Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática y en Administración y Dirección de Empresas

Trabajo Fin de Grado

Implementación Alternativa de una Tabla Hash en C++

Autor: Daniel García García
Tutor(a): Santiago Fernández Tapia

Madrid, enero 2022

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática y en Administración y Dirección de Empresas

Título: Implementación Alternativa de una Tabla Hash en C++

enero 2022

Autor: Daniel García García
Tutor: Santiago Fernández Tapia
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En el presente trabajo se analiza la problemática que presenta la estructura de datos `std::unordered_map` de la librería estándar de C++ y se proponen dos implementaciones alternativas. Acompañando a dichas implementaciones se muestra la lógica de diseño basada en los autores originales de las mismas, y las modificaciones que se han realizado sobre dicho dicho a la hora de proponer la implementación propia. En ambos casos se han realizado mediante plantillas de C++ y adaptando sus interfaces para facilitar la sustitución de la estructura estándar. Finalmente se realiza un análisis de rendimiento de la estructura más compleja de las propuestas con el fin de determinar si obtiene unos resultados razonables.

Abstract

This paper analyzes the problems presented by the std::unordered_map data structure of the C++ standard library and proposes two alternative implementations. Accompanying these implementations we show the design logic based on its original authors, and the modifications that have been made on this logic at the time of proposing our own implementation. In both cases they have been made using C++ templates and adapting their interfaces to facilitate the substitution of the standard structure. Finally, a performance analysis of the most complex of the proposed structures is carried out in order to determine whether it obtains reasonable results.

Índice general

1 Introducción

El presente trabajo consiste en el diseño, implementación y comparativa de rendimiento de una estructura de datos equivalente al `std::unordered_map` del ISO/IEC 14882:2017, estándar que define el lenguaje C++ y su librería estándar (versión C++17 en este caso). Para que la alternativa sea compatible con el estándar debe cumplir con todos los requisitos y restricciones que establece para los contenedores asociativos no ordenados. Sin embargo, en este caso se limitará a cumplir con un subconjunto de ellas con el fin de acotar el alcance del proyecto; en concreto, las relativas a las operaciones de inserción, búsqueda y borrado, dando también flexibilidad en lo relativo a su complejidad algorítmica.

El actual diseño del `std::unordered_map` realmente es una *hash table*, que utiliza un mecanismo de *hashing* para determinar la ubicación de un elemento dentro de una tabla. Para ello se asocia una clave única para cada elemento que se almacena, se aplica una función *hash* sobre su valor y se enmascara el resultado para determinar su posición en la lista. Esta secuencia de operaciones permiten obtener una complejidad algorítmica de `O(1)` cuando se trata de calcular la ubicación de un elemento en el mejor de los casos. Sin embargo, el principio de Dirichlet que afecta inherentemente a funciones que proyectan un espacio sobre otro de menor tamaño, y la calidad de dicha función (entendiendo por máxima calidad aquella que produce resultados normalmente distribuidos) garantizan la ocurrencia de colisiones que requieren de la intervención de mecanismos de resolución, que junto con las estrategias de redimensionamiento de la estructura y otros factores, provocan una reducción del rendimiento cuando se cumplen unas determinadas condiciones. El estándar establece que en el peor de los casos (colisiones) la complejidad algorítmica ha de ser `O(n)`.

Una de las implementaciones más conocidas de la librería estándar es la desarrollada por el compilador GCC (*GNU Compiler Collection*). Partiendo de la problemática anteriormente descrita que presenta una tabla *hash* a nivel teórico y la implementación concreta de GCC, se proponen diversas soluciones de diseño e implementación orientadas a la búsqueda de una mejoría en rendimiento, especialmente tratando de paliar las debilidades que presenta como el *rehashing*. Para ello el diseño de estas estructuras se emplean otras estructuras de datos subyacentes, estrategias de redimensionamiento y funciones hash diferentes a las originales. Además, conforme al nivel de abstracción de este tipo de estructuras, se han implementado en forma de *templates* de C++, permitiendo así una gran versatilidad para su uso en aplicaciones terceras.

El rendimiento de las alternativas propuestas son posteriormente evaluadas a través pruebas en diferentes escenarios que permitan determinar si hay una mejoría con respecto al `std::unordered_map`. Si bien la consecución real de dicha

mejoría no forma parte de los objetivos del proyecto, el nuevo diseño deberá estar orientado en esa dirección y tener dicha meta como el escenario ideal.

2 Análisis de la estructura unordered_map

2.1. Estándar

`std::unordered_map` es una estructura que se introdujo oficialmente en C++11, una versión del estándar ISO/IEC 14882 publicada en 2011 bajo el nombre de ISO/IEC 14882:2011 [1] que, como todas las revisiones del estándar 14882, define el lenguaje de programación C++ y su librería estándar, en la que se incluyen diversidad de estructuras y clases.

Una estas estructuras son los contenedores (*containers* en inglés), un tipo de objeto cuyo propósito es el almacenamiento de otros objetos y la asignación y liberación de memoria que requieran por medio de constructores, destructores y operaciones. Los contenedores se clasifican en tres tipos [2]:

- **Contenedores secuenciales**: almacena una secuencia lineal de objetos bajo un criterio que no tiene relación directa con el objeto a almacenar. El estándar define los siguientes: `std::array`, `std::deque`, `std::forward_list`, `std::list` y `std::vector`.
- **Contenedores asociativos**: almacenan los objetos a través de un criterio que depende directamente del valor de dicho objeto o de una clave que se asocia a él por medio de un par clave-valor. Existen dos tipos:
  - **Ordenados**: el criterio de ordenación utiliza el valor del objeto o de la clave en caso de tratarse de una estructura por clave-valor. El estándar define los siguientes: `std::map`, `std::multimap`, `std::set` y `std::multiset`.
  - **Desordenados**: en vez de utilizar el valor del objeto o de la clave, se utiliza el producto de aplicar una función hash sobre él para determinar la posición. El estándar define los siguientes: `std::unordered_map`, `std::unordered_multimap`, `std::unordered_set` y `std::unordered_multiset`.
- **Adaptadores de contenedor**: se trata de objetos que, no siendo plenamente contenedores, se limitan a definir una interfaz de interacción más restringida. De esta manera, se puede implementar un adaptador de contenedor utilizando cualquier tipo de contenedor subyacente, siempre y cuando se ajuste su interfaz para cumplir con las restricciones agregadas. El estándar define los siguientes: `std::queue`, `std::priority_queue` y `std::stack`.

Así pues, `std::unordered_map` se constituye como un contenedor asociativo desordenado. Como muchas de las estructuras especificadas en la librería estándar, está definida como una clase plantilla (*template class* en inglés) [3]. Las plantillas son una potente funcionalidad del lenguaje que permite abstraer los tipos de datos que maneja un clase o función, permitiendo implementar una clase

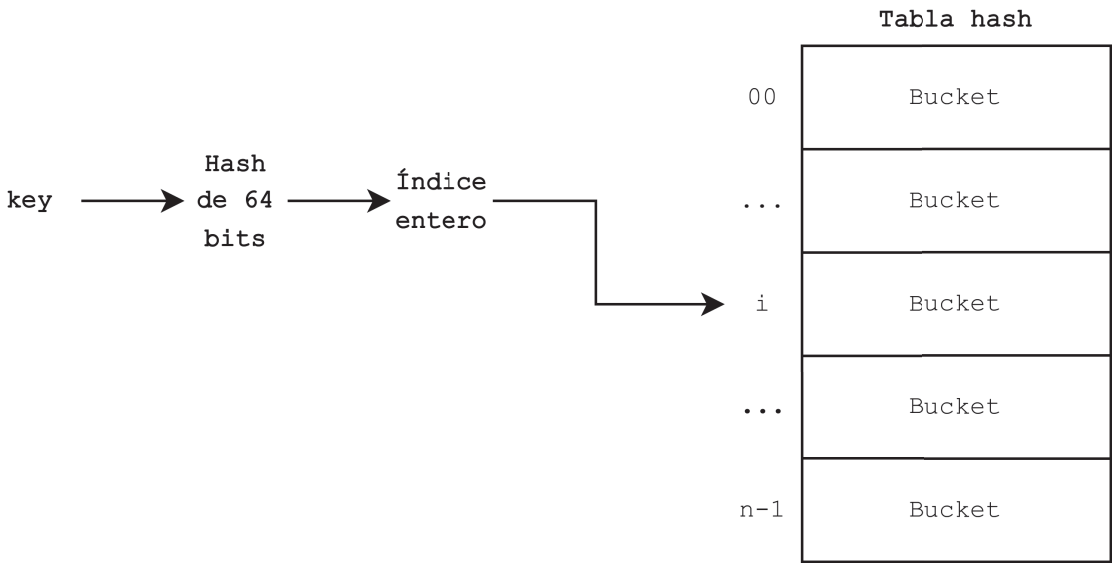
para tipos genéricos que se evalúan en tiempo de compilación. De esta manera se expone una estructura de datos que es capaz de operar con cualquier tipo de dato que se le pase sin necesidad de repetir sus métodos para cada tipo de dato posible.

