3, 3] = 110$ y verificamos $B[2] = 1$ (continúa)
- Nueva posición: $i_1 = \text{rank}(B, 2) = 1$
- Extraemos $A[5 \times 3, 3] = 000$ y verificamos $B[5] = 1$ (continúa)
- Nueva posición: $i_2 = 0$
- Extraemos $A[7 \times 3, 3] = 001$ y verificamos $B[7] = 0$ (termina)

El resultado es $001000110_2 = 70$, obtenido sin decodificar elementos previos.

2.2.2. Algoritmo de Acceso

Para acceder al elemento i, los DACs comienzan en el nivel 0 y leen el chunk almacenado en la posición i de A, inicializando el valor resultado. Si el bit correspondiente $B[i] = 0$, el valor termina en este nivel y se retorna inmediatamente. En caso contrario, la siguiente posición se calcula como $i' = \text{rank}_1(B, i)$, que localiza el chunk del elemento en el siguiente nivel. El valor acumulado se desplaza b_ℓ bits (donde ℓ es el nivel actual) y se agrega el nuevo chunk. Este proceso se repite siguiendo los punteros guiados por rank a través de los niveles, hasta alcanzar un chunk terminal ($B[i] = 0$).

Algorithm 1 Acceso a elemento i en DACs

Require: Estructura DAC con L niveles (donde $L = \lceil \log M/b \rceil$ es el número de niveles determinado en la construcción), arrays concatenados A y B con punteros de nivel p_A y p_B, posición i (1-indexada)
Ensure: Valor x_i

```
pos ← i − 1 {convertir a índice 0-base}
resultado ← 0
multiplicador ← 1
for k = 0 to L − 1 do
    chunk ← bitread(A, p_A[k] + pos · b_k, b_k) {leer b_k bits del nivel k en A}
    bit_cont ← B[p_B[k] + pos] {leer bit de continuación del nivel k en B}
    resultado ← resultado + chunk × multiplicador
    if bit_cont = 0 then
        return resultado
    end if
    pos ← rank_1(B, p_B[k] + pos) − 1 {posición en el siguiente nivel}
    multiplicador ← multiplicador × 2^{b_k}
end for
return resultado
```

Este proceso requiere tiempo $O(\lceil \log M/b \rceil)$ en el peor caso, donde M es el valor máximo de la secuencia, ya que cada iteración requiere una operación rank en tiempo constante.

2.2.3. Tamaños de Chunk Óptimos

El trabajo original de los DACs provee un algoritmo de programación dinámica para calcular los tamaños de chunk óptimos $b_0, b_1, \ldots, b_{L-1}$ que minimizan el espacio total para un dataset dado. El algoritmo analiza la distribución de valores y determina cuántos bits asignar en cada nivel para minimizar $\sum_{i=0}^{L-1} n_i(b_i+1)$, donde n_i es el número de enteros que requieren al menos $i+1$ niveles. La elección de los tamaños de chunk impacta significativamente tanto la razón de compresión como el tiempo de acceso, creando un compromiso ajustable entre espacio y tiempo.

2.2.4. Complejidad Espacial y Temporal

Los DACs logran compresión explotando el hecho de que los valores más pequeños requieren menos niveles. En cada nivel i, la estructura almacena n_i chunks (donde n_i es el número de enteros que requieren al menos $i+1$ niveles), con cada chunk requiriendo b_i bits de datos más un bit de continuación en el bitmap B. Para muchas distribuciones, particularmente aquellas con muchos valores pequeños, el espacio total se aproxima a la entropía de orden cero de la secuencia. Las estructuras rank/select sobre B añaden espacio $o(|B|)$ mientras soportan operaciones en tiempo $O(1)$ [3].

El acceso a un elemento toma tiempo $O(\lceil \log M/b \rceil)$ en el peor caso [1]. La descompresión secuencial de la secuencia completa requiere tiempo $O(n \cdot \lceil \log M/b \rceil)$ en el peor caso, ya que cada uno de los n elementos puede requerir recorrer todos los $L = \lceil \log M/b \rceil$ niveles.

2.3. SPSI: Sumas Parciales Buscables con Inserción

Prezza [13] introdujo el problema de Sumas Parciales Buscables con Inserción (SPSI, por sus siglas en inglés) y proveyó una implementación práctica como parte de la biblioteca DYNAMIC. La estructura SPSI mantiene una secuencia $s_1, \ldots, s_m$ de enteros no negativos de k bits y soporta cuatro operaciones: `sum`(i), que computa $\sum_{j=1}^{i} s_j$; `search`(x), que encuentra el menor i tal que $\sum_{j=1}^{i} s_j > x$; `update`(i, δ), que modifica s_i mediante un delta con signo; e `insert`(i), que inserta un cero entre s_{i-1} y s_i.[1]

La estructura se implementa como un árbol B que almacena enteros en sus hojas y contadores de tamaño de subárbol y suma parcial en los nodos internos. El tamaño de hoja l siempre está acotado por $0{,}5 \log m \leq l \leq \log m$, y el fanout f del nodo se elige para maximizar la eficiencia de caché. Los enteros dentro de cada hoja se empaquetan contiguamente en arrays de palabras, con cada entero asignado al tamaño en bits del entero más grande de su hoja. Esta estrategia evita desperdiciar espacio en hojas parcialmente llenas, garantizando al mismo tiempo acceso rápido mediante operaciones a nivel de bits.

Prezza demuestra que esta estructura ocupa a lo sumo

$$2 \cdot m \left(\log \frac{M}{m} + \log \log m + O\left(\frac{\log M}{\log m} \right) \right)$$

[1] La base de código actual de DYNAMIC también incluye una operación de eliminación, incorporada después de la publicación original. Código fuente disponible en `https://github.com/xxsds/DYNAMIC`.

bits de espacio y soporta las cuatro operaciones en tiempo $O(\log m)$, donde $M = m + \sum_{i=1}^{m} s_i$. Arroyuelo et al. [2] extendieron posteriormente la biblioteca DYNAMIC para soportar relaciones d-arias dinámicas comprimidas, usando SPSI como bloque constructivo central, siendo esta implementación extendida la que se utiliza como línea base experimental en este trabajo.

Esta flexibilidad tiene el costo de un uso de memoria sustancialmente mayor que la secuencia codificada sola, ya que la estructura de árbol y la sobrecarga por nodo consumen significativamente más espacio. Los DACs [1] adoptan un enfoque diferente: en lugar de soportar operaciones dinámicas, se enfocan en habilitar acceso directo rápido a cualquier elemento de una secuencia codificada con Vbyte sin añadir espacio asintóticamente significativo. Este trabajo une ambos objetivos.

2.4. Comparación con Codificaciones Alternativas

Resulta ilustrativo contrastar los DACs con las dos alternativas más simples que superan, tanto en términos de espacio como en cuanto al costo de modificar la secuencia en una posición arbitraria i.

Códigos de Longitud Fija (CLF). Los CLF asignan $w_{\text{máx}} = \lceil \log_2(M+1) \rceil$ bits uniformemente a cada elemento, donde M es el valor máximo de la secuencia. El acceso directo es trivialmente $O(1)$: el i-ésimo elemento comienza en el bit $i \cdot w_{\text{máx}}$. Sin embargo, el consumo de espacio es $n \cdot \lceil \log_2(M+1) \rceil$ bits independientemente de la distribución real de los valores. Para secuencias con distribución sesgada como los arrays LCP, donde la mayoría de los valores son pequeños respecto al máximo, esto es altamente ineficiente: cada elemento paga el costo completo del outlier más grande. Modificar la secuencia en la posición i requiere desplazar los $n - i$ elementos posteriores, cada uno ocupando $\lceil \log_2 M \rceil$ bits, dando un costo a nivel de palabra de:

$$\text{cost}_{\text{CLF}}(i) = \frac{(n-i) \cdot \lceil \log_2 M \rceil}{W}$$

donde W es el tamaño de la palabra de la máquina. En el peor caso ($i = 0$) esto es $O(n)$.

Códigos de Longitud Variable (CLV/Vbyte). Los CLV recuperan la compresión asignando codewords más cortas a valores más pequeños, aproximándose a la entropía de orden cero H_0 de la secuencia. El consumo de espacio disminuye significativamente para distribuciones sesgadas, pero el acceso directo se pierde por completo: decodificar el i-ésimo elemento requiere recorrer los $i - 1$ codewords precedentes, con costo $O(n)$. El costo de modificación es similarmente $O(n)$ ya que los límites entre elementos no son fijos.

DACs. Los DACs abordan este compromiso directamente. La reorganización por niveles revela por qué la modificación es más barata que en los CLF en la práctica. Insertar en la posición i requiere desplazar el sufijo de cada nivel j de forma independiente. Dado que en el nivel j solo están presentes los elementos cuyo valor requiere al menos $j + 1$ chunks, el costo total a nivel de palabra es:

$$\text{cost}_{\text{DACs}}(i) = \sum_{j=0}^{L-1} \frac{\big(n_j - \text{rank}_1(B,\, p_B[j] + i)\big) \cdot b_j}{W}$$

donde n_j es el número de elementos en el nivel j, $\text{rank}_1(B,\, p_B[j] + i)$ cuenta los bits 1 en B hasta la posición $p_B[j] + i$ (es decir, el número de elementos antes de la posición i que

alcanzan el nivel j), y b_j es el tamaño de chunk en el nivel j. Nótese que $n_j \leq n$ para todo j, con $n_0 = n$; para distribuciones sesgadas donde la mayoría de los valores son pequeños, $n_j \ll n$ para $j \geq 1$, por lo que el término dominante es el desplazamiento del nivel 0 y las contribuciones de niveles superiores disminuyen rápidamente. Esto contrasta directamente con los CLF, donde cada desplazamiento tiene el costo completo de $n-i$ independientemente del nivel.

Los DACs modificados heredan íntegramente las garantías de espacio y acceso de la implementación original de DACs. Las extensiones dinámicas no añaden sobrecarga a la representación comprimida ni a las operaciones de lectura. El costo de modificación a nivel de palabra sigue la misma suma por nivel que los DACs anteriores, sin abandonar en ningún momento el dominio comprimido y sin el costo $O(|C|)$ de reconstrucción completa que los DACs estáticos requieren ante cualquier modificación.

2.5. Relación con el Aporte del Trabajo Propuesto

El análisis del trabajo relacionado revela que, aunque existen múltiples técnicas para proporcionar acceso directo a códigos de longitud variable, ninguna de ellas aborda específicamente el problema de las operaciones dinámicas sobre DACs. Las técnicas como sparse sampling, Elias-Fano y Wavelet Trees se enfocan en estructuras estáticas o requieren reconstrucción completa para modificaciones. Las aplicaciones identificadas en estructuras geoespaciales y análisis de datos hiperespectrales demuestran la necesidad práctica de operaciones dinámicas eficientes.

Este trabajo llena un vacío importante al proponer algoritmos específicos para operaciones dinámicas (inserción, eliminación, reemplazo) sobre DACs que mantienen las propiedades de compresión y acceso directo sin requerir decodificación completa. La contribución se distingue de los enfoques existentes al operar directamente sobre la estructura comprimida, ofreciendo una solución que combina la eficiencia espacial de los DACs con la flexibilidad operacional requerida por las aplicaciones modernas de estructuras de datos compactas.

