Proyecto Estructuras de Datos: Análisis del TAD CompactChainList

Juan Luis Guevara Isabel Sofía Adrada

Noviembre 2025

1 Introducción

Este documento presenta el análisis formal de precondiciones y postcondiciones del Tipo Abstracto de Datos (TAD) CompactChainList, implementado como parte del proyecto final de la asignatura Estructuras de Datos. Una Compact Chain List es una estructura de datos lineal que representa secuencias comprimiendo bloques consecutivos de elementos idénticos mediante pares (valor, longitud).

2 Precondiciones y Postcondiciones del TAD CompactChainList

2.1 Operaciones Constructoras

Table 1: Operaciones Constructoras

Operación	**Precondición**	**Postcondición**
`CompactChainList()`	true	$l =<> \wedge s = 0$
`CompactChainList(vector <Element> &v)`	$(v = \{\} \vee v = \{e_{n-1}, ..., e_i, e_0\}) \wedge e \in$ Element	$l = \langle\rangle$ if $v = \{\}$ $l = \langle(v_1, l_1), ..., (v_k, l_k)\rangle$ if $v = \{e_0, e_1, ..., e_{n-1}\}$ $v_i \neq v_{i+1} \forall i < k$
`CompactChainList(CompactChainList &l2)`	$(l2 =<> \vee l2 = \{e_{n-1}, ..., e_i, e_0\}) \wedge e \in$ Element	$l = \langle\rangle$ if $v = \{\}$ $l = \langle(v_1, l_1), ..., (v_k, l_k)\rangle$ if $v = \{e_0, e_1, ..., e_{n-1}\}$ $v_i \neq v_{i+1} \forall i < k$

2.2 Operaciones Analizadoras

Table 2: Operaciones Analizadoras

Operación	**Precondición**	**Postcondición**
`int size()`	$l =<> \vee l =< (v1, l1), ..., (v_i, l_i), (v_k, l_k) > \wedge v_i \neq v_{i+1} \forall i < k$	`s = 0 if l <>` $\wedge s = \sum_{(v,l) \in l} l$ `otherwise`
`int searchElement(Element e)`	$l =<> \vee l =< (v1, l1), ..., (v_i, l_i), (v_k, l_k) > \wedge v_i \neq v_{i+1} \forall i < k$	resultado = s if l <> $\vee \min\{i \mid \text{seq}[i] = e\}$
`int getConsecutive Ocurrences(vector<Element> &v)`	$l =<> \vee l =< (v1, l1), ..., (v_i, l_i), (v_k, l_k) > \wedge v_i \neq v_{i+1} \forall i < k$	resultado = s if l <> $\vee \lvert\{i \mid 0 \leq i \leq s - \lvert v \rvert \wedge \forall j \in [0, \lvert v \rvert - 1], \text{seq}[i + j] = v[j]\}\rvert$
`int getIndexFirstConsecutive Ocurrence(vector<Element> &v)`	$l =<> \vee l =< (v1, l1), ..., (v_i, l_i), (v_k, l_k) > \wedge v_i \neq v_{i+1} \forall i < k$	resultado = s if l <> $\vee \min\{i \mid 0 \leq i \leq s - \lvert v \rvert \wedge \forall j \in [0, \lvert v \rvert - 1], \text{seq}[i + j] = v[j]\}$ $\vee s$ si no se encuentra
`int getOcurrences(vector<Element> &v)`	$l =<> \vee l =< (v1, l1), ..., (v_i, l_i), (v_k, l_k) > \wedge v_i \neq v_{i+1} \forall i < k$	resultado = s if l <> $\vee \lvert\{(i_0, \ldots, i_{m-1}) \mid 0 \leq i_0 < i_1 < \cdots < i_{m-1} < s \wedge \forall j \in [0, m-1], \text{seq}[i_j] = v[j]\}\rvert$
`int getIndexFirst Ocurrence(vector<Element> &v)`	$l =<> \vee l =< (v1, l1), ..., (v_i, l_i), (v_k, l_k) > \wedge v_i \neq v_{i+1} \forall i < k$	resultado = s if l <> $\vee \min\{i_0 \mid \exists i_1, \ldots, i_{m-1}$ s.t. $0 \leq i_0 < i_1 < \cdots < i_{m-1} < s \wedge \forall j \in [0, m-1], \text{seq}[i_j] = v[j]\}$ $\vee s$ si no se encuentra
`CompactChainList getLexicographic Fusion(CompactChainList &oth)`	$l =<> \vee l =< (v_1, l_1), ..., (v_i, l_i), (v_k, l_k) > \vee \text{oth} =<> \vee \text{oth} =< (v_1, oth_1), ..., (v_i, oth_i), (v_k, oth_k) > \wedge v_i \neq v_{i+1}\ \forall i < k$	resultado =<> if $l =<> \wedge$oth =<> $\vee$ resultado = l if $l =< (v_1, l_1), ..., (v_i, l_i), (v_k, l_k) > \wedge$ oth =<> $\vee$ resultado = oth if l = <>$\wedge$oth =< $(v_1, oth_1), ..., (v_i, oth_i), (v_k, oth_k) >$ resultado = CompactChainList(merge_sorted(expand(l), expand(oth))) otherwise
`list<Element> expand()`	$l =<> \vee l =< (v1, l1), ..., (v_i, l_i), (v_k, l_k) > \wedge v_i \neq v_{i+1} \forall i < k$	resultado =< $e_1, e_2, ..., e_s$ >