El estándar establece que, como bien indica su nombre, `std::unordered_map` se trata de un mapa, por lo que permite almacenar objetos identificados de manera unívoca por una clave, que será otro objeto. De esta manera, la unidad de información de este tipo de estructuras es el par clave-valor, siendo la clave el elemento que se emplea para ubicar al valor dentro de una estructura de datos subyacente que sirva de almacenamiento. Existen numerosos tipos de mapas que emplean mecanismos diferentes para traducir la clave a una ubicación donde encontrar al valor. Como la mayoría de estándares, ISO/IEC 14882:2011 se limita a definir, dejando la implementación a los compiladores. En concreto este proyecto utiliza la proporcionada por el compilador GCC (*GNU Compiler Collection*) que se estudia a continuación.

2.2. Funcionamiento interno

La implementación de GCC de `std::unordered_map` consiste en un mapa o tabla *hash* (*hash map* o *hash table* en inglés), similar al que ofrecen otros lenguajes en sus librerías oficiales, como el `HashMap` en Java. Los *hash map* son un tipo de estructura de datos que se componen de una tabla implementada como una lista cuyos elementos almacenan los pares clave-valor. La lista tiene un tamaño fijo de n huecos denominados *buckets*, que según la implementación puede darse como parámetro de inicialización. Todos los *buckets* se cargan en memoria inicializándose como ceros, por lo que cuanto mayor sea el número de *buckets* más ocupará en memoria la estructura al inicializarse, independientemente de su uso posterior.

Para calcular en qué posición de la lista se ubica un determinado valor `std::unordered_map` aplica una función hash sobre su clave, dando lugar a un resumen (*hash digest* en inglés). Este resultado posteriormente se enmascara de tal manera que se obtenga un número entero comprendido entre 0 y $n-1$ que corresponderá con la posición i de la lista. El proceso de enmascaramiento puede realizar mediante operaciones de desplazamiento binario o simplemente operando con módulos: $i = hash(value) \mod n$. Finalmente el valor resultante se utiliza como índice en la lista y dicho valor le corresponde el *bucket* `table[i]`.



Por tanto, el proceso de convertir la clave en un índice de la lista se compone de dos pasos: calcular el hash del valor y enmascarar. El enmascaramiento es una operación eficiente y rápida que puede realizarse en la mayoría de arquitecturas en una única instrucción, por lo que la eficiencia de la función *hash* es un factor importante para el rendimiento.

En cualquier caso, el principio de Dirichlet, también conocido como el principio del palomar o de las cajas, establece un axioma fundamental que sienta las bases de las debilidades del *hash map* en todas sus implementaciones: si x elementos se colocan en y contenedores, siendo $x > y$, entonces al menos un contenedor deberá contener más de un elemento. Una función *hash* no es más que una proyección sobreyectiva de un espacio X de tamaño x a uno Y de tamaño y, $X \to Y$, siendo $x > y$ ya que admite cualquier entrada y produce salidas de tamaño fijo z, siendo $y = 2^z$. Aplicando el principio de Dirichlet se garantiza la existencia de colisiones, es decir, de situaciones en las que la función *hash* produce el mismo resultado para dos valores diferentes. Por tanto, puede darse que dos pares clave-valor cuyas claves provoquen colisión den conflicto en la tabla que debe tratarse con una estrategia de resolución colisiones.

Otra manera de resolver el problema de las colisiones es seleccionar un subconjunto de claves con un número y valor conocidos, de manera que pueda buscarse selectivamente una función *hash* que para dicho subconjunto no provoque colisiones, en cuyo caso se trata de una función *hash* perfecta. Esto naturalmente limita la cantidad de información (pares clave-valor) que puede almacenar la tabla *hash*, ya que se sigue manteniendo la restricción de que a cada valor le corresponde unívcamente una clave. El nombre "función *hash* perfecta" no debe engañar, ya que la perfección no proviene de la naturaleza de la función, sino del conocimiento *a priori* del conjunto de entrada y la elección deliberada de dicha función para dicho conjunto. La función *per se* sigue siendo una aplicación sobreyectiva, sin embargo al acotarse el dominio de claves a efectos prácticos se comporta como biyectiva. En cualquier caso, en la mayoría de ocasiones no será posible reducir el subconjunto de claves, por lo que elegir una estrategia de

resolución de colisiones adecuada es una labor capital en el diseño de la tabla.

2.2.1. Estrategias de resolución de colisiones

Las estrategias de resolución de colisiones arrojan certidumbre sobre la operativa cuando a dos valores les corresponde el mismo *bucket*. El uso de una u otra tendrá un impacto significativo en el rendimiento medio de la estructura, ya que dado el caracter fijo del tamaño de la tabla, cuantos más pares clave-valor se introduzcan más frecuentes serán las colisiones al reducirse el número de *buckets* libres. Algunos mecanismos comunes son:

- **Encadenamiento separado**: consiste en implementar el *bucket* como una estructura de datos que permita introducir más de un elemento identificable por su clave. De esta manera, en vez de directamente obtenerse el par clave-valor directamente al acceder a la tabla *hash*, se accede a esta estructura subyacente. La principal desventaja de este mecanismo es que la complejidad algortimica de la tabla *hash* quedará supeditada por la de la estructura subyacente para los casos de colisión.
- **Direccionamiento abierto**: a diferencia del encadenamiento separado, esta alternativa no utiliza estructuras de datos subyacentes, sino que el par clave-valor se sigue almacenando directamente en la tabla. Para ello se busca secuencialmente un *bucket* que esté libre, partiendo del que le corresponde al par en cuestión. El sondeo se puede realizar en en intervalos constantes (sondeo lineal), por ejemplo de uno en uno o de dos en dos; en intervalos linealmente crecientes (sondeo cuadrátrico), o empleando algún otro criterio. La principal desventaja es que el número de pares clave-valor que pueden introducirse como máximo coincide con el tamaño de la tabla, a diferencia de el encadenamiento separado que permite introducir más pares que *buckets* tiene la tabla. Además, conforme la tabla agota sus *buckets* libres, el sondeo es potencialmente más costoso.

En el caso de la implementación de GCC, se utiliza el encadenamiento separado teniendo como estructura subyacente una lista enlazada. Esta es la manera más común de tratar colisiones en *hash maps*.

2.2.2. Factor de carga y rehashing

El factor de carga es un estadístico que representa el nivel de ocupación de un *hash map*. Se obtiene como el cociente entre el número de elementos almacenados en el contenedor y el número de *buckets*. Existe una correlación entre la probabilidad de producirse una colisión y lo cercano que el factor de carga es a la unidad. Por esta razón se utiliza como un estimador de la necesidad de tomar medidas que eviten el exceso de colisiones y por ende, la degradación del rendimiento medio de las operaciones.

Se define como

$$F = \frac{k}{n} \tag{2.1}$$

donde:

- k es el número de elementos almacenados en el contenedor, y
- n es el número de *buckets*.

La implementación del *hash map* debe establecer un valor del factor de carga a partir del cual ejecutar un procedimiento para reducir la probabilidad de colisiones. El mecanismo más frecuente es un *rehashing*, que consiste en instanciar una nueva estructura de datos en memoria cuyo número de *buckets* sea superior a la anterior tabla, recalcular la ubicación de cada uno de los elementos ya almacenados y colocarlos donde les corresponde en la nueva estructura. Se trata de una operación costosa, ya que se debe aplicar la reubicación para todos y cada uno de los elementos, por lo que cuantos más elementos tiene una tabla más tiempo consumirá la operación de *rehashing*.

El del factor de carga para la ejecución del *rehashing* (llamado factor de carga máximo) es 1,00 [4] en la implementación de `std::unordered_map` estándar de C++ y 0,75 en la de `HashMap` estándar de Java, por lo que la primera es más flexible a la hora de admitir mayor grado de carga en la tabla.

2.3. Problemática de la actual implementación

Aunque las tablas *hash* generalmente presentan un buen rendimiento en comparación con otras alternativas gracias al bajo coste computacional de convertir una clave en una ubicación, presentan una serie de desventajas [5]:

- El rendimiento depende en gran medida de la calidad de la función *hash*. La calidad en este caso se mide su uniformidad, es decir, por la capacidad de la función de mapear el dominio de claves a resultados uniformemente distribuidos. Esto significa que los resúmenes producidos se generan con la misma probabilidad: $P = \frac{1}{|X|}$, donde X es el conjunto de resúmenes posibles. Si no se cumple esta propiedad entonces existirá resúmenes que tengan una probabilidad mayor a otros de ser la imagen de una clave dada.
- Aunque solo se ejecutan puntualmente, los *rehashing* pueden degradar considerablemente el rendimiento de la operación que lo haya provocado, especialmente si la estructura contiene muchos elementos.
- Presenta una baja proximidad referencial. Dado que la ubicación del elemento depende del *hash* de la clave, puede corresponderle cualquier *bucket* de la tabla, por lo que no se respeta la secuencialidad. Además, en los *buckets* donde se encuentre más de un elemento, al utilizarse una lista enlazada como estructura subyacente, los elementos no están almacenados en memoria secuencialmente como ocurriría en caso de utilizar un *array*. Esta

distribución en memoria puede provocar fallos en la caché del procesador, reduciendo en consecuencia el rendimiento de la operación.