3. Hipótesis y Objetivos

3.1. Hipótesis

Es posible extender los Directly Addressable Codes con operaciones dinámicas (adición, inserción, eliminación y modificación) manteniendo:

1. Complejidad temporal sub-lineal para las operaciones dinámicas.
2. Overhead de espacio comparable a los DACs estáticos.
3. Acceso directo eficiente a cualquier elemento.
4. Mejor rendimiento en comparación a reconstruir la estructura completa.

3.2. Objetivo General

Diseñar, implementar y evaluar algoritmos eficientes para operaciones dinámicas sobre DACs que mejoren significativamente el rendimiento respecto a la reconstrucción completa, manteniendo las propiedades de compresión y acceso directo.

3.3. Objetivos Específicos

1. Desarrollar algoritmos para adición, inserción, eliminación y modificación que operen directamente sobre la estructura DAC sin decodificación completa.
2. Optimizar el manejo de memoria durante las operaciones dinámicas para minimizar el overhead espacial.
3. Implementar estructuras auxiliares que aceleren las operaciones dinámicas frecuentes.
4. Analizar teóricamente la complejidad temporal y espacial de los algoritmos propuestos.
5. Evaluar experimentalmente el rendimiento con datasets reales y sintéticos.
6. Comparar con soluciones alternativas y el enfoque naive de reconstrucción.

4. Artículo

Dynamic Directly Addressable Codes: Implementing Modification Operations on Compressed Integer Sequences

José Benavente*, Rodrigo Torres-Avilés, Luis Cabrera-Crot

Department of Information Systems, University of the Bío-Bío, Concepción, Chile

Abstract

Directly Addressable Codes (DACs) provide an efficient variable-length encoding scheme for integer sequences with direct access capabilities. While DACs achieve excellent compression ratios and fast random access, they remain fundamentally static, requiring complete reconstruction for any modification of the sequence. In this paper we present a modification of the original DACs implementation that incorporates dynamic operations directly on the compressed representation. Append and pop are implemented in polylogarithmic time with respect to the maximum value, with no bit shifting, while insert, delete, and replace operate in $O(n/W)$ time, where n is the number of elements and W is the machine word size. Crucially, our modification preserves all space and access time properties of the original DACs without requiring full reconstruction. This means that the inherent advantages of DACs over dynamic competitors are fully retained. Experiments on five datasets confirm this: the modified DACs consume 2–3× less memory and achieve 2–11× faster random access than the state-of-the-art dynamic structure SPSI, as a direct consequence of the DACs representation itself. In dynamic operations, append and pop further outperform SPSI on datasets with few levels, while SPSI achieves lower append times on multi-level LCP datasets. Insert, delete, and replace are slower due to bit-level shifting, representing the only trade-off introduced by our modification.

Keywords: Compact data structures, Variable-length codes, Dynamic

*Corresponding author

Email address: jose.benavente2001@alumnos.ubiobio.cl (José Benavente)

operations, Directly addressable codes, Compressed sequences

1. Introduction

Variable-length coding is a fundamental approach in data compression, assigning shorter codewords to smaller values and longer ones to larger values, thereby reducing storage when the sequence contains integers of widely varying magnitudes [7]. Several variable-length codes have been proposed for integer sequences, including Elias γ- and δ-codes [7], Huffman codes [12], and schemes based on wavelet trees [8] and dense sampling [11], each offering different trade-offs between compression ratio and access time. However, when values are uniform in size, fixed-length codes are more space-efficient, as variable-length schemes introduce overhead from chunk boundaries, continuation bits, and auxiliary structures that outweigh any potential gain [1]. Therefore, a critical limitation generally arises: accessing the i-th element in an encoded sequence typically requires decoding all previous elements [1]. This sequential access constraint is particularly problematic in applications requiring both efficient compression and fast random access, such as compressed suffix trees [9], inverted indexes [10], and geospatial data structures [5].

Directly Addressable Codes (DACs) [1] build upon prior approaches to this problem, such as Huffman codes [12], which already provided partial solutions to the trade-off between compression and direct access. Rather than resolving this trade-off entirely, DACs offer a highly effective application of the level-based reorganization principle: by restructuring variable-length codewords into a hierarchical representation using b bits per level, DACs maintain compression properties while enabling direct access to any position in $O(\lceil \log M/b \rceil)$ worst-case time, where M is the maximum integer in the sequence and b is a tunable parameter that controls the trade-off between compression ratio and access speed [1].

Despite their advantages, DACs is essentially a static structure. Any modification such as inserting, deleting, or replacing an element requires complete reconstruction of the representation: decompressing the entire structure, performing the modification on the uncompressed data, and recompressing. This reconstruction cost becomes particularly problematic in applications where compressed data must also support dynamic modifications. Prezza [13] identifies several scenarios in string manipulation, including dynamic FM-indexes

that require insertions into compressed representations to support online construction of the Burrows-Wheeler transform, and entropy- and run-length compressed strings that must support modifications during indexing tasks such as LZ77 factorization. In compressed geospatial data structures such as k^2-raster [5], where DACs serve as a fundamental component, modifications to the underlying integer sequences are likewise required to support map algebra and temporal analysis operations [4, 6]. In all of these settings, the inability to modify the compressed representation directly forces a costly decompress-modify-recompress cycle that undermines the practical utility of the encoding.

In this paper, we present a modification of the original DACs implementation that incorporates dynamic operations such as appending, inserting, deleting, and replacing directly on the compressed representation without requiring full decompression and recompression. The append and pop procedures run in $O(L)$ time with no bit shifting, where $L = \lceil \log M/b \rceil$ is the number of levels. The insert, delete, and replace procedures run in $O(n/W)$ time where W is the machine word size, and an explicit rebuild restores optimal compression in $O(n \cdot L + M + \log^2 M)$ time. All operations use $O(|C|)$ space, where C is the DACs structure, n is the number of elements, L is the number of levels, and M is the maximum value.

Our experimental evaluation compares the modified DACs against the state-of-the-art dynamic compressed structure SPSI [2] (Searchable Partial Sums with Insert), a B-tree-based structure that supports dynamic operations on compressed integer sequences [13], and against the original static DACs implementation as a baseline for compression ratio and access time. We evaluate dynamic operations (append, insert, delete, replace, pop) and static operations (random access and full decompression) across diverse datasets including Longest Common Prefix (LCP) arrays and raw data.

When compared to SPSI, the results reveal a clear trade-off. Since our modification preserves the space and access time properties of the original DACs, the inherent compactness of the DAC representation means the modified DACs consume up to 3× less memory than SPSI across tested datasets and maintain consistently faster random access times. In dynamic operations, append and pop outperform SPSI on datasets encoded with few levels, such as raw byte sequences, where tail operations require no bit shifting and no tree traversal. On multi-level LCP datasets, however, SPSI achieves lower append times due to its more compact internal representation for skewed distributions. SPSI also affords faster execution on arbitrary insert and delete

operations due to DACs incurring $O(n/W)$ bit-shifting costs across all level arrays. DACs are therefore best suited for workloads where space efficiency and query performance are the primary concern, and where modifications are dominated by appends, pops and replacements rather than arbitrary insertions and deletions.

Regarding static operations, modified DACs and the original DACs achieve equivalent results across all tested datasets, confirming that the incorporation of dynamic capabilities introduces no overhead to query performance or space usage.

The paper is organised as follows. Section 2 reviews the static DACs structure. Section 3 introduces our dynamic modifications and their theoretical analysis. Section 4 presents the experimental evaluation. Section 5 concludes the paper with final remarks and directions for future work.

2. Conceptual Framework

To understand our dynamic operations implementation, we first review the static DACs structure and its fundamental properties.

2.1. Variable-Length Encoding

Several families of variable-length codes have been proposed for encoding sequences of integers, each with different trade-offs between compression ratio, access speed, and support for dynamic operations [1].

2.1.1. Huffman Coding

One approach is to assign shorter codewords to more frequent values. Huffman coding [12] is the optimal such code, achieving the minimum total encoded length that is uniquely decodable. However, when the set of distinct values is large, storing the Huffman table introduces prohibitive overhead [1].

2.1.2. Elias and Rice Codes

When smaller integers are assumed to be more frequent, one can use prefix codes that directly exploit magnitude. Well-known examples include Elias γ- and δ-codes [7], which represent $\lfloor \log_2 x_i \rfloor$ in unary or recursively compressed form before encoding the remaining bits, and Rice codes, which split x_i at a fixed radix r, representing $\lfloor x_i/2^r \rfloor$ in unary followed by the lowest r bits verbatim [1]. These codes require no stored model and perform well when the integer distribution is skewed toward small values.

2.1.3. Vbyte Coding

Introduced by Williams and Zobel [14], this code splits the $\lfloor \log_2 x_i \rfloor + 1$ bits of x_i into blocks of b bits, storing each block as a chunk of $b+1$ bits. The highest bit of each chunk is a continuation flag: 0 marks the most significant (final) chunk, while 1 indicates that more chunks follow [1]. For example, encoding $x_i = 70$ with $b = 3$: the binary representation 1000110_2 produces three chunks 001, 000, 110 (most to least significant), yielding the encoded form 0001 1000 1110. Although Vbyte loses one bit per b bits compared to an optimal encoding, it is exceptionally fast to decode [1], which makes it the foundation of DACs.

2.1.4. Searchable Partial Sums with Insert (SPSI)

Prezza [13] introduced the Searchable Partial Sums with Insert (SPSI) problem and provided a practical implementation as part of the DYNAMIC library. The SPSI structure maintains a sequence $s_1, \ldots, s_m$ of non-negative k-bit integers and supports four operations: `sum`(i), which computes $\sum_{j=1}^{i} s_j$; `search`(x), which finds the smallest i such that $\sum_{j=1}^{i} s_j > x$; `update`(i, δ), which modifies s_i by a signed delta; and `insert`(i), which inserts a zero between s_{i-1} and s_i.[1]

The structure is implemented as a B-tree storing integers in its leaves and subtree size and partial sum counters in internal nodes. The leaf size l is always bounded by $0.5 \log m \leq l \leq \log m$, and the node fanout f is chosen to maximise cache efficiency. Integers within each leaf are packed contiguously in word arrays, with each integer assigned the bit-size of the largest integer in its leaf. This strategy avoids wasting space for partially filled leaves while still guaranteeing fast access through bitwise operations.

Prezza proves that this structure takes at most

$$2 \cdot m \left(\log \frac{M}{m} + \log \log m + O\left(\frac{\log M}{\log m} \right) \right)$$

bits of space and supports all four operations in $O(\log m)$ time, where $M = m + \sum_{i=1}^{m} s_i$. Arroyuelo et al. [2] later extended the DYNAMIC library to support dynamic compressed d-ary relations, using SPSI as a core building

[1] The current DYNAMIC codebase also includes a delete operation, which was added after the original publication. Source code available at `https://github.com/xxsds/DYNAMIC`.

block, and it is this extended implementation that we use as our experimental baseline.

This flexibility comes at the cost of substantially higher memory usage than the encoded sequence alone, as the tree structure and per-node overhead consume significantly more space. DACs [1] take a different approach: rather than supporting dynamic operations, they focus on enabling fast direct access to any element of a Vbyte-encoded sequence without adding asymptotically any extra space. Our work bridges these two goals.

2.2. The DACs Structure

The key innovation of DACs [1] is a level-based reorganization of Vbyte-encoded data. Instead of storing all chunks of each integer consecutively, DACs group chunks by level: all first chunks together, all second chunks together, and so on. This reorganization, combined with efficient rank operations on bitmaps, enables direct access to any element without sequential decoding [1].