)

2.3 Operaciones Modificadoras

Table 3: Operaciones Modificadoras

Operación	Precondición	Postcondición
`void set(int p, Element e)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$ $\wedge e \in$ Element	$l = l_{\text{anterior}} if \text{p} \geq s$ $l = l_{\text{anterior}}$ if seq[p] $= e$ $l = \langle (v_1, l_1), ..., (e, 1)...,$ $(\text{v}_k, l_k) \rangle$ if $l_p = 1$ $l = \langle (v_1, l_1), ..., (v_i, l_i - 1), (e, 1)...(v_k, l_k) \rangle$ if $l_p > 1 \wedge p =$ inicio_bloque(p) $l = \langle (v_1, l_1), ..., (e, 1), (v_i, l_i - 1), ..., (v_k, l_k) \rangle$ if $l_p > 1 \wedge p =$ fin_bloque(p) seq[p] $= e \wedge$ estructura compacta mantenida otherwise
`void removeFirstOcurrence(Element e)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$ $\wedge e \in$ Element	$l = l_{\text{anterior}}$ if $l =<> \vee \nexists i, \text{seq}[i] = e$ Sea (v_i, l_i) primer bloque donde $v_i = e$ $l = \langle ..., (v_i, l_i - 1), ... \rangle$ if $l_i > 1$ $l = \langle ..., (v_{i+1}, l_{i+1}), ... \rangle$ if $l_i = 1$
`void removeAllOcurrences(Element e)`	$l =<> \vee l = \langle (v_1, l_1)$ $, ..., (\text{v}_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$ $\wedge e \in$ Element	$l = l_{\text{anterior}}$ if $l =<> \vee \nexists i, \text{seq}[i] = e$ Sea $\{(v_i, l_i) \mid v_i = e\}$ el conjunto de bloques con $v_i = e$ $l = \langle ..., (v_{i-1}, l_{i-1}), (v_{i+1}, l_{i+1}), ... \rangle$ eliminando todos los bloques (v_i, l_i) donde $v_i = e$ otherwise
`void removeBlockPosition(int p)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$	$l = l_{\text{anterior}}$ if $p < 0 \vee p \geq k$ El bloque (v_p, l_p) se elimina de l $l = \langle ..., (v_{p-1}, l_{p-1}), (v_{p+1}, l_{p+1}), ... \rangle$ otherwise
`void insertElement(int p, Element e)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$ $\wedge e \in$ Element	$l = l_{\text{anterior}}$ if $p < 0 \vee p > s$ Sea i tal que $\sum_{j=1}^{i-1} l_j \leq p \leq \sum_{j=1}^{i} l_j$ $l = \langle (e, 1) \rangle$ if $l =<>$ if $p = \sum_{j=1}^{i-1} l_j \wedge v_i = e$: $l_i = l_i + 1$ if $p = \sum_{j=1}^{i-1} l_j \wedge v_i \neq e$: insertar bloque $(e, 1)$ antes de (v_i, l_i) if $p = \sum_{j=1}^{i} l_j \wedge v_i = e$: $l_i = l_i + 1$ if $p = \sum_{j=1}^{i} l_j \wedge v_i \neq e$: insertar bloque $(e, 1)$ después de (v_i, l_i) if $\sum_{j=1}^{i-1} l_j < p < \sum_{j=1}^{i} l_j$: dividir (v_i, l_i) en (v_i, p_i), $(e, 1)$, $(v_i, l_i - p_i)$ otherwise
`void modifyAllOcurrences(Element e1, Element e2)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i$ $\neq v_{i+1} \forall i < k$ $\wedge e_1 \in$ Element $\wedge e_2 \in$ Element	$l = l_{\text{anterior}}$ if $l =<> \vee \nexists i, \text{seq}[i] = e_1$ Sea $\{(v_i, l_i) \mid v_i = e_1\}$ el conjunto de bloques con $v_i = e_1$ Para cada bloque $(v_i, l_i) : v_i = e_2$ Si (v_j, l_j) y (v_{j+1}, l_{j+1}) tienen $\text{v}_j = v_{j+1}$ después del cambio $l = \langle ..., (v_j, l_j + l_{j+1}), ... \rangle$ fusionando bloques consecutivos otherwise
`void push_front(Element e, int count)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$ $\wedge e \in$ Element $\wedge$ *count* > 0	$l = \langle (e, 1) \rangle$ if $l =<>$ $l = \langle (v_1, l_1 + count), (v_2, l_2), ..., (v_k, l_k) \rangle$ if $v_1 = e$ $l = \langle (e, 1), (v_1, l_1), ..., (v_k, l_k) \rangle$ if $v_1 \neq e$ otherwise
`void push_back(Element e, int count)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$ $\wedge e \in$ Element $\wedge$ *count* > 0	$l = \langle (e, 1) \rangle$ if $l =<>$ $l = \langle (v_1, l_1), ..., (v_k, l_k + count) \rangle$ if $v_k = e$ $l = \langle (v_1, l_1), ..., (v_k, l_k), (e, 1) \rangle$ if $v_k \neq e$
`void sortVectorCCL(vector <CompactChainList> &v)`	$l =<> \vee l = \langle (v_1, l_1), ..., (v_k, l_k) \rangle$ $\wedge v_i \neq v_{i+1} \forall i < k$ $\wedge e \in$ Element $\wedge$ *count* > 0	$v = v_{\text{anterior}}$ if $v = \{\}$ Sea $v = [l_1, l_2, ..., l_n]$ ordenado lexicográficamente $\forall i < j, l_i \leq l_j$ Si $l_i = l_{i+1}$ (instancias consecutivas idénticas), se elimina la duplicada manteniendo solo una otherwise
`void lol()`		No se ha decidido la operación especial impresa