- El tamaño inicial de la tabla tiene un impacto directo sobre el tiempo que se dedica al *rehashing*. Cuanto más grande sea, más número de inserciones serán necesarias para que el factor de carga sobrepase el máximo a partir del cual realizar el primer *rehash*. En este sentido, conocer exactamente el número máximo de elementos que van a estar insertados a la vez en la tabla puede ser ventajoso, pues bastará con inicializar la tabla a un tamaño que evite que se produzca cualquier *rehashing*.

Estos son algunos de los problemás más destacables del `std::unordered_map`, por lo que una estructura alternativa debería tratar de subsanarlos en su diseño.

3 Diseño alternativo: HAMT

3.1. Diseño

3.1.1. Introducción

La estructura *Hashed Array Mapped Retrievable Tree* o *Hashed Array Mapped Trie* fue descrita por Phil Bawgell en su artículo *Ideal Hash Trees* en noviembre de 2001 [6]. Se trata de una evolución de la estructura *Array Mapped Tries* (AMT) descrita en su artículo previo *Fast And Space Efficient Trie Searches* de junio del año 2000. El objetivo de esta estructura de datos es la de ofrecer una implementación de contenedor asociativo desordenado, similar a la funcionalidad que ofrece el `std::unordered_map`, combinando las propiedades del *hashing* y los árboles de tipo *Trie*, en concreto empleando como estructura subyacente un AMT.

Se trata de una estructura más compleja que un *hash map* tradicional, pero ofrece unas ventajas importantes como contraprestación: el uso eficiente de memoria, el crecimiento dinámico de tamaño (y por tanto, la ausencia de *rehashing*), y rendimientos similares al *hash map*. Por ello, y aunque sea una alternativa más reciente, varios lenguajes de programación la han adoptado como la implementación estándar de sus contenedores asociativos desordenados. Por ejemplo, la estructura `PersistenHashMap` de Clojure implementa una variación inmutable del HAMT propuesto por Bagwell [7]. Erlang también utiliza un HAMT persistente para representar intermanete mapas de gran tamaño gracias al uso eficiente de memoria que ofrece [8]. También es común encontrar implementaciones de esta estructura en otros lenguajes aunque no sea en sus librerías estándares.

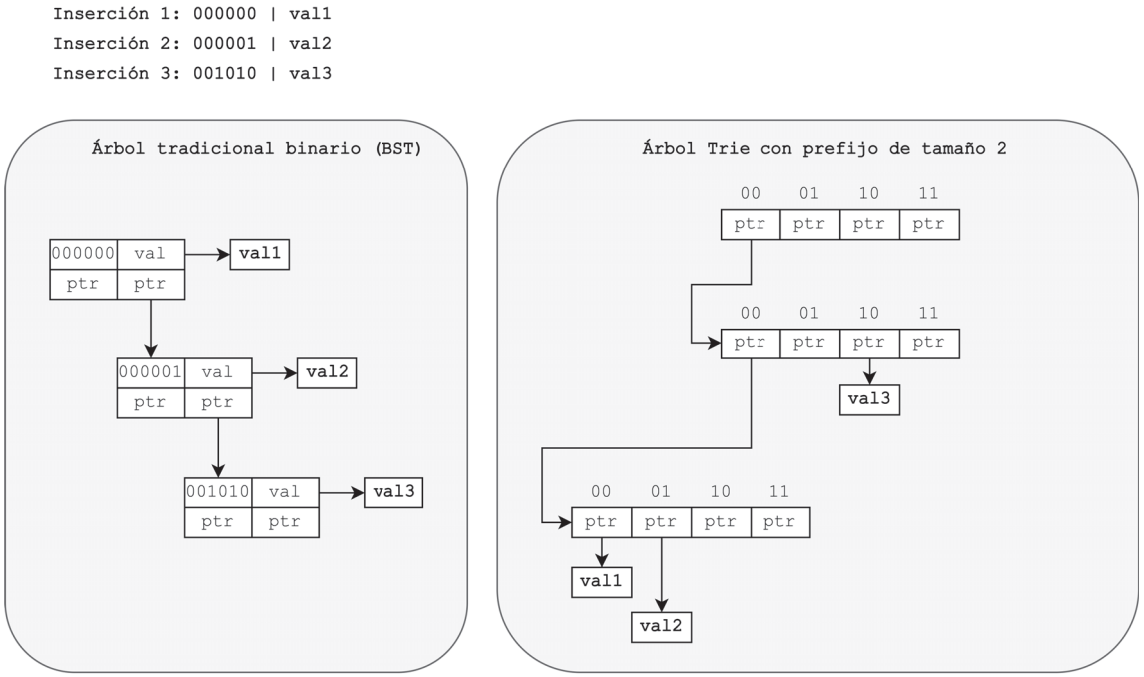
3.1.2. Array Mapped Trie (AMT)

Como se ha comentado anteriormente, el HAMT es una combinación del uso de *hashes* (H) y la estructura subyecente *Array Mapped Trie* (AMT). Por tanto, la segunda conforma el núcleo del diseño del HAMT. El *Array Mapped Trie* es una implementación de árbol *Trie* utilizando *arrays* y *bitmaps* para optimizar el espacio en memoria. Antes de explicar su funcionamiento conviene tratar los árboles *Trie* y qué problemas presenta.

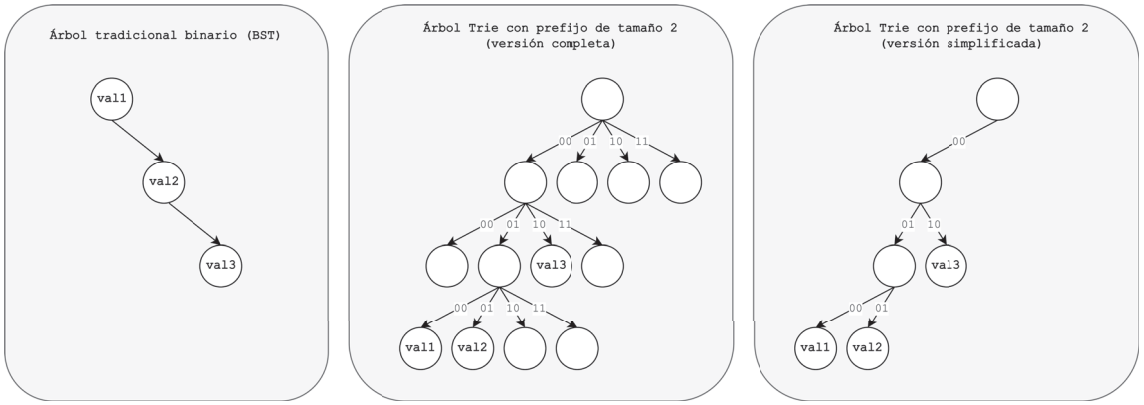
Los árboles *Trie* (contracción del inglés *re**trie**vable tree*), también conocidos como *prefix trees*, son un tipo particular de árbol que en vez de visitar los nodos comparando la totalidad de la clave almacenada con la dada, utiliza solo un fragmento de la misma denominado prefijo, por lo que únicamente los nodos hoja almacenan la información mientras que el resto actúan de caminos hacia ellos.

Cada nivel que se avanza por el árbol se consume un número n de símbolos de la clave para construir el prefijo, que es utilizado para determinar por qué rama continuar la búsqueda.

Existen numerosas maneras de implementar un *Trie*, siendo la más básica la que, dado un nodo, representa a sus hijos mediante punteros contiguos en un *array* de tamaño igual al número de combinaciones posibles que puede tomar el prefijo. Siendo Σ el alfabeto de la clave, esto significa que cada nodo tendrá $m = |\Sigma|^n$ nodos hijo, dando lugar a *arrays* de tamaño m. Por ejemplo, si el alfabeto de la clave es el inglés (26 símbolos) y el tamaño de prefijo es 2, cada nivel consumirá las dos siguientes letras y cada nodo dispondrá de un puntero que permita visitar el *array* de sus $26^2 = 676$ hijos.



La figura muestra un ejemplo de la inserción, tanto en un árbol BST como en un Trie básico con prefijo de tamaño 2, de las claves `000000`, `000001` y `000010` codificadas como cadenas de caracteres con alfabeto $\Sigma \equiv \{0, 1\}$ y tamaño fijo $t = 6$. Como puede observarse en el *trie*, el primer *array* representa a los hijos del nodo raíz (no representado en memoria). Las combinaciones posibles del prefijo son `00`, `01`, `10` y `11`. Las tres claves empiezan por el prefijo `00`, por lo que el nodo raíz solo tendrá un hijo. Para el segundo nivel las dos primeras claves comparten el mismo prefijo: `01`, y la tercera presenta uno diferente: `10`. Dado que no existía previamente valores almacenados cuyas claves comiencen por `0010`, el valor de la tercera clave se inserta directamente en el segundo nivel para ahorrar un *arrays* de nodos hijo. Avanzando al tercer nivel, las posiciones donde insertar el valor se encuentran en los prefijos que corresponden a los últimos dos caracteres de las claves: `00` y `01` respectivamente. En la figura puede observarse una representación más abstracta de este caso con ambas estructuras de datos.