Formally, a DACs structure $C = (A, B, p_A, p_B, b)$ consists of [1]:

- A: Array storing data chunks at each level, concatenated sequentially.
- B: Bitmap array with one bit per chunk in A, indicating chunk continuation (0 = value terminates at this level, 1 = continues to next level).
- p_A, p_B: Pointer arrays marking level boundaries in A and B respectively.
- $b = \{b_0, b_1, \ldots, b_{L-1}\}$: Chunk sizes per level, where L is the number of levels.

2.3. Rank Operations

DACs rely on the *rank* operation on bitmaps to navigate between levels [1]. Given a bitmap B and position i, $\text{rank}_1(B, i)$ returns the number of 1-bits in $B[0..i]$. This operation can be supported in $O(1)$ time using $o(|B|)$ auxiliary space [3]. The rank operation allows DACs to determine the position of a value's next chunk in the subsequent level [1].

2.4. Access Operation

To access element i, DACs start at level 0 and read the chunk stored at position i in A, initialising the result value. If the corresponding bit $B[i] = 0$, the value terminates at this level and is returned immediately. Otherwise, the next position is computed as $i' = \text{rank}_1(B, i)$, which locates the element's chunk in the next level. The accumulated value is then shifted by b_ℓ bits (where ℓ is the current level) and the new chunk is added. This process repeats, following the rank-guided pointers across levels, until a terminating chunk ($B[i] = 0$) is reached.

This process requires $O(\lceil \log M/b \rceil)$ time in the worst case, where M is the maximum value in the sequence.

2.5. Optimal Chunk Sizes

The original DACs work provides a dynamic programming algorithm to compute optimal chunk sizes $b_0, b_1, \ldots, b_{L-1}$ that minimise total space for a given dataset. The algorithm analyses the distribution of values and determines how many bits to allocate at each level to minimise $\sum_{i=0}^{L-1} n_i(b_i + 1)$, where n_i is the number of integers requiring at least $i + 1$ levels. The choice of chunk sizes significantly impacts both compression ratio and access time, creating a tunable space-time trade-off.

2.6. Space and Time Complexity

DACs achieve compression by exploiting the fact that smaller values require fewer levels. At each level i, the structure stores n_i chunks (where n_i is the number of integers requiring at least $i + 1$ levels), with each chunk requiring b_i data bits plus one continuation bit in the bitmap B. For many distributions, particularly those with many small values, the total space approaches the zero-order entropy of the sequence. The rank/select structures on B add $o(|B|)$ space overhead while supporting operations in $O(1)$ time [3].

Accessing an element takes $O(\lceil \log M/b \rceil)$ time in the worst case, where b represents the chunk size parameter and M is the maximum value [1]. Sequential decompression of the entire sequence requires $O(n \cdot \lceil \log M/b \rceil)$ time in the worst case, as each of the n elements may require traversing all $L = \lceil \log M/b \rceil$ levels.

2.7. Comparison with Alternative Encodings

It is instructive to contrast DACs against the two simpler alternatives they supersede, both in terms of space and in terms of the cost of modifying the sequence at an arbitrary position i.

Fixed-Length Codes (FLC) allocate $w_{\max} = \lceil \log_2(M+1) \rceil$ bits uniformly to every element, where M is the maximum value in the sequence. Direct access is trivially $O(1)$: the i-th element begins at bit $i \cdot w_{\max}$. However, space consumption is $n \cdot \lceil \log_2(M+1) \rceil$ bits regardless of the actual distribution of values. For sequences with a skewed distribution like LCP arrays, where most values are small relative to the maximum, this is highly wasteful. Every element pays the full cost of the largest outlier. Modifying the sequence at position i requires shifting all $n - i$ subsequent elements, each occupying $\lceil \log_2 M \rceil$ bits, giving a word-level cost of:

$$\text{cost}_{\text{FLC}}(i) = \frac{(n-i) \cdot \lceil \log_2 M \rceil}{w}$$

where w is the machine word size. In the worst case ($i = 0$) this is $O(n)$.

Variable-Length Codes (VLC/Vbyte) recover compression by assigning shorter codewords to smaller values, approaching the zero-order entropy H_0 of the sequence. Space consumption drops significantly for skewed distributions, but direct access is lost entirely: decoding the i-th element requires scanning all $i - 1$ preceding codewords, yielding $O(n)$ access. Modification cost is similarly $O(n)$ since element boundaries are not fixed.

DACs address this trade-off directly. The level-based reorganisation reveals why modification is cheaper than FLC in practice. Inserting at position i requires shifting the suffix of each level j independently. Since only elements whose value requires at least $j + 1$ chunks are present at level j, the total word-level cost is:

$$\text{cost}_{\text{DACs}}(i) = \sum_{j=0}^{L-1} \frac{\big(n_j - \text{rank}_1(B,\, p_B[j] + i)\big) \cdot b_j}{W}$$

where n_j is the number of elements at level j, $\text{rank}_1(B,\, p_B[j] + i)$ counts the 1-bits in B up to position $p_B[j] + i$ (i.e., the number of elements before position i that reach level j), and b_j is the chunk size at level j. Note that $n_j \leq n$ for all j, with $n_0 = n$; for skewed distributions where most values are small, $n_j \ll n$ for $j \geq 1$, so the dominant term is the level-0 shift and higher-level contributions diminish rapidly. This stands in direct contrast to FLC, where every shift carries the full $n - i$ cost regardless of level.

3. Dynamic Operations on DACs

We now present our algorithms for dynamic modifications to the DACs structure. Our approach manipulates the compressed representation directly through bit-level operations at each level independently, never reconstructing the integer sequence. Our modified DACs inherits the space and access guarantees of the original DACs implementation entirely. The dynamic extensions add no overhead to the compressed representation nor to read operations. The word-level modification cost follows the same per-level summation as the static DACs cost presented in Section 2, without ever leaving the compressed domain, and without the $O(|C|)$ full rebuild that static DACs require for any modification.

3.1. Structural Representation

The original DACs implementation [1] stores all level data in a single contiguous bit array A, with auxiliary pointer arrays p_A and p_B marking the starting positions of each level within A and the shared bitmap B respectively. This layout is well-suited for static access but makes in-place modification impractical: inserting or deleting a chunk at any level would require shifting all subsequent data across every remaining level in the same array.

Our implementation modifies this by decomposing the monolithic arrays into independent per-level structures. Formally, our modified DACs structure C consists of:

- $A_0, A_1, \ldots, A_{L-1}$: independent bit arrays, one per level, where A_i stores exclusively the data chunks of level i.
- $B_0, B_1, \ldots, B_{L-2}$: independent bitmaps, one per level, where B_i stores the continuation bits of level i together with its own rank support structure; the last level carries no bitmap since all elements that reach it terminate there by definition.
- $b = \{b_0, b_1, \ldots, b_{L-1}\}$: chunk sizes per level, unchanged from the original DACs.

This decomposition means that an insertion or deletion at level i only requires shifting bits within A_i and updating B_i, leaving all other levels entirely untouched.

From a theoretical standpoint, this decomposition introduces no additional space cost over the original DACs. The monolithic arrays A and B in the original implementation are navigated through the pointer arrays p_A and p_B, which mark the boundary of each level. Our per-level decomposition replaces these boundary pointers with independent array references (one per level) achieving the same addressing at identical cost. The space and access guarantees of the original DACs are therefore fully preserved. For a concrete illustration of both representations, see Example 1.

Example 1. *Consider the sequence* $S = \langle 5,\ 20,\ 100,\ 3,\ 60,\ 80\rangle$ *encoded as a 3-level DACs structure with uniform chunk sizes* $b_0 = b_1 = b_2 = 3$ *bits (base-8 per level). Each value* v *is decomposed into 3-bit chunks by expressing it in base 8:* $v = \delta_0 + \delta_1 \cdot 8 + \delta_2 \cdot 64$*, where* $\delta_i \in [0, 7]$ *is the chunk written to level* i*. Table 1 summarises the decomposition in binary.*

Table 1: Chunk decomposition of $S = \langle 5, 20, 100, 3, 60, 80\rangle$ under $b_0 = b_1 = b_2 = 3$ bits. A dash indicates the element does not reach that level.

Elem.	Value	Term. level	δ_0	δ_1	δ_2
s_1	5	0	101	—	—
s_2	20	1	100	010	—
s_3	100	2	100	100	001
s_4	3	0	011	—	—
s_5	60	1	100	111	—
s_6	80	2	000	010	001

Original DACs representation. *In the original DACs [1], all chunks are concatenated into a single bit array* A *and all continuation bits into a single bitmap* B*, with pointer arrays* p_A *and* p_B *marking level boundaries:*

$$A = \overbrace{\underbrace{101}_{s_1}\ \underbrace{100}_{s_2}\ \underbrace{100}_{s_3}\ \underbrace{011}_{s_4}\ \underbrace{100}_{s_5}\ \underbrace{000}_{s_6}}^{\text{level 0}}\ \overbrace{\underbrace{010}_{s_2}\ \underbrace{100}_{s_3}\ \underbrace{111}_{s_5}\ \underbrace{010}_{s_6}}^{\text{level 1}}\ \overbrace{\underbrace{001}_{s_3}\ \underbrace{001}_{s_6}}^{\text{level 2}}$$

$$B = \overbrace{\underbrace{0}_{s_1}\ \underbrace{1}_{s_2}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_4}\ \underbrace{1}_{s_5}\ \underbrace{1}_{s_6}}^{\text{level 0}}\ \overbrace{\underbrace{0}_{s_2}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_5}\ \underbrace{1}_{s_6}}^{\text{level 1}}$$

Modifying any chunk in this layout requires shifting all subsequent data across every remaining level in the same monolithic array.

Our modified representation. *Each A_i and B_i is a fully independent array. A modification at level i shifts bits only within A_i and updates only B_i, leaving every other level entirely untouched.*

Level 0 ($b_0 = 3$ bits per chunk, $n_0 = 6$ chunks):

$$A_0 = \underbrace{\texttt{101}}_{s_1}\ \underbrace{\texttt{100}}_{s_2}\ \underbrace{\texttt{100}}_{s_3}\ \underbrace{\texttt{011}}_{s_4}\ \underbrace{\texttt{100}}_{s_5}\ \underbrace{\texttt{000}}_{s_6}$$

$$B_0 = \underbrace{0}_{s_1}\ \underbrace{1}_{s_2}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_4}\ \underbrace{1}_{s_5}\ \underbrace{1}_{s_6}$$

Elements s_1 and s_4 terminate at level 0 ($bit = 0$); elements s_2, s_3, s_5, and s_6 continue to level 1 ($bit = 1$).

Level 1 ($b_1 = 3$ bits per chunk, $n_1 = 4$ chunks):

$$A_1 = \underbrace{\texttt{010}}_{s_2}\ \underbrace{\texttt{100}}_{s_3}\ \underbrace{\texttt{111}}_{s_5}\ \underbrace{\texttt{010}}_{s_6}$$

$$B_1 = \underbrace{0}_{s_2}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_5}\ \underbrace{1}_{s_6}$$

Elements s_2 and s_5 terminate at level 1 ($bit = 0$); elements s_3 and s_6 continue to level 2 ($bit = 1$).