2.4 Sobrecarga de Operadores

Table 4: Sobrecarga de Operadores

Operación	**Precondición**	**Postcondición**
`CompactChainList operator+(CompactChainList &oth)`	$l =<> \vee l =< (v_1, l_1), ..., (v_i, l_i), (v_k, l_k) > \vee \text{oth} =<> \vee \text{oth} =< (v_1, oth_1), ..., (v_i, oth_i), (v_k, oth_k) > \wedge v_i \neq v_{i+1}\ \forall i < k$	resultado $=<>$ if $l =<> \wedge \text{oth} =<>$ $\vee$ resultado $= l$ if $l =< (v_1, l_1), ..., (v_i, l_i), (v_k, l_k) > \wedge \text{oth} =<>$ $\vee$ resultado $=$ oth if l $=$ $<>\wedge$oth $= < (v_1, oth_1), ..., (v_i, oth_i), (v_k, oth_k) >$ resultado = CompactChainList(merge_sorted(expand(l), expand(oth))) otherwise
`Element operator[](int indx)`	$0 \leq p \leq s$ $l =<(v_1, l_1), ..., (v_i, l_i), (v_k, l_k) > \vee \text{l} =<> \forall i < k$	resultado = seq[$indx$]
`bool operator<(CompactChainList &oth)`	$l =<> \vee l =< (v_1, l_1), ..., (v_i, l_i), (v_k, l_k) > \vee \text{oth} =<> \vee \text{oth} =< (v_1, oth_1), ..., (v_i, oth_i), (v_k, oth_k) > \wedge v_i \neq v_{i+1}\ \forall i < k$	resultado $=$ true if $\forall i\ (v_i, l_i) < (v_i, oth_i) \vee false\ otherwise$
`bool operator==(const CompactChainList &oth) const`	$l =<> \vee l =< (v_1, l_1), ..., (v_i, l_i), (v_k, l_k) > \vee \text{oth} =<> \vee \text{oth} =< (v_1, oth_1)..., (v_i, oth_i), (v_k, oth_k) > \wedge v_i \neq v_{i+1}\ \forall i < k$	resultado $=$ true if $\forall i\ (v_i, l_i) == (v_i, oth_i) \vee false\ otherwise$

3 Notación Formal

En las tablas anteriores se utilizó la siguiente notación formal:

- l: Lista de bloques $\langle (v_1, l_1), (v_2, l_2), ..., (v_k, l_k) \rangle$
- s: Tamaño total de la secuencia ($s = \sum_{i=1}^{k} l_i$)
- seq[i]: Elemento en la posición i de la secuencia expandida
- $|v|$: Cardinalidad (tamaño) del vector v
- $\emptyset$: Conjunto vacío
- $\exists$: Cuantificador existencial ("existe")
- $\forall$: Cuantificador universal ("para todo")
- $\wedge$: Conjunción lógica ("y")
- $\vee$: Disyunción lógica ("o")
- $\nexists$: No existe
- min{...}: Mínimo valor del conjunto
- $|\{...\}|$: Cardinalidad del conjunto

4 Análisis de Complejidad Computacional

4.1 Operaciones de Tiempo Constante $O(1)$

- `size()`: Retorna el valor almacenado en la variable s sin necesidad de recorrer la estructura.
- `push_front(Element e, int count)`: Agrega un bloque al inicio de la lista enlazada o incrementa la cantidad de elementos del primer bloque, operación constante en las listas de STL.
- `push_back(Element e, int count)`: Agrega un bloque al final de la lista enlazada o incrementa la cantidad de elementos del último bloque, operación constante en las listas de STL.

4.2 Operaciones de Tiempo Lineal $O(n)$

- `CompactChainList(vector<Element> &v)`: Procesa cada elemento del vector una vez para construir la representación compacta, donde n es el tamaño del vector.
- `searchElement(Element e)`: Recorre los bloques hasta encontrar el elemento buscado. En el peor caso, cuando el elemento buscado está en el último bloque o el valor buscado no está en el CCL, se deben recorrer todos los bloques, por lo tanto, n es el número de bloques del CCL.
- `expand()`: Genera la secuencia expandida bloque por bloque, donde n es el tamaño total de la secuencia.
- `modifyAllOcurrences(Element e1, Element e2)`: Recorre todos los bloques para buscar y modificar ocurrencias, donde n es el número de bloques del CCL.
- `set(int p, Element e)`: Se recorren los bloques hasta el bloque correspondiente al índice p. En el peor de los casos el índice p corresponde al último bloque, por lo tanto, n es el número de bloques.

4.3 Operaciones de Mayor Complejidad

- `getConsecutiveOcurrences(vector<Element> &v)`: $O(n \times m)$ donde n es el tamaño total de la secuencia y m es el tamaño del patrón buscado. Requiere expandir la secuencia y comparar en cada posición.
- `getLexicographicFusion()`: $O(k_1 + k_2)$ donde k_1 y k_2 son los números de bloques de las dos listas respectivamente. Opera de manera similar a un merge.

5 Decisiones de Implementación

La implementación utiliza la clase `list` de la STL de C++ para almacenar los pares (valor, longitud). Esta decisión proporciona:

- **Inserción y eliminación eficiente**: Operaciones en tiempo constante o lineal según la posición, aprovechando la estructura de lista enlazada.
- **Iteradores bidireccionales**: Permiten recorrer la estructura en ambas direcciones, facilitando operaciones complejas.
- **Manipulación eficiente de bloques**: La estructura permite dividir y fusionar bloques de manera natural.

El tamaño total s se mantiene como variable miembro para permitir operaciones de tamaño en tiempo constante, evitando recalcular el tamaño cada vez que se requiere.

6 Consideraciones de Implementación

Las precondiciones y postcondiciones presentadas se basan en la especificación formal del TAD CompactChainList. Es importante destacar que:

- Algunas operaciones marcadas como "no implementadas completamente" en el código (como `getOcurrences`, `getIndexFirstOcurrence`, `insertElement` y `sortVectorCCL`) requieren verificación adicional de su implementación final.
- La notación seq[i] se refiere a la secuencia expandida representada por la CompactChainList.
- La fusión lexicográfica sigue las reglas especificadas en el documento del proyecto: los bloques del mismo elemento se fusionan y se ordenan según el orden lexicográfico.
- Todas las operaciones mantienen la propiedad fundamental de la CompactChainList: bloques consecutivos del mismo valor están comprimidos en un solo bloque.

7 Conclusión

El análisis formal de precondiciones y postcondiciones presentado en este documento proporciona una especificación completa del comportamiento esperado del TAD CompactChainList. Esta especificación sirve como base para la verificación de corrección de la implementación y para el análisis de complejidad computacional de cada operación.

La notación formal utilizada permite una descripción precisa de los contratos de cada operación, facilitando la comprensión de las garantías que ofrece el TAD y las condiciones bajo las cuales puede ser utilizado correctamente. El TAD CompactChainList proporciona una representación eficiente para secuencias con repeticiones consecutivas, optimizando tanto el espacio de almacenamiento como el tiempo de ejecución para ciertas operaciones cuando los datos presentan homogeneidad.