El problema que se presenta es bastante claro: disponer de nodos como listas de tamaño m significa tener una cantidad inasumible de punteros con valor `null` que consumen memoria sin ser útiles, especialmente cuando el alfabeto contiene numerosos símbolos (*arrays* muy grandes) y la clave es relativamente larga comparado con el tamaño del prefijo (muchos niveles). En un *trie* vacío cuyo alfabeto sea el inglés, el tamaño del prefijo sea 2, la clave "`trabajos`", y el recorrido hacia la clave implique siempre una nueva lista de nodos hijo, se requiere al menos $4 \cdot 26^2 = 2704$ punteros inicializados.

Para resolver este problema es necesario compactar los punteros `null`. Uno de los diseño*Ideal Hash Trees*s para lograrlo es el *Array Mapped Trie*, propuesto por Bagwell en su artículo *Fast And Space Efficient Trie Searches* [9], en que también explora otras alternativas tales como el *Ternary Search Tree* (TST) o el *Array Compacted Tre*e (ACT).

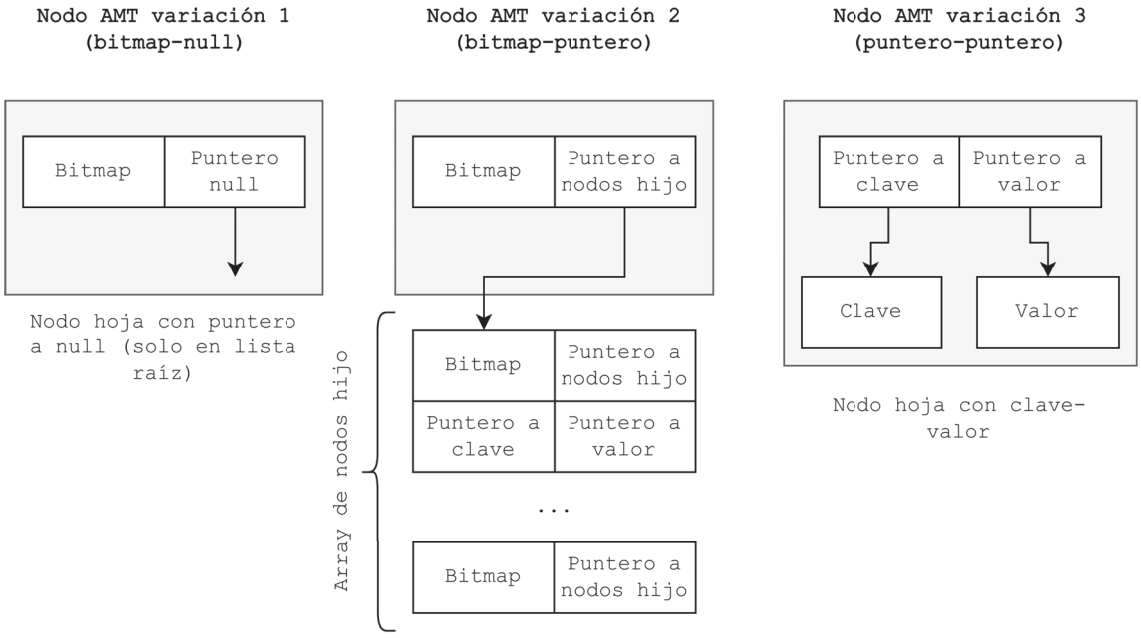
El AMT respeta la representación de los nodos hijo como un *array*, por lo que su información se encuentra contigua en memoria. Los hijos se dispondrán en el *array* en función del orden de su prefijo. Se parte de una lista que representa a los nodos hijo de la raíz, y será la única lista en mantener un tamaño constante igual al de la cardinalidad del alfabeto, siendo por tanto la única en contener punteros `null`. El resto de listas contenidas en el AMT solo dispondrán de punteros a siguientes listas o a valores almacenados, sin punteros `null`. Para que esto sea posible, los nodos se componen de dos elementos, dando como resultado los siguientes tres tipos de nodos:

- **Nodo *bitmap-null***: se trata de la primera de las tres formas que puede tomar un nodo AMT. Su primer elemento es un *bitmap* y el segundo es un valor `null`. El *bitmap* contendrá tantos bits como combinaciones posibles puede dar el prefijo, y a cada una le corresponde el bit, realizando la asociación en orden. Por tanto, estos bits representan a cada uno de los nodos hijo que teóricamente puede tener. Si un bit vale 0 significa que no existe el nodo hijo para el prefijo asociado al bit, mientras que si su valor es 1 entonces existe y se puede seguir explorando el árbol por esa rama. Por ejemplo, el alfabeto inglés consta de 26 letras, por lo que el *bitmap* de cada nodo se compondrá de 26 bits. Suponiendo un AMT con tamaño de prefijo 1, la letra `c` siempre le corresponderá el bit de posición 2 porque es la tercera letra del alfabeto. El segundo elemento es un valor `null`, lo que

significa que el nodo no tiene hijos. Por esta razón, todos los bits del *bitmap* estarán a 0. En definitiva, se trata de un nodo "vacío", y este tipo de nodos sólo pueden aparecer en la tabla raíz, que es la única que tiene todos los hijos de la raíz inicializados. Siguiendo el anterior ejemplo, si dada la lista raíz, el nodo de posición $i = 2$ tiene como primer elemento un *bitmap* a 0 y su puntero a `null`, entonces no existe ningún par clave-valor almacenado en el AMT cuya clave comience por la `c`.

- **Nodo *bitmap*-puntero a tabla**: la segunda forma de nodo AMT tiene como primer elemento un *bitmap*, que se utiliza con el mismo propósito que en el tipo de nodo anterior: representa la existencia de hijos. Acompañándolo se encuentra el segundo elemento, un puntero a la tabla de hijos, que tendrá tantos como bits a 1 haya en el *bitmap*.
- **Nodo puntero a clave-puntero a valor**: se trata de la forma que adoptan los nodos hoja y, dado que es un *trie*, serán los que contengan información.

La figura y tabla muestran estas tres posibles formas. Puede observarse como la segunda forma es la que tiene un puntero que permite visitar la lista de sus nodos hijo. Para acceder a ellos se utilizan unas técnicas con los bits del *bitmap* que se explicará a continuación.



Las combinaciones posibles quedan de la siguiente manera:

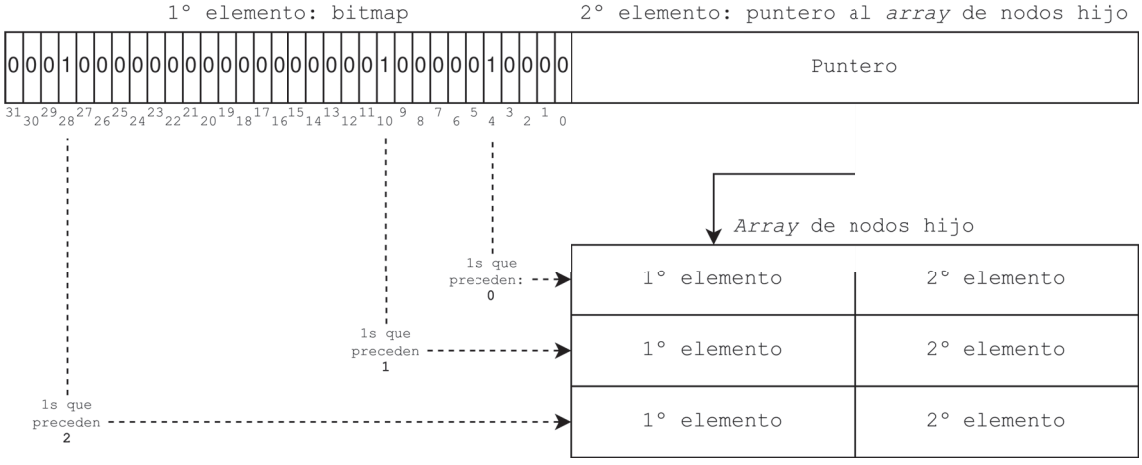
Primer elemento	Segundo elemento
Bitmap	Puntero a `null`
	Puntero a lista de nodos hijo
Puntero a clave	Puntero a valor

Utilizando estos nodos se van enlazando listas de nodos hijo que estarán compactadas, en el sentido de que sólo contendrán información útil y no punteros

`null`. Esto permite reducir su coste de memoria a 1 bit, quedando resuelto el problema planteado al comienzo del apartado.