Level 2 ($b_2 = 3$ bits per chunk, $n_2 = 2$ chunks, last level—no bitmap):

$$A_2 = \underbrace{\texttt{001}}_{s_3}\ \underbrace{\texttt{001}}_{s_6}$$

No bitmap B_2 is allocated: all elements reaching the last level terminate there by definition.

3.2. Design Strategy

With per-level independent arrays in place, modifications can be performed directly on the compressed representation without ever reconstructing the integer sequence. All dynamic operations operate through bit-level manipulation on the independent level arrays, maintaining the existing chunk sizes and never reorganizing the structure. Operations that act exclusively at the tail of the structure, such as `append` and `pop`, require no bit shifting and run in $O(\lceil \log M/b \rceil)$ amortized time, where M is the maximum value in the sequence and b is the chunk size parameter. Operations that act at

arbitrary positions such as `insert`, `delete`, and `replace`, require shifting all subsequent bits in each affected level array and run in $O(n/W)$ amortized time, where n is the number of elements and W is the machine word size. Both bounds are amortized over the reallocation cost of the underlying bit arrays, which use a standard doubling strategy so that reallocations occur at exponentially decreasing frequency. All operations are memory-efficient, as chunk sizes are fixed at creation time and the structure is never reorganized. Note that this also means that the chunk size configuration does not adapt to changes in the data distribution after many modifications.

3.3. Core Bit-Level Operations

All dynamic operations are built upon three primitive bit-array operations: `insertBits`, `removeBits`, and `bitwrite`. These operate at word granularity for efficiency, running in $O(n/W)$ time, where n is the number of bits in the array and W is the machine word size. Three additional operations manipulate the per-level bitmaps while maintaining their rank support structures: `insertRank`, which inserts a bit at a given position and updates the rank structure; `deleteRank`, which removes a bit and returns its value; and `replaceRank`, which overwrites a bit and returns its old value. The rank support structure uses the classical block-and-superblock scheme with blocks of $\Theta(\log^2 n)$ bits. After any insertion or deletion, the prefix-sum counts stored in the blocks to the right of the modified position become stale and must be recomputed; updating the full block structure therefore costs $O(n/\log^2 n)$. Since $W \ll \log^2 n$ for any practical n, this update cost is dominated by the $O(n/W)$ cost of the underlying bitmap modification, so the overall time for `insertRank` and `deleteRank` remains $O(n/W)$. Together these six operations form the foundation for `append`, `pop`, `insert`, `replace`, `delete`, and `rebuild`.

3.4. Optimal Chunk Size Computation

A critical component of DACs is determining the chunk sizes $b_0, b_1, \ldots, b_{L-1}$ that minimize the total space required to represent a dataset. The original DACs work [1] introduced a dynamic programming algorithm called `optimize` for this purpose. Our implementation uses this algorithm, which we refer to as `optimizationk`, following the naming convention in the original DACs codebase. This algorithm is invoked only during two scenarios: initial structure creation and explicit rebuild operations.

Algorithm 1 optimizationk(D, n)

Find maximum value M in dataset D of size n.
Build frequency histogram: $weight[v] \leftarrow$ count of value v in D.
Convert to cumulative frequency array $acumFreq$:
    $acumFreq[i] \leftarrow$ number of values requiring at least i bits.
Use dynamic programming to find optimal chunk sizes $b_0, \ldots, b_{L-1}$:
    Minimize total space while balancing level count and chunk sizes.
return optimal chunk sizes array and number of levels L.

The algorithm scans D once to find the maximum value M and build the frequency histogram, requiring $O(n)$ time and $O(M \log n)$ bits of space, where n is the number of elements and each of the M histogram counters requires $O(\log n)$ bits. The cumulative frequency conversion takes $O(M)$ time. The dynamic programming step evaluates all candidate chunk size configurations in $O(\log^2 M)$ time and $O(\log M)$ bits of space [1]. The total time cost is therefore $O(n+M+\log^2 M)$, and the dominant space requirement is $O(M \log n)$ bits for the histogram.

3.5. Append Operation

Appending a value v adds it to the end of the sequence. Chunk sizes remain unchanged; the existing b configuration is used, adding new levels if v requires more than currently exist while maintaining the same chunk size pattern. No reorganization or optimization occurs.

The procedure works by splitting v into its base-2^{b_i} chunks and writing each chunk at the tail of the corresponding level array. Because every write targets the last position, no existing data needs to be shifted: level 0 receives the least significant chunk, and each subsequent level receives the next chunk until the value is fully encoded. A continuation bit of 0 is appended to each intermediate bitmap B_i to signal that the value continues to the next level, and a terminating bit of 1 marks the final level. If v requires more levels than currently exist, new level arrays are allocated on demand by promoting the former last level to a non-terminal level.

Algorithm 2 Append(C, v)

Input: DACs $C = (A_0, \ldots, A_{L-1},\ B_0, \ldots, B_{L-2},\ b)$, integer $v \geq 0$.

Output: DACs $C' = (A'_0, \ldots, A'_{L'-1},\ B'_0, \ldots, B'_{L'-2},\ b')$ with v appended to the sequence.

1: Determine termination level k (the highest level required to represent v under the current b configuration).
2: **if** $k \geq L$ **then**
3: Allocate new level A_L with chunk size $b_L \leftarrow \lceil \log_2(v+1) \rceil - \sum_{i=0}^{L-1} b_i$.
4: Create bitmap B_{L-1} with all `sizeLastLevel` bits set to 0. {existing elements terminate at level $L-1$}
5: $L' \leftarrow L + 1$.
6: **end if**
7: $v' \leftarrow v$.
8: **for** $i = 0$ **to** $k - 1$ **do**
9: Write $v' \bmod 2^{b_i}$ to the end of A_i using `bitwrite`.
10: $v' \leftarrow \lfloor v'/2^{b_i} \rfloor$.
11: Append bit 1 to B_i. {element continues to next level}
12: **end for**
13: Write $v' \bmod 2^{b_k}$ to the end of A_k using `bitwrite`.
14: **if** $k < L - 1$ **then**
15: Append bit 0 to B_k. {element terminates at this level}
16: **else**
17: `sizeLastLevel` $\leftarrow$ `sizeLastLevel` $+ 1$.
18: **end if**
19: `listLength` $\leftarrow$ `listLength` $+ 1$.

For a better understanding of the algorithm, see Example 2.

Example 2. *Starting from the DACs structure of Example 1 ($L = 3$, $b_0 = b_1 = b_2 = 3$, `listLength` $= 6$), we append $v = 13$.*

Line 1. *Determine the termination level k. Since $8 \leq 13 < 64$, the value requires two chunks and terminates at level 1, so $k = 1$.*

Line 2. *$k = 1 < L = 3$, so no new level is needed; skip to Line 6.*

Line 6. *$v' \leftarrow 13$.*

Lines 7–10 (loop, $i = 0$). *Write $v' \bmod 2^{b_0} = 13 \bmod 8 = 5 = \texttt{101}_2$ to the tail of A_0, update $v' \leftarrow \lfloor 13/8 \rfloor = 1$, and append bit 1 to B_0 (element*

continues to level 1):

$$A_0 = \underbrace{\texttt{101}}_{s_1}\ \underbrace{\texttt{100}}_{s_2}\ \underbrace{\texttt{100}}_{s_3}\ \underbrace{\texttt{011}}_{s_4}\ \underbrace{\texttt{100}}_{s_5}\ \underbrace{\texttt{000}}_{s_6}\ \underbrace{\texttt{101}}_{s_7}$$

$$B_0 = \underbrace{0}_{s_1}\ \underbrace{1}_{s_2}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_4}\ \underbrace{1}_{s_5}\ \underbrace{1}_{s_6}\ \underbrace{1}_{s_7}$$

Line 11. *Write* $v' \bmod 2^{b_1} = 1 \bmod 8 = 1 = \texttt{001}_2$ *to the tail of* A_1*:*

$$A_1 = \underbrace{\texttt{010}}_{s_2}\ \underbrace{\texttt{100}}_{s_3}\ \underbrace{\texttt{111}}_{s_5}\ \underbrace{\texttt{010}}_{s_6}\ \underbrace{\texttt{001}}_{s_7}$$

Lines 12–13. $k = 1 < L - 1 = 2$*, so append bit* 0 *to* B_1 *(element terminates at level 1):*

$$B_1 = \underbrace{0}_{s_2}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_5}\ \underbrace{1}_{s_6}\ \underbrace{0}_{s_7}$$

Line 15. `listLength` $\leftarrow 7$*.* A_2 *and* B_2 *are unchanged since* s_7 *does not reach level 2. No bit shifting occurred at any level.*

Time complexity. The algorithm visits exactly $k + 1 \leq L$ levels. At each level i, it performs one `bitwrite` call in $O(1)$ time under the Word-RAM model, since `bitwrite` writes a single word and every chunk size b_i is $O(\log n)$, and appends one bit to B_i in $O(1)$ time. No bit shifting occurs since all writes are at the tail of each array. The total time is therefore $O(k + 1) \subseteq O(L) = O(\lceil \log M/b \rceil)$, where M is the maximum value in the sequence.

Space complexity. Each invocation adds exactly one chunk of b_i bits to each visited level array A_i, and one continuation bit to each visited bitmap B_i. No existing data is moved or reallocated. The additional storage per operation is $O(1)$ words beyond the data appended.

3.6. Pop Operation

Popping removes the last element from the sequence. It is the symmetric counterpart of `append`: the procedure traverses levels from 0 upward, reading the continuation bit at the last position of each bitmap B_i to determine whether the element extends to the next level. At each visited level, the final b_i-bit chunk is removed from A_i and the corresponding bit is removed from B_i. Because every removal targets the tail of each array, no existing data needs to be shifted, and the operation completes in $O(L)$ time. If the last level becomes empty after the removal, it is discarded and L is decremented.

3.7. Insert Operation

Inserting a value v at position p requires shifting all subsequent chunks at each affected level. At level 0, the insertion position is $p - 1$. At each subsequent level i, the position is computed as $\text{rank}_1(B_{i-1}, p-1)$, which counts how many of the preceding elements extend to that level. The `insertBits` function shifts all subsequent bits in A_i rightward by b_i positions, the new chunk is written, and the appropriate bit is inserted into B_i. If $p = \text{listLength} + 1$, the operation delegates to `append`.

Algorithm 3 Insert(C, p, v)

Input: DACs $C = (A_0, \ldots, A_{L-1},\ B_0, \ldots, B_{L-2},\ b)$, integer p $(1 \leq p \leq \texttt{listLength} + 1)$, integer $v \geq 0$.

Output: DACs $C' = (A'_0, \ldots, A'_{L'-1},\ B'_0, \ldots, B'_{L'-2},\ b')$ with v inserted at position p.

if $p = \texttt{listLength} + 1$ **then**
  return Append(C, v).
end if
Determine termination level k for v under the current b configuration.
if $k \geq L$ **then**
  Allocate new level A_L with chunk size $b_L \leftarrow \lceil \log_2(v+1) \rceil - \sum_{i=0}^{L-1} b_i$.
  Create bitmap B_{L-1} with all `sizeLastLevel` bits set to 0. {existing elements terminate at level $L-1$}
  $L' \leftarrow L + 1$.
end if
$v' \leftarrow v$.
$pos \leftarrow p - 1$. {convert to 0-based index}
for $i = 0$ **to** $k - 1$ **do**
  if $i > 0$ **then**
    $pos \leftarrow pos - \text{rank}_1(B_{i-1},\ pos-1)$. {recalculate position at this level}
  end if
  `insertBits`$(A_i,\ pos \cdot b_i,\ b_i)$. {shift bits right to make space}
  `bitwrite`$(A_i,\ pos \cdot b_i,\ b_i,\ v' \bmod 2^{b_i})$.
  $v' \leftarrow \lfloor v'/2^{b_i} \rfloor$.
  `insertRank`$(B_i,\ pos,\ 1)$. {element continues to next level; updates rank structure}
end for
if $k > 0$ **then**
  $pos \leftarrow pos - \text{rank}_1(B_{k-1},\ pos - 1)$.
end if
`insertBits`$(A_k,\ pos \cdot b_k,\ b_k)$.
`bitwrite`$(A_k,\ pos \cdot b_k,\ b_k,\ v' \bmod 2^{b_k})$.
if $k < L - 1$ **then**
  `insertRank`$(B_k,\ pos,\ 0)$. {element terminates at this level; updates rank structure}
else
  `sizeLastLevel` $\leftarrow$ `sizeLastLevel` $+ 1$.
end if
`listLength` $\leftarrow$ `listLength` $+ 1$.

For a better understanding of the algorithm, see Example 3.

Example 3. *Starting from the DACs structure of Example 1 ($L = 3$, $b_0 = b_1 = b_2 = 3$, $\texttt{listLength} = 6$), we insert $v = 11$ at position $p = 3$, producing $S' = \langle 5, 20, 11, 100, 3, 60, 80 \rangle$.*

Line 1. *$p = 3 \neq \texttt{listLength} + 1 = 7$; the operation does not delegate to* ***Append****.*

Line 3. *Since $8 \leq 11 < 64$, the value requires two chunks and terminates at level 1, so $k = 1$.*

Line 4. *$k = 1 < L = 3$; no new level is needed.*

Line 8. *$v' \leftarrow 11$.*

Line 9. *$pos \leftarrow p - 1 = 2$.*

Lines 10–17 (loop, $i = 0$). *$i = 0$, so Line 11 is skipped (pos stays 2).* ***insertBits*** *shifts all bits from position $pos \cdot b_0 = 6$ onwards in A_0 rightward by $b_0 = 3$ positions, opening a slot at index 2.* ***bitwrite*** *writes $v' \bmod 8 = 3 = \texttt{011}_2$ into that slot, then $v' \leftarrow \lfloor 11/8 \rfloor = 1$.* ***insertRank*** *inserts bit 1 at position 2 in B_0 (element continues to level 1):*

$$A_0 = \underbrace{\texttt{101}}_{s_1}\ \underbrace{\texttt{100}}_{s_2}\ \underbrace{\texttt{011}}_{\mathbf{s_7}}\ \underbrace{\texttt{100}}_{s_3}\ \underbrace{\texttt{011}}_{s_4}\ \underbrace{\texttt{100}}_{s_5}\ \underbrace{\texttt{000}}_{s_6}$$

$$B_0 = \underbrace{0}_{s_1}\ \underbrace{1}_{s_2}\ \underbrace{1}_{\mathbf{s_7}}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_4}\ \underbrace{1}_{s_5}\ \underbrace{1}_{s_6}$$

Line 20. *$k = 1 > 0$, so the position at level 1 is recalculated using the already-updated B_0: $pos \leftarrow 2 - \text{rank}_1(B_0, 1) = 2 - 1 = 1$.*

Lines 21–22. ***insertBits*** *opens a slot at index 1 of A_1.* ***bitwrite*** *writes $v' \bmod 8 = 1 = \texttt{001}_2$ into that slot:*

$$A_1 = \underbrace{\texttt{010}}_{s_2}\ \underbrace{\texttt{001}}_{\mathbf{s_7}}\ \underbrace{\texttt{100}}_{s_3}\ \underbrace{\texttt{111}}_{s_5}\ \underbrace{\texttt{010}}_{s_6}$$

Lines 23–24. *$k = 1 < L - 1 = 2$, so* ***insertRank*** *inserts bit 0 at position 1 in B_1 (element terminates at level 1):*

$$B_1 = \underbrace{0}_{s_2}\ \underbrace{0}_{\mathbf{s_7}}\ \underbrace{1}_{s_3}\ \underbrace{0}_{s_5}\ \underbrace{1}_{s_6}$$

Line 27. $\texttt{listLength} \leftarrow 7$. *$A_2$ is unchanged since s_7 does not reach level 2.*

Time complexity. At level 0, `insertBits` must shift all bits after position $pos \cdot b_0$ in A_0. Since A_0 contains n elements each occupying b_0 bits, this costs $O(n \cdot b_0/W)$ word operations. At each subsequent level $j > 0$, the shift operates on $|A_j|$ elements each occupying b_j bits, costing $O(|A_j| \cdot b_j/W)$. Each rank computation and bitmap insertion takes $O(1)$. Summing over all L levels:

$$T_{\text{insert}} = \sum_{j=0}^{k-1} O\left(\frac{|A_j| \cdot b_j}{W}\right) + O(L).$$

Since $|A_0| = n$ and $|A_j| \leq n$ for all j, and b_j is bounded by a constant, the level-0 term $O(n \cdot b_0/W) = O(n/W)$ dominates. The total time is therefore $O(n/W)$, consistent with the bound stated in Section 1.

Space complexity. Each invocation adds exactly one chunk per visited level array and one bit per visited bitmap, contributing $O\left(\sum_{j=0}^{k}(b_j + 1)\right)$ bits of new storage. No existing data is reallocated. The additional storage per operation is $O(1)$ words beyond the data inserted.

3.8. Delete Operation

Deleting the element at position p removes its chunk from each level it occupies, shifting all subsequent chunks at each affected level.

The procedure traverses downward from level 0. At each level, `removeBits` removes the b_i-bit chunk at the current position. `deleteRank` inspects and removes the continuation bit: if it was 0 (terminating), the traversal stops. If it was 1 (continuing), the position is recalculated as $\text{rank}_1(B_i, pos)$ and the traversal continues to the next level. If the deletion empties the last level, that level is discarded and L is decremented.

Algorithm 4 Delete(C, p)

Input: DACs $C = (A_0, \ldots, A_{L-1},\ B_0, \ldots, B_{L-2},\ b)$, integer p ($1 \le p \le$ `listLength`).

Output: DACs $C' = (A'_0, \ldots, A'_{L'-1},\ B'_0, \ldots, B'_{L'-2},\ b')$ with the element at position p removed.

1: $pos \leftarrow p - 1$.
2: $k \leftarrow 0$.
3: **while** $k < L$ **do**
4: **if** $k > 0$ **then**
5: $pos \leftarrow pos - \text{rank}(B_{k-1},\ pos - 1)$.
6: **end if**
7: `removeBits`$(A_k,\ pos \cdot b_k,\ b_k)$.
8: **if** $k = L - 1$ **then**
9: **if** `sizeLastLevel` > 1 **then**
10: `sizeLastLevel` $\leftarrow$ `sizeLastLevel` $- 1$.
11: **else**
12: `sizeLastLevel` $\leftarrow |B_{L-2}|$.
13: A_{L-1} and B_{L-2} are discarded; $L \leftarrow L - 1$.
14: **end if**
15: **break**.
16: **end if**
17: $deleted \leftarrow$ `deleteRank`$(B_k,\ pos)$.
18: **if** $deleted = 1$ **then**
19: **break**.
20: **end if**
21: $k \leftarrow k + 1$.
22: **end while**
23: `listLength` $\leftarrow$ `listLength` $- 1$.

Time complexity. At level 0, `removeBits` must shift all bits after position $pos \cdot b_0$ in A_0. Since A_0 contains n elements each occupying b_0 bits, this costs $O(n \cdot b_0/W)$ word operations. At each subsequent level $k > 0$, the shift operates on $|A_k|$ elements each occupying b_k bits, costing $O(|A_k| \cdot b_k/W)$. Each rank computation and the `deleteRank` call take $O(1)$. Summing over the at most L levels visited:

$$T_{\text{delete}} = \sum_{k=0}^{L-1} O\left(\frac{|A_k| \cdot b_k}{W}\right) + O(L).$$

Since $|A_0| = n$ dominates all higher-level terms, the total time reduces to $O(n/W)$, consistent with the bound stated in Section 1. If the last level becomes empty and is discarded, this adds $O(1)$ work and does not affect the asymptotic cost.

Space complexity. Each invocation removes exactly one chunk per visited level array and one bit per visited bitmap. No new storage is allocated. The space change per operation is $O(1)$ words.

3.9. Replace Operation

Replacing the element at position p with value v handles three cases based on the number of levels occupied by the old value (k_{old}) versus the new value (k_{new}).

The procedure traverses levels from 0 upward. At each level, `bitwrite` overwrites the chunk and `replaceRank` updates the continuation bit, returning the old bit value. If $k_{old} = k_{new}$, all levels are overwritten in place. If $k_{old} > k_{new}$, the new value terminates earlier and the remaining chunks at deeper levels are removed using `removeBits` and `deleteRank`. If $k_{old} < k_{new}$, the new value extends deeper and additional chunks are inserted using `insertBits` and `insertRank`. Levels where both values exist are modified in place; only levels where the depths differ require bit shifting.

Algorithm 5 Replace(C, p, v)

Input: DACs $C = (A_0, \ldots, A_{L-1},\ B_0, \ldots, B_{L-2},\ b)$, integer p $(1 \leq p \leq$ `listLength`$)$, integer $v \geq 0$.

Output: DACs $C' = (A'_0, \ldots, A'_{L'-1},\ B'_0, \ldots, B'_{L'-2},\ b')$ with the element at position p replaced by v.

Determine termination level k_{new} for v under the current b configuration.
if $k_{new} \geq L$ **then**
  Allocate new level A_L with chunk size $b_L \leftarrow \lceil \log_2(v+1) \rceil - \sum_{i=0}^{L-1} b_i$.
  Create bitmap B_{L-1} with all `sizeLastLevel` bits set to 0. {existing elements terminate at level $L-1$}
  $L \leftarrow L + 1$.
end if
$v' \leftarrow v$.
$pos \leftarrow p - 1$.
$k \leftarrow 0$; $e \leftarrow 0$.
while $k \leq k_{new}$ **and** $k < L$ **do**
  if $k > 0$ **then**
    $pos \leftarrow pos - \text{rank}_1(B_{k-1},\ pos - 1)$.
  end if
  `bitwrite`$(A_k,\ pos \cdot b_k,\ b_k,\ v' \bmod 2^{b_k})$.
  $v' \leftarrow \lfloor v'/2^{b_k} \rfloor$.
  if $k = L - 1$ **then**
    return.
  end if
  if $k < k_{new}$ **then**
    $e \leftarrow$ `replaceRank`$(B_k,\ pos,\ 1)$. {element continues to next level}
  else
    $e \leftarrow$ `replaceRank`$(B_k,\ pos,\ 0)$. {element terminates at this level}
  end if
  if $e = 0$ **then**
    break.
  end if
  $k \leftarrow k + 1$.
end while
if $e = 1$ **and** $k > k_{new}$ **then**
  Old value was deeper; delete remaining chunks from levels $k+1 \ldots k_{old}$ using `removeBits` and `deleteRank`, removing empty levels as needed.
end if
if $e = 0$ **and** $k \leq k_{new}$ **then**
  Old value was shallower; insert remaining chunks into levels $k+1 \ldots k_{new}$ using `insertBits`, `bitwrite`, and `insertRank`.
end if

Time complexity. The algorithm first traverses levels 0 through $\min(k_{old}, k_{new})$ using only `bitwrite` and `replaceRank`, each in $O(1)$ per level. If the depths differ, the remaining levels require `insertBits` or `removeBits` calls. The most expensive such call occurs at the first diverging level j^*, which contains $|A_{j^*}|$ elements. Since $|A_{j^*}| \leq |A_0| = n$, the cost of that shift is at most $O(n \cdot b_{j^*}/W) = O(n/W)$. Summing over all visited levels:

$$T_{\text{replace}} = \sum_{j=0}^{\max(k_{old},k_{new})} O\left(\frac{|A_j| \cdot b_j}{W}\right) + O(L) = O(n/W),$$

consistent with the bound stated in Section 1.