Conociendo la estructura de los nodos queda por resolver la incógnita de cómo convertir la información mostrada en el *bitmap* en un índice que permita acceder al elemento correspondiente de cada bit en tabla de nodos hijo. Como cada hijo existente está represetado por un 1 en su bit correspondiente bastará con contar el número de 1s que preceden al bit del prefijo en cuestión. Dicha cuenta será el índice de la tabla. En muchas arquitecturas esta operación puede realizarse directamente con una única instrucción de tipo *popcount* (*population count*) obteniendo una mejora de rendimiento en comparación a realizar la suma manualmente. En la microarquitectura Intel Core forma parte del juego de instrucciones SSE4 y se denomina `POPCNT` [10].

En la siguiente figura se muestra un ejemplo de está dinámica. Se trata de un nodo perteneciente a un AMT en el que los prefijos pueden dar lugar a 32 combinaciones diferentes, habiendo por tanto 32 bits para representarlas. Como puede observarse, los bits 4, 10 y 28 están a 1, mientras que el resto está a 0. Esto significa que el presente nodo tiene tres hijos, lo cual se refleja en el número de elementos del *array* de nodos hijo. Para calcular el índice de la tabla correspondiente al prefijo número 28 (en cuanto a ordenación) se accede al bit de su mismo orden y se cuenta con una instrucción `POPCNT` el número de 1s que le preceden. En este caso son 2: los de la posición 4 y 10. Por tanto, el elemento es accesible utilizando el índice 2.



3.1.3. Hash Array Mapped Trie (HAMT)

La segunda pieza del HAMT es la introducir el uso del *hashing* en la estructura AMT subyacente para obtener un efecto similar al del `std::unordered_map`: almacenar el valor en función del *hash* de la clave. Esto, además de acotar la longitud de la clave a un tamaño constante, permite abstraer el conocimiento de cuál es su alfabeto, ya que el HAMT siempre operará con *hashes*, cuyo alfabeto es siempre $\Sigma \equiv \{0, 1\}$. En ocasiones interesa utilizar como clave un objeto diferente a una cadena de caracteres, por lo que hacer transparente el alfabeto es una propiedad valiosa de cara a implementar la estructura utilizando genéricos,

pudiéndose así emplear para cualquier tipo de dato. Utilizar *hashes* en vez de la clave inalterada introduce la posibilidad de colisiones, por lo que es necesario disponer de mecanismos de resolución. Es conveniente alinear el tamaño de algunas estructuras como los *bitmaps* a la arquitectura del computador sobre la que se implemente, ya que de esta manera ocuparán exactamente una palabra.

El primer paso para el uso del HAMT es calcular el *hash* de la clave, dando lugar a una secuencia binaria de h bits. A continuación, tal y como se ha visto en el funcionamiento del AMT, es necesario determinar el tamaño de los prefijos. Estas decisiones dan lugar a oportunidades de optimizar el uso de memoria (por alineamiento) y de rendimiento (por menos llamadas a memoria y aprovechamiento de estrategias de caché):

- En primer lugar, se puede elegir un tamaño de hash h tal que ocupe exactamente una palabra. Para ello bastará con igualar h al tamaño de palabra del computador. Por ejemplo, para una arquitectura de 32 bits, $h = 32$.
- En segundo lugar, se puede elegir un tamaño de prefijo t tal que el número de sus posibles combinaciones sea igual al tamaño de palabra del computador. Dado que el número de bits de los *bitmap* depende de este número de combinaciones, esta estrategia permitiría que cupiesen exactamente en una palabra. Por ejemplo, para una arquitectura de 32 bits, $t = 5$, ya que cada prefijo tendrá $2^5 = 32$ posibles combinaciones, dando lugar a *bitmaps* de 32 bits. Esto además significa que cada nodo tendrá como máximo 32 hijos, representados en *arrays* de 32 elementos.

Ambas estrategias se proponen en el propio artículo de Bagwell como formas de optimización de memoria y rendimiento.

3.1.3.1. Búsqueda

La búsqueda comienza aplicando la operación de *hash* sobre la clave dada. Se toman los t bits más significativos del *hash* para construir el prefijo y se convierte a un número entero. El resultado se utiliza como índice para acceder a la tabla raíz. Al hacerlo, pueden darse tres casos:

- La entrada está vacía (tanto el *bitmap* como el puntero están a cero). Esto significa que la clave no existe en la estructura de datos.
- La entrada tiene contenido y se trata de un par clave-valor, donde los primeros 32 bits corresponden al puntero de la clave y los siguientes 32 al puntero del valor.
- La entrada tiene contenido y se trata de un par *bitmap*-puntero, donde los primeros 32 bits corresponden al *bitmap* y los siguientes 32 al puntero de la tabla de hijos del nodo visitado.

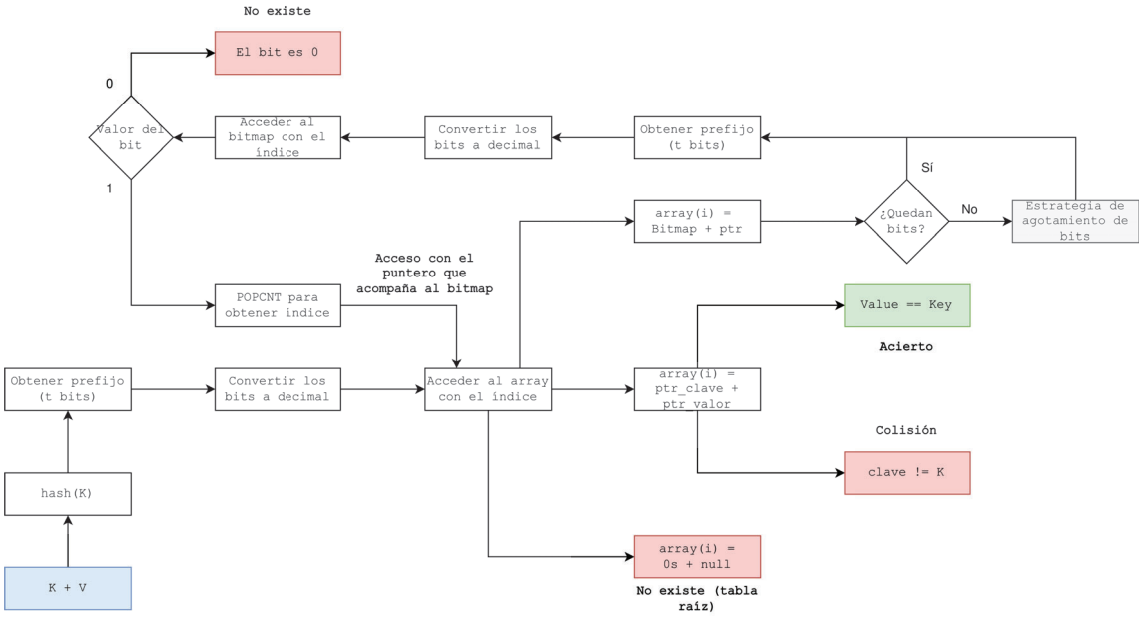
La distinción entre los dos tipos de entrada con contenido se hace a través de un *bit flag*, sin embargo, la implementación propuesta en este trabajo aborda este problema de otra manera, como se explicará más adelante. En caso de que el contenido sea un par *bitmap*-puntero, es necesario realizar el salto a la siguiente

tabla. Para ello se toman los siguientes t bits del *hash* para construir el siguiente prefijo, se convierte a un entero y se utiliza como índice para acceder al *bitmap* para obtener el valor de bit que le corresponde. Pueden darse dos situaciones:

- Si el valor es 0 no existe un nodo hijo para ese prefijo, lo que se traduce en que la clave no existe en la estructura de datos.
- Si el valor es 1, entonces se realiza una operación de conteo de bits precedentes al bit leído (a través de la instrucción `POPCNT` o equivalente). El resultado del conteo se utiliza como índice para la tabla a la que apunta el puntero que acompaña el *bitmap*. Este proceso se repite hasta que la búsqueda concluye con un error por no existencia de la clave, ya sea porque se ha encontradon un bit a 0 en algún *bitmap* visitado o porque se ha encontradon un par clave-valor pero la clave no coincide con la dada, o bien con éxito por encontrarla.

Nótese cómo la clave inalterada y completa solo se compara cuando se ha hallado un par clave-valor en la posición que le corresponde y se desea verificar que la clave de dicho par coincide con la clave dada, es decir, no se trata de una colisión.

Algoritmo de búsqueda



3.1.3.2. Inserción

La inserción sigue el mismo algoritmo que la búsqueda y diverge al darse alguna de las siguientes situaciones:

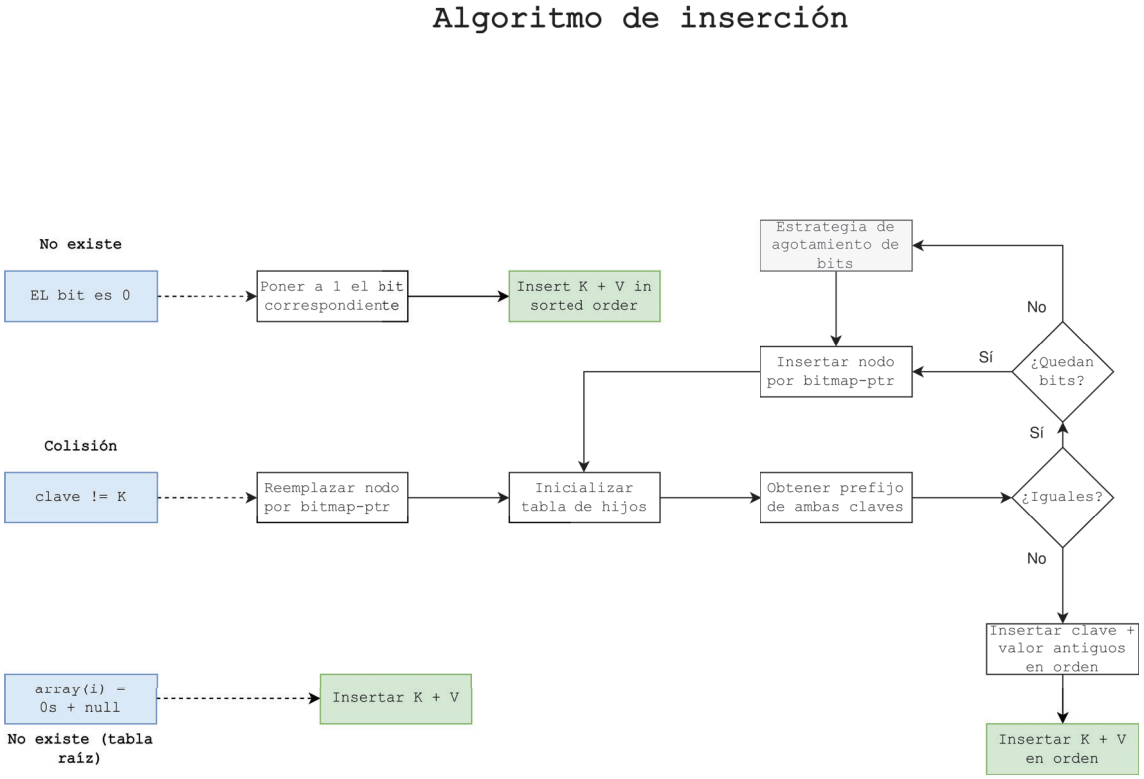
- **El nodo está vacío (*bitmap* a 0 y puntero a `null`)**. Como se ha visto en el algoritmo de búsqueda esto puede darse únicamente en la tabla raíz. La inserción en este caso consiste en sustituir los valores nulos de la entrada

por punteros a la clave y al valor.

- **La entrada tiene contenido y se trata de un par clave-valor**. Esto significa que se ha producido una colisión: el *hash* de la clave almacenada y el *hash* de la clave a almacenar comparten el mismo prefijo para el actual recorrido del árbol. Para resolver la colisión se consume el siguiente prefijo de ambas claves y se comprueba si vuelven a ser iguales. En caso de serlo, se ha producido una colisión nuevamente. Para reflejar esta situación se reemplaza el par clave-valor almacenado por un par *bitmap*-puntero cuyo puntero apunte a una nueva tabla de nodos hijo, poniendo a 1 el bit que corresponde al nuevo prefijo colisionante. En esta nueva tabla se tratará de resolver la colisión consumiendo y comparando el siguiente prefijo de ambas claves. Si la colisión persiste entonces el proceso se repite, anidando tablas de hijos de un elemento, los *bitmap*-puntero. Cada prefijo que se consuma reduce la probabilidad de persistencia de la colisión en un factor de $\frac{1}{32}$. En caso de desaparecer la colisión, basta con añadir a la nueva tabla los dos pares clave-valor colisionantes, respetando siempre el orden de las claves.

- **La entrada tiene contenido, se trata de un par *bitmap*-puntero y el bit que corresponde al prefijo tiene valor 0**. Esto significa que la inserción puede realizarse dando como resultado un nuevo nodo hoja del árbol. Para ello, se pone a 1 el valor del bit correspondiente y se añade a la tabla a la que apunta el puntero una entrada de tipo par clave-valor en la posición que le corresponda.

Puede darse el caso de que el bucle de resolución de colisiones indicado en el segundo punto de lugar al agotamiento del *hash* habiendo consumido todos sus prefijos, en cuyo caso se trata de una colisión completa a nivel de función *hash*, no a nivel de prefijo: $h_1(k_1) = h_2(k_2)$. En este caso es necesario establecer algún mecanismo para generar nuevos *hashes* que permitan diferenciar a ambas claves. El artículo no entra en detalle sobre la implementación de este mecanismo.



3.1.3.3. Borrado

El borrado, análogamente a la inserción, sigue el mismo algoritmo que la búsqueda hasta encontrar el par clave-valor cuya clave coincida con la dada. En este caso, bastará con destruir dicha entrada en la tabla donde reside y poner a 0 el bit que le corresponde en el *bitmap* del nodo padre. Si la tabla únicamente contenía esa entrada, entonces la tabla también se destruye, eliminando consigo el nodo padre. Esta destrucción se aplica retroactivamente mientras el caso se repita.

3.2. Implementación propia

3.2.1. Particularidades

La implementación propia se ha realizado en el lenguaje C++, concretamente en su versión 20 debido a que es la primera que dispone de la función `std::popcount()` en la librería estándar, facilitando la operación de conteo de bits. Además, se ha utilizado gcc, la implementación de C++ libre y de código abierto perteneciente al proyecto GNU.

Dado que la mayoría de computadores modernos disponen de una arquitectura de 64 bits, se ha adaptado el diseño del HAMT a dicha arquitectura, con *hashes* y estructuras de 64 bits y $t = 6$, ya que $2^6 = 64$. Esto tiene un claro beneficio: reduce la probabilidad de colisiones a nivel de función *hash* notablemente, ya

que el duplicar el número de dígitos de la cadena se amplía el espacio de proyección posible en un factor de $\frac{2^{64}}{2^{32}} = 4,29$ billones de veces. Suponiendo que la función produce resultados uniformemente distribuidos (calidad perfecta) esto se traduce en una probabilidad de colisión a nivel de función de $\frac{1}{2^{64}}$.

Para dar forma a los diferentes elementos internos del HAMT se han utilizado diversas estructuras de datos provistas por la libería estándar, facilitando así la programación y el manejo de las mismas. En concreto, destacan:

- `std::vector`: permite almacenar elementos como un *array*, garantizando la contigüidad en memoria de sus elementos, pero permite el crecimiento dinámico de la estructura. Dado que se instancia con un tamaño inicial, si el contenedor crece más allá de dicho tamaño es necesario instanciar un nuevo *array* subyacente con el tamaño objetivo y recolocar los elementos. Es importante tener en cuenta este aspecto porque la operación de alterar el tamaño de un vector es costosa en términos de ciclos de procesador [11]. El vector ha sido utilizado como estructura de las tablas.

- `std::bitset`: la librería estándar provee una estructura de mapa de bits nativa que almacena los bits de manera eficiente. Naturalmente se ha utilizado esta estructura para los mapas de bits necesarios en los pares *bitmap*-puntero, facilitando su manejo gracias a las diferentes operaciones a nivel de bit que la clase pone a disposición del programador.

- `std::pair`: permite almacenar dos objetos en forma de par, ofreciendo constructores y operaciones sobre dichos elementos, que se acceden a través de los atributos `first` y `second`. Las entradas de las tablas se han programado utilizando esta estructura.

- `std::list`: finalmente la lista se ha utilizado para crear una pila FIFO sobre la que ir añadiendo los diferentes nodos por los cuales el algoritmo de búsqueda viaja a la hora de realizar una operación de borrado. De esta manera se construye un camino ordenado desde la raíz al nodo que procede borrarse. Este camino será posteriormente utilizado para recorrerse a la inversa, desde el nodo a destruir hasta la raíz, para ir destruyendo tablas o modificando *bitmaps* en función de las necesidades de la operación de borrado.