Space complexity. The number of chunks stored changes by at most $|k_{new} - k_{old}|$ chunks across the affected levels, which is bounded by L. This constitutes $O(1)$ additional words beyond the data modified.

3.10. Rebuild Operation

The rebuild operation reconstructs the structure with optimal chunk sizes for the current data distribution.

The procedure decompresses all n values, invokes `optimizek` to compute optimal chunk sizes, and reconstructs the DACs structure with the new configuration. If the new chunk sizes match the existing ones, the structure is unchanged. The complexity is $O(|C|)$.

Algorithm 6 Rebuild(C)

Input: DACs $C = (A_0, \ldots, A_{L-1},\ B_0, \ldots, B_{L-2},\ b)$.

Output: DACs $C' = (A'_0, \ldots, A'_{L'-1},\ B'_0, \ldots, B'_{L'-2},\ b')$ with optimal chunk sizes for the current data distribution.

1: $D \leftarrow$ `decompress`(C).
2: $b',\ L' \leftarrow$ `optimizationk`$(D,$ `listLength`$)$.
3: **if** $b' = b$ **and** $L' = L$ **then**
4: **return**.
5: **end if**
6: $C_{new} \leftarrow$ `create`$(D,$ `listLength`$,\ b')$.
7: Assign $A'_0, \ldots, A'_{L'-1}, B'_0, \ldots, B'_{L'-2}$, `base`, `base_bits`, and L' from C_{new}.

The rebuild operation restores optimal compression after extensive modifications. It first traverses the entire structure to recover all n integer values,

then invokes `optimizationk` to compute the optimal chunk size configuration for the current data distribution. If the optimal configuration matches the existing one, no rebuild is necessary and the operation returns immediately. Otherwise, a new structure is built with the optimal encoding and its internal arrays replace those of the original, which are freed. The decision to rebuild involves a fundamental trade-off: during periods of frequent modifications it is more practical to accept slightly suboptimal compression, while at strategic points, such as when transitioning from an update-heavy to a query-heavy workload, the one-time reconstruction cost becomes worthwhile. We quantify this trade-off experimentally in Section 4.3.

The operation first decompresses all n values by traversing the structure in $O(n \cdot L)$ time, requiring $O(n \log M)$ bits of additional space for the decompressed array, where each of the n integers requires $\lceil \log_2(M+1) \rceil$ bits. It then invokes `optimizationk` in $O(n + M + \log^2 M)$ time using $O(M \log n)$ bits of additional space for the frequency histogram, where M is the maximum value. Finally, it constructs the new structure in $O(n \cdot L')$ time. The total time cost is therefore $O(n \cdot L + M + \log^2 M)$. The peak additional space required during reconstruction is $O(n \log M + M \log n)$ bits, after which the old arrays are discarded and only the new optimally encoded structure is retained.

4. Experimental Analysis

We present our modified DACs implementation written in C and built directly upon the original DACs codebase [1]. We compare it against static DACs as a baseline for compression ratio and access time, and against SPSI [2], the state-of-the-art dynamic compressed structure based on a B-tree representation. We make the source code available at GitHub.[2] All implementations are single-threaded and compiled with gcc 7.5.0 using the `-O3` optimization flag. Experiments were performed on a dedicated workstation with an AMD Ryzen 7 3700X processor (3.59 GHz, 8 cores, 16 threads) and 23 GB RAM, running Ubuntu 18.04.6 LTS with kernel 6.6.87.2 (64-bit). We evaluated on five datasets summarized in Table 2, all sourced from the Pizza&Chili corpus.[3] The LCP datasets (`dblp`, `proteins`, `dna`) were derived

[2] `https://github.com/BGMP/DACs`

[3] `http://pizzachili.dcc.uchile.cl/`

from the 100 MB Pizza&Chili collections used in the original DACs evaluation [1]. Each raw text file was processed using an LCP Array Constructor tool,[4] which builds the suffix array via a doubling algorithm and computes LCP values using Kasai's algorithm, writing each value as a 4-byte integer. Consequently, a 100 MB input text yields a 400 MB LCP file, as each of the n input bytes produces one 32-bit LCP entry. The `boost.200MB` dataset follows the DYNAMIC benchmark [2], enabling direct comparison with SPSI under the same conditions used in its original evaluation. The `dna.100MB` raw dataset complements this by testing performance on byte-level sequences with a very restricted alphabet.

Table 2: Dataset characteristics.

Dataset	Type	Elements	Max	Avg	Median	Mode
dblp.xml.100MB.lcp	LCP	104,857,600	1,084	44.45	32.0	10
proteins.100MB.lcp	LCP	104,857,600	35,246	220.65	6.0	6
dna.100MB.lcp	LCP	104,857,600	17,772	28.10	13.0	13
boost.200MB	Raw	209,715,200	126	76.01	97.0	32
dna.100MB	Raw	104,857,600	89	72.24	71.0	65

We conducted two experiments. The first measures dynamic operations over a pre-built DACs structure: `append`, `insert`, `replace`, `delete`, and `pop`. Each procedure is tested for $N \in \{1, 10, 100, 1000\}$ operations, averaged over 10 independent iterations. Two variants are measured for each case: without rebuild, which reflects the raw cost of the bit-level operation, and with rebuild, which additionally calls `rebuild` to restore optimal compression after the operations. Both the inserted values and the target indices are supplied as externally generated sequences of uniform random 32-bit unsigned integers, produced using a random numbers generator tool.[5]

The second experiment measures static operations on our modified DACs implementation built directly from each dataset: structure creation time (`create`), random access time per element (`access`), and full decompression time (`decompress`). Access time is measured by accessing all n elements in a uniformly shuffled random order, using a fixed seed for reproducibility, and is reported as the average of 10 independent iterations together with its

[4] `https://github.com/BGMP/LCPArrayConstructor`

[5] `https://github.com/BGMP/RandomNumbersGenerator`

standard deviation. The same procedure is applied for creation and decompression.

4.1. Dynamic Operation Setup

For the dynamic operation experiments we depart from the uniform $[1, 100{,}000]$ range used for `append` and instead draw inserted values from a range calibrated to each dataset's existing level structure. Inserting a value larger than the maximum the current DACs levels can represent would force the creation of new levels, which adds structural overhead unrelated to the operation being measured. For LCP datasets, which encode sequences over a bounded domain, insertions outside that domain are unlikely in practice; although such insertions may occur, their time impact is marginally higher and averages out over representative workloads. For raw byte-level datasets, however, inserting a value outside the representable range is not a realistic scenario, and doing so would cause a significant increase in processing time due to the creation of entirely new levels in a structure that otherwise consists of a single level. To standardise the experimental conditions across all datasets and isolate the cost of the dynamic operations themselves, we therefore restrict inserted values to the full capacity of the base used by the structure for each dataset: the largest value expressible within the existing level configuration. The resulting per-dataset ranges are summarised in Table 3.

Table 3: Value and index ranges used for dynamic operation experiments. Element ranges span the full representable capacity of each dataset's DACs level structure.

Dataset	Encoding base	DACs Levels	Elements (min–max)	Indices (min–max)
dblp.xml.100MB.lcp	Base-64	5	[0, 63]	[1, 104,857,600]
dna.100MB.lcp	Base-16	8	[0, 15]	[1, 104,857,600]
proteins.100MB.lcp	Base-8	8	[0, 7]	[1, 104,857,600]
dna.100MB	Base-128	1	[0, 127]	[1, 104,857,600]
boost.200MB	Base-128	1	[0, 127]	[1, 209,715,200]

4.2. DACs vs. SPSI: Dynamic Operations

Figures 1 and 2 compare the execution time of DACs and SPSI for all five dynamic operations at $n = 1{,}000$, across all datasets. Each row corresponds to a dataset; within each row, operations are plotted on a logarithmic scale, with colour distinguishing operations and shape distinguishing structure (circle for DACs, square for SPSI). A segment connects the two structures for each operation, making the relative performance gap immediately visible. Bars show the standard deviation across 10 independent runs.

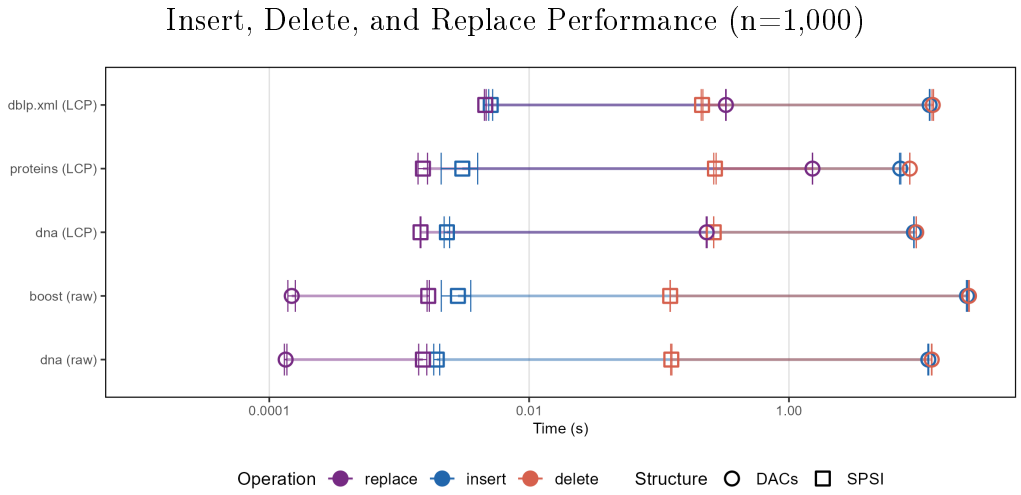

Figure 1: Execution time (seconds, log scale) for `insert`, `delete`, and `replace` on a batch of $n = 1{,}000$ operations, averaged over 10 runs ($\pm\sigma$). Each row corresponds to one dataset (see Table 2). Colour encodes operation; shape encodes structure ($\circ$ DACs, $\square$ SPSI). Segments connect the two structures for the same operation to highlight the performance gap.

Figure 1 covers `insert`, `delete`, and `replace`. `insert` and `delete` are inherently the most expensive operations: both insert or remove a chunk at an arbitrary position in each level array, requiring `insertBits` or `removeBits` to physically shift every bit after the target position. The cost therefore scales with $O(n \cdot L/W)$, where W is the machine word size, and SPSI outperforms DACs on both of these operations across all datasets. `replace` is cheaper in our setting: since inserted values are drawn from the same base range as the existing data (Table 3), the old and new values always occupy identical level footprints, and the operation reduces to L in-place `bitwrite` calls at fixed bit positions with no bit shifting required.