Uno de los aspectos más alterados del diseño original a la hora de implementar el HAMT es el tratamiento del *bit flag* que se utiliza para distinguir entre un par clave-valor y *bitmap*-puntero. En vez de ello, el problema se ha abordado interpretando a las entradas de las tablas siempre como pares *bitmap*-puntero, por lo que las entradas *null-null* y clave-valor dejan de existir. Cuando se evalúa el contenido de una entrada, primero se comprueba el *bitmap* y acto seguido, si procede, el puntero:

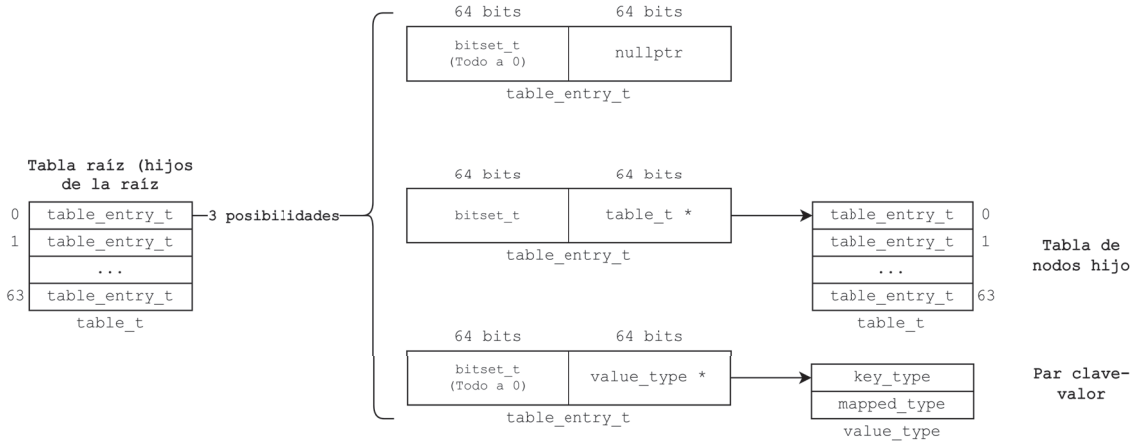
- Si el *bitmap* tiene todos sus bits a 0:

  - Si el puntero es un `nullptr` (cero): entonces se trata de una entrada vacía de la tabla raíz.

- Si el puntero es una dirección de memoria: dicho puntero está apuntando a un par puntero_clave-puntero_valor, que apuntan a la clave y al valor respectivamente. Nótese que esto implica añadir un salto más que el diseño original del HAMT, en el que la entrada era directamente puntero_clave-puntero_valor.

- Si el *bitmap* tiene algún bit a 1: el puntero asociado apunta a una tabla que representa los hijos del presente nodo. Por tanto, el *bitmap* representa aquellos nodos hijo con valor en la tabla siguiente.

En la siguiente figura se muestra esta adaptación con los nombres de las clases utilizadas en el código.



Además de lo anterior, se han desarrollado algunas funciones complementarias como `count()` y `size()`, que indican si existe un par clave-valor con la clave dada y el número de elementos almacenados, respectivamente. El propósito de ello es tratar de ampliar el subconjunto de operaciones que el HAMT propio ofrece en comparación con el `std::unordered_map`, de manera que esta interfaz pudiese ser transparente a la estructura subyacente, de manera que el programador pueda usar cualquiera de las dos sin alterar las llamadas a métodos y su comportamiento esperado. Esta ventaja precisamente se explota con el *benchmark*, pudiendo hacer las mismas llamadas a las diferentes estructuras independientemente de cuál se use.

Finalmente destacar que no se han implementado las mejoras que propone Phil Bagwell en su artículo original. Estas mejoras están orientadas a optimizar los accesos a memoria, aprovechando la caché y reduciendo los tiempos de operaciones en memoria.

3.2.2. Código fuente

```
#include <vector>
#include <bitset>
#include <bit>
#include <stdexcept>
#include <list>

#define PREFIX_SIZE 6
#define BASE_SIZE 64
```

```
enum situations{EMPTY_ROOT, NOT_EMPTY, EMPTY_SUBTABLE};
enum operations{INSERT, SEARCH, ERASE};

template<
        class Key,
                class T,
                class Hash = std::hash<Key>,
                class Pred = std::equal_to<Key>,
                class Alloc = std::allocator<std::pair<const Key, T>>>
class hamt {

public:
        typedef Key key_type;
        typedef T mapped_type;
        typedef Hash hasher;
        typedef Pred key_equal;
        typedef Alloc allocator_type;
        typedef size_t size_type;
        typedef std::pair<const key_type, mapped_type> value_type;

protected:
        typedef std::pair<std::bitset<BASE_SIZE>, void *> table_entry_t;
        typedef std::vector<table_entry_t> table_t;
        typedef std::bitset<sizeof(size_type) * 8> bitset_t;

public:

        hamt() {
                root_table_ptr = new table_t(BASE_SIZE);
                locate_stack = new std::list<locate_stack_log>();
                elements = 0;
        }

        mapped_type &at(const key_type &key) {
                locate_result target = locate(key, SEARCH);
                switch (target.situation) {
                        case NOT_EMPTY:
                        {
                                return static_cast<value_type *>(target.entry_ptr->second)->second;
                        }
                        default:
                        {
                                throw std::out_of_range("Not_found");
                        }
                }
        }

        bool insert(const value_type &value) {
                auto *new_pair_ptr = new value_type(value);
                locate_result locate_result = locate(value.first, INSERT);
                switch (locate_result.situation) {
                        // Insertion into the root table. No new further tables must be allocated.
                        case EMPTY_ROOT:
                        {
                                locate_result.entry_ptr->second = new_pair_ptr;
                                break;
                        }

                        // Collision insertion.
                        case NOT_EMPTY:
                        {
                                // Saves the old values pointer to reinsert afterwards
                                auto *old_pair_ptr = static_cast<value_type *>(locate_result.entry_ptr->second);

                                if (is_equals(old_pair_ptr->first, value.first)) {
                                        return false;
                                }

                                // Hashes the old key, iterates the production of tokens as many times as the new key
                                      has so both tokens are at the same position in the hash.
                                bitset_t hashed_old_key = hash(static_cast<value_type *>(locate_result.entry_ptr->
                                    second)->first);
                                for (int i = 0; i < locate_result.tokens_consumed + 1; ++i) { // tokens_consumed:
                                     numer of tokens at the left of the token that produced the collision.
                                        hashed_old_key = hashed_old_key << PREFIX_SIZE;
                                }

                                auto *current_entry_ptr = locate_result.entry_ptr;
                                int new_key_index, old_key_index = locate_result.entry_bitset_index;
                                bool conti = true;
                                while (conti) {

                                        old_key_index = (int) (hashed_old_key >> (sizeof(size_type) * 8 - PREFIX_SIZE)
                                            ).to_ulong();
                                        hashed_old_key = hashed_old_key << PREFIX_SIZE;
                                        new_key_index = (int) (locate_result.hashed_key >> (sizeof(size_type) * 8 -
                                            PREFIX_SIZE)).to_ulong();
                                        locate_result.hashed_key = locate_result.hashed_key << PREFIX_SIZE;
                                        if (new_key_index == old_key_index) {
                                                auto *new_table_ptr = new table_t(1);
                                                current_entry_ptr->second = new_table_ptr;
```

```
                                        // Replaces the entry bitset with a real bitset with the correct bits
                                             set, points the entry's second value pointer to nullptr.
                                        current_entry_ptr->first.set(old_key_index);
                                        current_entry_ptr = &new_table_ptr->at(0);
                                } else {
                                        auto *new_table_ptr = new table_t(2);
                                        current_entry_ptr->second = new_table_ptr;
                                        current_entry_ptr->first.set(old_key_index);
                                        current_entry_ptr->first.set(new_key_index);
                                        if (old_key_index < new_key_index) {
                                                new_table_ptr->at(0).second = old_pair_ptr;
                                                new_table_ptr->at(1).second = new_pair_ptr;
                                        } else {
                                                new_table_ptr->at(0).second = new_pair_ptr;
                                                new_table_ptr->at(1).second = old_pair_ptr;
                                        }
                                        conti = false;
                                }
                        }
                        break;
                }

                // Inserton into a sub-hash table. The table must be resized.
                case EMPTY_SUBTABLE:
                {
                        locate_result.entry_ptr->first.set(locate_result.entry_bitset_index);
                        table_entry_t new_entry;
                        new_entry.second = new_pair_ptr;
                        auto &table = *locate_result.table_ptr;
                        auto table_index = locate_result.table_index;
                        if (table.size() > table_index && table.at(table_index).first.none() && table.at(
                              table_index).second == nullptr) {
                                table.at(1) = new_entry;
                        } else {
                                table.insert(table.begin() + table_index, new_entry);
                        }
                        break;
                }
        }
        ++elements;
        return true;
}

size_type erase(const key_type &key) {
        locate_result target = locate(key, ERASE);
        switch (target.situation) {
                case NOT_EMPTY:
                {
                        auto info = locate_stack->back();
                        auto value_pair = static_cast<value_type *>(info.entry_ptr->second);

                        if (is_equals(value_pair->first, key)) {
                                auto table_ptr = info.table_ptr;
                                auto entry_ptr = info.entry_ptr;
                                auto entry_bitset_index = info.entry_bitset_index;
                                auto table_index = info.table_index;

                                delete value_pair;
                                if (table_ptr->size() == 1 && table_ptr != root_table_ptr) {
                                        bool cont = true;
                                        while (cont) {
                                                delete table_ptr;
                                                locate_stack->pop_back();
                                                info = locate_stack->back();
                                                table_ptr = info.table_ptr;
                                                entry_ptr = info.entry_ptr;
                                                entry_bitset_index = info.entry_bitset_index;
                                                table_index = info.table_index;
                                                if (info.table_ptr->size() != 1) {
                                                        cont = false;
                                                }
                                        }
                                }
                                if (table_ptr == root_table_ptr) {
                                        entry_ptr->first.set(entry_bitset_index, 0);
                                        entry_ptr->second = nullptr;
                                } else {
                                        table_ptr->erase(table_ptr->begin() + table_index);
                                        locate_stack->pop_back();
                                        info = locate_stack->back();
                                        info.entry_ptr->first.set(entry_bitset_index, 0);
                                }
                                --elements;
                                return true;
                        }
                }

                default:
                {
                        return false;
                }
```

```
            }
        }

        size_type count(const key_type &key) {
            locate_result target = locate(key, SEARCH);
            return (target.situation == NOT_EMPTY && is_equals(static_cast<value_type *>(target.entry_ptr->second)
                ->first, key));
        }

        size_type size() const {
            return elements;
        }

private:
        table_t *root_table_ptr;
        size_type elements;
        hasher hash;
        key_equal is_equals;

        struct locate_stack_log {
            table_t *table_ptr;
            table_entry_t *entry_ptr;
            unsigned int table_index;
            unsigned int entry_bitset_index;
        };
        std::list<locate_stack_log> *locate_stack;

        struct locate_result {
            unsigned int situation;
            table_t *table_ptr;
            table_entry_t *entry_ptr;
            unsigned int table_index;
            unsigned int entry_bitset_index;
            unsigned int tokens_consumed;
            bitset_t hashed_key;

        };

        /*
         * Returns the precise location where the key-value pair is supposed to be.
         */
        locate_result locate(const key_type &key, unsigned int operation) {

            locate_stack->clear();

            bitset_t hashed_key = hash(key);
            unsigned int tokens_consumed = 0, table_index, entry_bitset_index = (hashed_key >> (sizeof(size_type)
                * 8 - PREFIX_SIZE)).to_ulong();
            table_t *current_table_ptr  = root_table_ptr;
            table_index = entry_bitset_index;
            table_entry_t *current_entry_ptr = &(*current_table_ptr)[table_index];

            hashed_key <<= PREFIX_SIZE;

            do {

                if (operation == ERASE && current_table_ptr != root_table_ptr) {
                    locate_stack->push_back({current_table_ptr, current_entry_ptr, table_index,
                        entry_bitset_index});
                }

                if ((*current_entry_ptr).first.none()) {
                    if ((*current_entry_ptr).second == nullptr) {
                        /* Empty slot (does not exist) in the root hash table */
                        return {EMPTY_ROOT, nullptr, current_entry_ptr, 0, 0, 0, 0};
                    } else {

                        if (operation == ERASE && locate_stack->empty()) {
                            locate_stack->push_back({current_table_ptr, current_entry_ptr,
                                table_index, entry_bitset_index});
                        }

                        /* Non-empty slot (exists) */
                        return {NOT_EMPTY, current_table_ptr, current_entry_ptr, table_index,
                            entry_bitset_index, tokens_consumed, hashed_key};
                    }
                } else {
                    entry_bitset_index = (int) (hashed_key >> (sizeof(size_type) * 8 - PREFIX_SIZE)).
                        to_ulong();
                    hashed_key = hashed_key << PREFIX_SIZE;

                    if (operation == ERASE && current_table_ptr == root_table_ptr) {
                        locate_stack->push_back({current_table_ptr, current_entry_ptr, table_index,
                            entry_bitset_index});
                    }

                    if (current_entry_ptr->first.test(entry_bitset_index)) {
                        /* Continues searching */
                        current_table_ptr = static_cast<table_t *>(current_entry_ptr->second);
                        table_index = std::popcount((current_entry_ptr->first << (BASE_SIZE -
                            entry_bitset_index - 1)).to_ulong()) - 1;
```