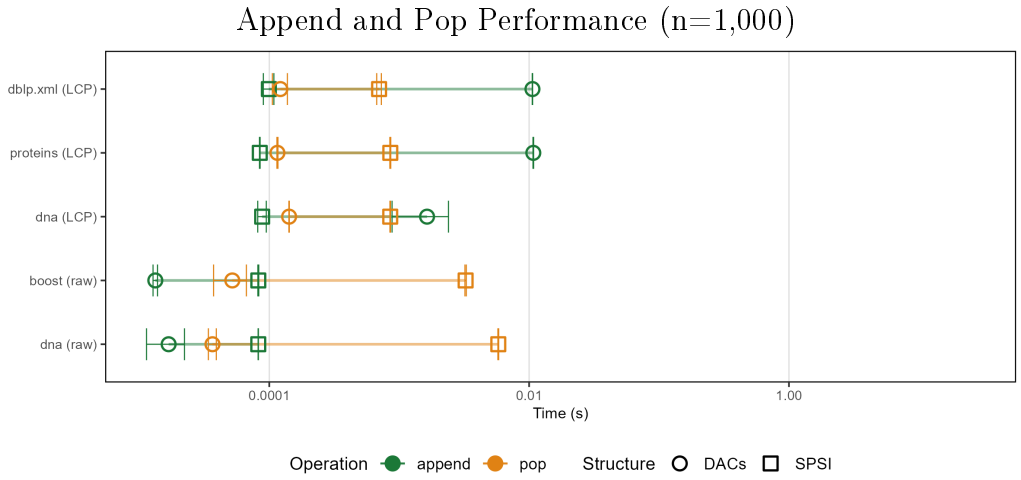

Figure 2: Execution time (seconds, log scale) for `append` and `pop` on a batch of $n = 1{,}000$ operations, averaged over 10 runs ($\pm\sigma$). Each row corresponds to one dataset (see Table 2). Colour encodes operation; shape encodes structure ($\circ$ DACs, $\square$ SPSI).

Figure 2 shows `append` and `pop`, which act exclusively at the tail of each level array, writing or removing the final chunk in $O(\lceil \log M/b \rceil)$ `bitwrite` or `removeBits` calls with no data movement. The relative performance of DACs and SPSI on these operations depends on the number of levels in the DACs structure. On the raw datasets (`boost.200MB` and `dna.100MB`), which are encoded with a single level, DACs outperform SPSI on both `append` and `pop`, as the tail operation reduces to a single `bitwrite` call with no tree traversal. On the multi-level LCP datasets (`dblp`, `proteins`, `dna`), SPSI achieves lower execution times on `append`, as its more compact internal representation yields an advantage on the skewed distributions characteristic of these datasets.

4.3. Rebuild vs. No Rebuild

Table 4 summarises static operation results across all datasets, and Figure 3 illustrates the rebuild cost trade-off using `append` on the proteins dataset as a representative example.

DACs (modified) and static DACs achieve equivalent compressed sizes and access times across all datasets, confirming that the incorporation of dynamic capabilities introduces no overhead to either space usage or query performance. SPSI requires 2–3$\times$ more space than DACs and DACs (modified) on LCP datasets, and its access times are consistently 3–8$\times$ slower.

Table 4: Static operation results for DACs, DACs (modified), and SPSI across all datasets. Access time is reported per element in microseconds, averaged over 10 iterations (± standard deviation).

Dataset	Structure	Orig. (bytes)	Comp. (bytes)	Access (μs)	$\pm\sigma$	Create (s)	Decomp. (s)
dblp.xml.100MB.lcp	DACs	419,430,400	98,594,835	0.0803	0.0007	2.561	1.053
	DACs (modified)	419,430,400	98,594,747	0.0810	0.0041	2.762	1.081
	SPSI	419,430,400	205,555,813	0.2691	0.0031	7.955	3.507
proteins.100MB.lcp	DACs	419,430,400	86,241,069	0.1718	0.0011	3.573	1.677
	DACs (modified)	419,430,400	86,240,993	0.1665	0.0044	3.808	1.754
	SPSI	419,430,400	245,170,069	0.2869	0.0074	8.000	3.544
dna.100MB.lcp	DACs	419,430,400	72,657,801	0.0883	0.0013	2.882	1.369
	DACs (modified)	419,430,400	72,657,709	0.0902	0.0062	3.085	1.188
	SPSI	419,430,400	245,164,469	0.2885	0.0039	7.986	3.546
boost.200MB	DACs	209,715,200	183,500,875	0.0367	0.0007	2.675	1.936
	DACs (modified)	209,715,200	183,500,815	0.0357	0.0002	3.173	1.637
	SPSI	209,715,200	260,965,156	0.3879	0.0170	16.757	7.354
dna.100MB	DACs	104,857,600	91,750,475	0.0348	0.0013	1.308	0.953
	DACs (modified)	104,857,600	91,750,415	0.0339	0.0004	1.605	0.819
	SPSI	104,857,600	130,482,541	0.2441	0.0007	8.081	3.538

Creation and decompression times follow the same pattern, with SPSI taking roughly 2.5× longer than DACs (modified) across all datasets.

To illustrate the cost trade-off between performing raw dynamic operations and invoking `rebuild` afterwards, Figure 3 shows execution time and structure size for `append` on the proteins dataset, with values drawn uniformly at random from $[1, 100{,}000]$. Each point is the average of 10 independent runs ($\pm\sigma$). Two variants are compared: `DACs (rebuild)`, which calls `rebuild` once after every batch of n operations, and `DACs`, which performs only the raw bit-level append. The x-axis corresponds to the batch size $n \in \{1, 10, 100, 1000\}$, that is, the number of `append` operations performed before `rebuild` is invoked.

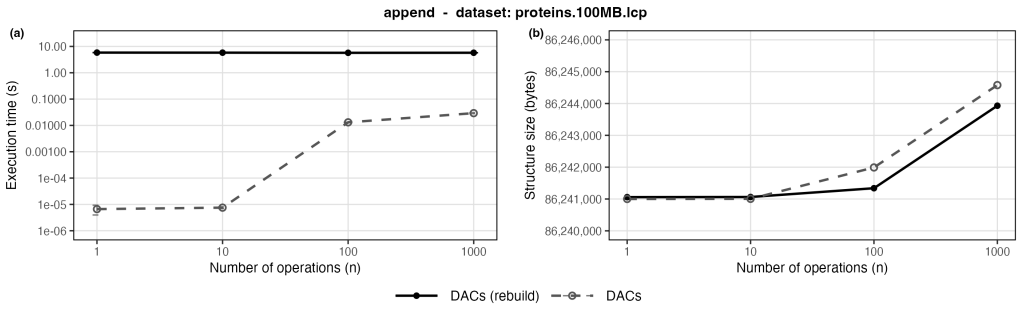

Figure 3: `append` on `proteins.100MB.lcp` with values drawn uniformly from $[1, 100{,}000]$, averaged over 10 runs ($\pm\sigma$). Panel (a): execution time (seconds) vs. batch size n. Panel (b): structure size (bytes) vs. batch size n. `DACs (rebuild)` invokes `rebuild` after each batch; `DACs` performs only the raw bit-level append.

The results tell a consistent story. In panel (a), `DACs` completes a single append in the microsecond range and remains well below 15 ms even for batches of 1,000 operations, while `DACs (rebuild)` incurs a near-constant cost of several seconds regardless of n. This flat profile is a direct consequence of `rebuild` traversing and reconstructing the *entire* compressed structure irrespective of how many values were appended: its cost is $O(|C|)$ in the size of the compressed sequence, not in the number of operations performed. Panel (b) shows that both variants reach equivalent structure sizes in all cases, with the two curves overlapping almost entirely. The rebuild therefore offers no meaningful compression benefit for `append`, making the plain `DACs` variant the clear choice for append-heavy workloads. The decision to invoke `rebuild` should be reserved for scenarios where a strict space guarantee is required and the application can tolerate the $O(|C|)$ reconstruction overhead.

5. Final Remarks

We presented our implementation that modifies the original Directly Addressable Codes to incorporate efficient in-place modifications: `append`, `insert`, `replace`, `delete`, and `pop`, operating directly on the compressed representation through bit-level manipulation without requiring full reconstruction. An explicit `rebuild` operation is also provided to restore optimal chunk sizes at the user's discretion, at the cost of a full $O(|C|)$ reconstruction. The source code is publicly available on GitHub.[6]

Experimental results on five datasets confirm that our modification preserves the space and access time properties of the original DACs entirely: compressed sizes remain within a few bytes of the static baseline, and random access times remain within 5% of it. As a direct consequence, the inherent advantages of the DAC representation over SPSI are fully retained. DACs use 2–3× less space on LCP datasets and achieve 2–11× faster random access. The trade-off introduced by our modification lies exclusively in dynamic operations. On datasets encoded with few levels, such as raw byte sequences, `append` and `pop` outperform SPSI as they reduce to a single `bitwrite` call with no bit shifting and no tree traversal. On multi-level LCP datasets, however, SPSI achieves lower `append` times due to its more compact internal representation for skewed distributions. `insert` and `delete`

[6] `https://github.com/BGMP/DACs`

are slower than SPSI across all datasets due to bit-level shifting across all levels. DACs are therefore best suited for applications where compressed size and access speed are prioritised and modifications are dominated by appends rather than arbitrary insertions and deletions.

As future work, we envision extending the experimental evaluation to cover a broader range of dynamic operation patterns, such as interleaved sequences of insertions, deletions, and replacements at varying positions, as well as workloads that deliberately push the structure toward suboptimal compression to better characterise the conditions under which periodic rebuilding is most beneficial. Evaluating DACs on additional dataset types beyond LCP arrays and raw byte sequences would also help establish the generality of the results. On the implementation side, migrating the underlying bit arrays from 32-bit to 64-bit word granularity is a natural next step, as it would halve the number of word-level operations required by `insertBits` and `removeBits` on 64-bit architectures and could yield measurable throughput improvements for all dynamic operations. Additionally, replacing the static bit arrays with dynamic bitvectors [15] could eliminate the $O(n/W)$ bit-shifting cost entirely, reducing insert and delete to $O(\log n)$ time and making DACs competitive with SPSI on arbitrary-position modifications while retaining their superior compression and access time properties.

Acknowledgements

Rodrigo Torres-Avilés acknowledges the support of the Grupo ALBA GI2638801 and the Fondecyt project 1230647.

References

[1] N. R. Brisaboa, S. Ladra, G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing & Management*, 49(1):392–404, 2013.

[2] D. Arroyuelo, G. de Bernardo, T. Gagie, G. Navarro. Faster dynamic compressed d-ary relations. In *String Processing and Information Retrieval (SPIRE)*, pages 419–433, 2019.

[3] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[4] M. Caniupán, R. Torres-Avilés, T. Gutiérrez-Bunster, M. Lepe. Efficient computation of map algebra over raster data stored in the k2-acc compact data structure. *Geoinformatica*, 26:95–123, 2022.

[5] S. Ladra, J. Paramá, F. Silva-Coira. Compact and queryable representation of raster datasets. In *Proc. 28th International Conference on Scientific and Statistical Database Management*, 15:1–12, 2016.

[6] F. Silva-Coira, J. Paramá, S. Ladra. Map algebra on raster datasets represented by compact data structures. *Software: Practice and Experience*, 53(6):1362–1390, 2023.

[7] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[8] G. Navarro. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7354, pages 2–26, 2012.

[9] R. Grossi, A. Gupta, J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[10] N. R. Brisaboa, R. Cánovas, F. Claude, M. Martínez-Prieto, G. Navarro. Compressed string dictionaries. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*, LNCS 6630, pages 136–147, 2011.

[11] P. Ferragina, R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proc. 18th Annual Symposium on Discrete Algorithms (SODA)*, pages 690–696, 2007.

[12] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[13] N. Prezza. A framework of dynamic data structures for string processing. In *42nd Conference on Very Important Topics (CVIT 2016)*, Article No. 23, pp. 23:1–23:12, 2017.

[14] H. E. Williams, J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[15] G. Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

5. Conclusiones Generales y Recomendaciones

Este trabajo abordó el problema de extender los Directly Addressable Codes con operaciones de modificación eficientes sobre la representación comprimida. A continuación se evalúa en qué medida los objetivos específicos planteados fueron alcanzados y en qué medida los resultados obtenidos se corresponden con las hipótesis formuladas al inicio del proyecto.

5.1. Desarrollo de Algoritmos Dinámicos sobre la Estructura Comprimida

El primer objetivo planteaba desarrollar algoritmos de adición, inserción, eliminación y modificación que operaran directamente sobre la estructura comprimida sin requerir decodificación completa. Se esperaba que esto fuera posible y que representara una mejora sustancial sobre el enfoque de reconstrucción completa.

Este objetivo se cumplió en su totalidad. Se implementaron cinco operaciones dinámicas que actúan exclusivamente sobre la representación comprimida, sin reconstruir en ningún momento la secuencia de enteros subyacente. Las operaciones que actúan sobre el extremo final de la estructura no requieren ningún desplazamiento de datos y se ejecutan en tiempo proporcional al número de niveles, mientras que las operaciones en posiciones arbitrarias requieren desplazar los datos posteriores en cada nivel afectado. Adicionalmente se implementó una operación de reconstrucción explícita con tamaños de chunk óptimos, a disposición del usuario para escenarios donde se requiera restaurar la compresión óptima tras una serie de modificaciones.

5.2. Optimización del Manejo de Memoria

El segundo objetivo buscaba que el overhead espacial de la estructura modificada fuera comparable al de los DACs estáticos originales. La hipótesis era que esto era alcanzable sin sacrificar las propiedades de compresión.

Los resultados experimentales confirman plenamente esta hipótesis. La estrategia clave consistió en reemplazar los arrays monolíticos de la implementación original por arrays independientes por nivel, de modo que una modificación en un nivel no afecte a los demás. Desde el punto de vista teórico, este cambio no introduce ningún costo espacial adicional respecto a la implementación original. En la práctica, los DACs modificados alcanzan tamaños comprimidos prácticamente idénticos a los DACs estáticos en todos los datasets evaluados, con diferencias de apenas unos pocos bytes. En comparación con SPSI, los DACs modificados consumen entre dos y tres veces menos memoria en los datasets de arrays LCP, e incluso en el dataset de mayor tamaño SPSI supera el tamaño original sin comprimir mientras los DACs modificados mantienen una compresión efectiva.

5.3. Implementación de Estructuras Auxiliares

El tercer objetivo consistía en implementar las estructuras auxiliares necesarias para que las operaciones dinámicas fueran ejecutables con eficiencia práctica. Se esperaba que estas estructuras no introdujeran penalización perceptible en las operaciones de lectura.

Se desarrollaron operaciones primitivas a nivel de bits y de bitmaps que constituyen la base de todas las operaciones dinámicas, operando con granularidad de palabra para maximizar la eficiencia. El resultado experimental más relevante respecto a este objetivo es que el tiempo de acceso aleatorio de los DACs modificados permanece dentro de un 5 % del tiempo de los DACs estáticos originales en todos los datasets evaluados, confirmando que las estructuras auxiliares incorporadas no introducen penalización alguna en las consultas de lectura.

5.4. Análisis Teórico de Complejidad

El cuarto objetivo exigía establecer cotas teóricas formales de complejidad temporal y espacial. La hipótesis planteaba que sería posible alcanzar complejidad sub-lineal para al menos algunas de las operaciones dinámicas.

Esta hipótesis se verificó parcialmente, con una distinción importante entre dos categorías de operaciones. Las operaciones que actúan sobre el extremo final de la estructura se ejecutan en tiempo logarítmico respecto al valor máximo de la secuencia, lo que constituye una complejidad sub-lineal respecto a n. Las operaciones en posiciones arbitrarias requieren tiempo $O(n/W)$, donde W es el tamaño de palabra de la máquina; esta cota es sub-lineal respecto al número de bits procesados gracias a la aritmética de palabras, aunque lineal en n/W. La complejidad espacial adicional de todas las operaciones es constante más allá de los datos insertados o eliminados. Estas cotas son consistentes entre sí y están respaldadas por el análisis formal de cada operación.

5.5. Evaluación Experimental

El quinto objetivo consistía en evaluar experimentalmente el rendimiento con datasets reales y sintéticos. Se esperaba demostrar que las operaciones dinámicas implementadas son significativamente más eficientes que la reconstrucción completa de la estructura.

Se realizaron dos experimentos sobre cinco datasets provenientes del corpus Pizza&Chili, incluyendo arrays LCP y datos crudos de distinta naturaleza. El primer experimento midió las cinco operaciones dinámicas para distintos tamaños de lote, promediados sobre diez iteraciones independientes, tanto con como sin reconstrucción posterior. El segundo experimento midió operaciones estáticas: creación de la estructura, acceso aleatorio y descompresión completa. Los resultados confirmaron que los DACs modificados preservan íntegramente las propiedades de espacio y tiempo de acceso de los DACs originales. En cuanto a la reconstrucción, los experimentos demostraron que su costo es proporcional al tamaño de la estructura completa, independientemente del número de operaciones realizadas, lo que confirma que debe reservarse para transiciones de carga de trabajo y no usarse rutinariamente tras cada modificación.

5.6. Comparación con Soluciones Alternativas

El sexto objetivo exigía comparar la solución propuesta con alternativas existentes. La hipótesis planteaba que los DACs modificados obtendrían mejor rendimiento que la reconstrucción completa y resultados competitivos frente a estructuras dinámicas del estado del arte.

Los resultados revelan un trade-off claro y bien definido frente a SPSI. Los DACs modificados consumen entre dos y tres veces menos memoria que SPSI en los datasets LCP y logran acceso aleatorio entre dos y once veces más rápido. El rendimiento relativo de las operaciones sobre el extremo de la secuencia depende del número de niveles de la estructura DACs. En datasets codificados con pocos niveles, como secuencias de bytes crudos, las operaciones de adición y extracción superan a SPSI, dado que se reducen a una única escritura sin desplazamiento de bits ni recorrido de árbol. En datasets LCP multi-nivel, sin embargo, SPSI logra menores tiempos de adición debido a su representación interna más compacta para distribuciones sesgadas. Las operaciones de inserción y eliminación en posiciones arbitrarias son más lentas que en SPSI en todos los datasets, debido al costo de desplazar todos los datos posteriores en cada nivel, representando el único trade-off introducido por la modificación. Los DACs modificados son por tanto la mejor opción para aplicaciones donde la eficiencia espacial y la velocidad de consulta son prioritarias y las modificaciones están dominadas por adiciones al final y reemplazos, mientras que SPSI afronta con menor costo las inserciones y eliminaciones arbitrarias frecuentes.

Frente al enfoque naive de reconstrucción completa, la ventaja de los DACs modificados es grande en todas las operaciones implementadas, validando la hipótesis central del trabajo.

5.7. Reflexiones Finales sobre el Proyecto de Investigación

El desarrollo de este trabajo representó una experiencia formativa integral en el contexto del Magíster en Ciencias de la Computación de la Universidad del Bío-Bío. El proyecto exigió recorrer completamente el ciclo de la investigación científica: desde la revisión del estado del arte y la identificación de un problema abierto, pasando por la formulación de una hipótesis, el diseño e implementación de una solución y su análisis teórico, hasta la evaluación experimental rigurosa y la escritura de un artículo científico destinado a publicación en una revista internacional.

En particular, trabajar sobre una estructura de datos compacta real implicó enfrentarse a decisiones de diseño de bajo nivel cuyas consecuencias son inmediatamente visibles en los resultados experimentales, lo cual desarrolló una comprensión más profunda de la relación entre representación de datos, uso de memoria y rendimiento en tiempo de ejecución. Esta perspectiva es difícil de adquirir en contextos más abstractos y constituye una de las contribuciones más valiosas del programa de magíster a la formación como investigador.

La colaboración con los profesores guía Rodrigo Torres-Avilés y Luis Cabrera-Crot, quienes son coautores del artículo asociado, permitió integrar el trabajo de tesis directamente en una línea de investigación activa del Departamento de Ciencias de la Computación de la Universidad del Bío-Bío, con proyección concreta hacia la publicación y la continuación de esta línea en trabajos futuros.

Referencias

[1] N. R. Brisaboa, S. Ladra, G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing & Management*, 49(1):392–404, 2013.

[2] D. Arroyuelo, G. de Bernardo, T. Gagie, G. Navarro. Faster dynamic compressed d-ary relations. In *String Processing and Information Retrieval (SPIRE)*, pages 419–433, 2019.

[3] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[4] M. Caniupán, R. Torres-Avilés, T. Gutiérrez-Bunster, M. Lepe. Efficient computation of map algebra over raster data stored in the k2-acc compact data structure. *Geoinformatica*, 26:95–123, 2022.

[5] S. Ladra, J. Paramá, F. Silva-Coira. Compact and queryable representation of raster datasets. In *Proc. 28th International Conference on Scientific and Statistical Database Management*, 15:1–12, 2016.

[6] F. Silva-Coira, J. Paramá, S. Ladra. Map algebra on raster datasets represented by compact data structures. *Software: Practice and Experience*, 53(6):1362–1390, 2023.

[7] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[8] G. Navarro. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7354, pages 2–26, 2012.

[9] R. Grossi, A. Gupta, J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[10] N. R. Brisaboa, R. Cánovas, F. Claude, M. Martínez-Prieto, G. Navarro. Compressed string dictionaries. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*, LNCS 6630, pages 136–147, 2011.

[11] P. Ferragina, R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proc. 18th Annual Symposium on Discrete Algorithms (SODA)*, pages 690–696, 2007.

[12] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[13] N. Prezza. A framework of dynamic data structures for string processing. In *42nd Conference on Very Important Topics (CVIT 2016)*, Article No. 23, pp. 23:1–23:12, 2017.

[14] H. E. Williams, J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[15] R. F. Rice, J. R. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communications*, 19(6):889–897, 1971.

[16] O. Plaza, M. Caniupán, R. Torres-Avilés, T. Gutiérrez-Bunster. Map algebra algorithms over raster data stored in the k2-raster compact data structure. In *2022 41st International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–4, 2022.

[17] K. Chow, D. Tzamarias, M. Hernández-Cabronero, I. Blanes, J. Serra-Sagrista. Analysis of variable-length codes for integer encoding in hyperspectral data compression with the k2-raster compact data structure. *Remote Sensing*, 12(12), 2020.

[18] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[19] J. Ziv, A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.