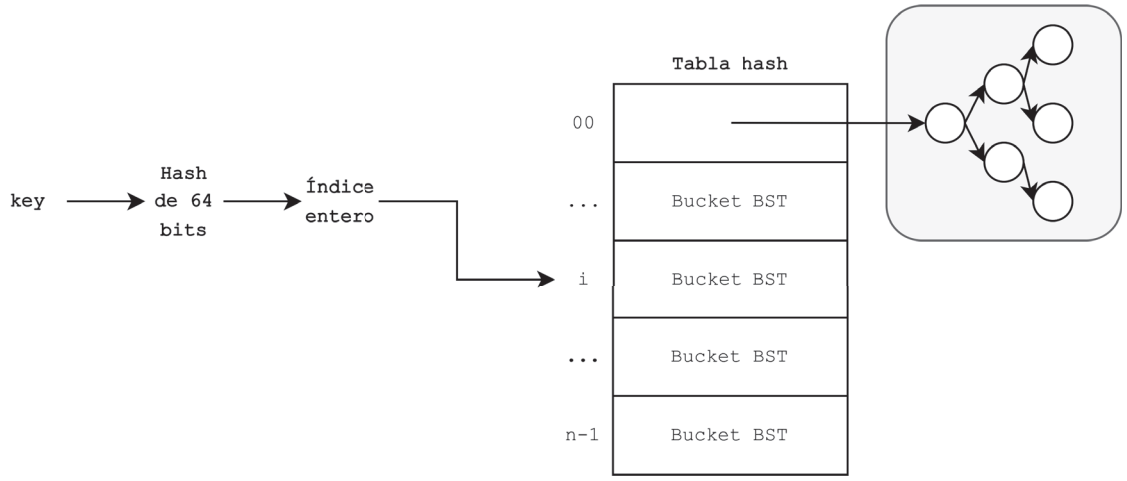
```
                                current_entry_ptr = &current_table_ptr->at(table_index);
                                ++tokens_consumed;
                            } else {
                                /* Empty slot (does not exist) in a sub-hash table */
                                auto next_table_ptr = static_cast<table_t *>(current_entry_ptr->second);
                                table_index = std::popcount((current_entry_ptr->first << (BASE_SIZE -
                                    entry_bitset_index - 1)).to_ulong());
                                return {EMPTY_SUBTABLE, next_table_ptr, current_entry_ptr, table_index,
                                    entry_bitset_index, 0, 0};
                            }
                        }
                    } while (true);
                }
};
```

4 Diseño alternativo: *hash table* con *binary search trees* (BST)

4.1. Diseño

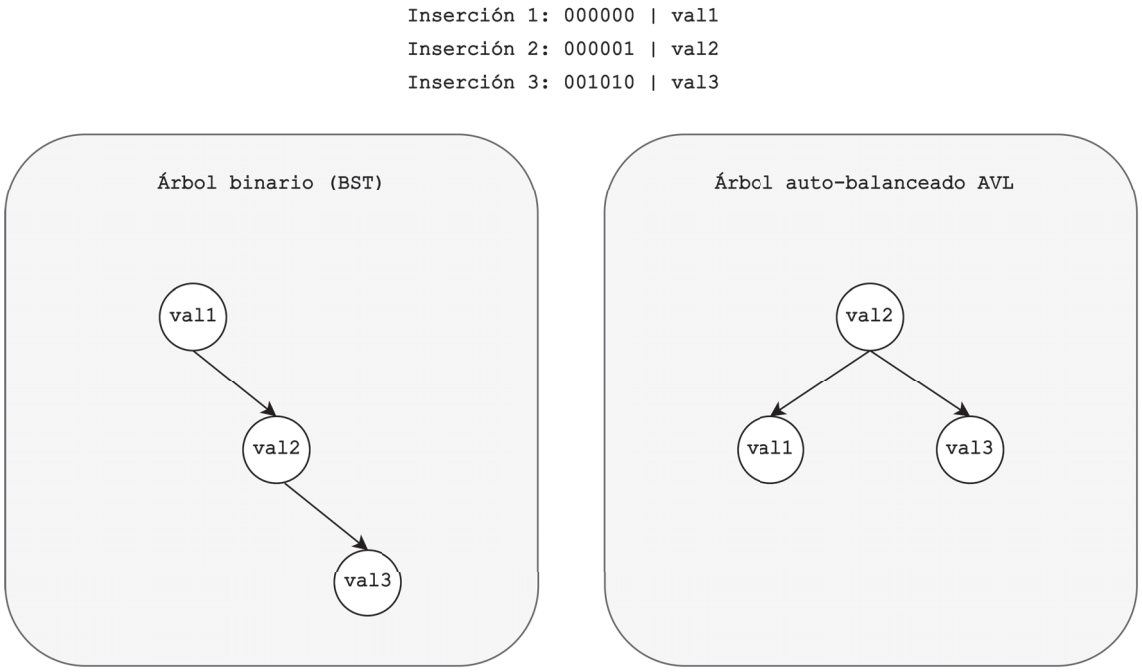
Como se ha visto anteriormente, la estructura `std::unordered_map` es una tabla *hash* cuyos *buckets* realmente son estructuras de datos subyacentes, concretamente listas enlazadas. Las listas enlazadas tienen algunas desventajas: en primer lugar carecen de proximidad espacial, por lo que no se aprovecha la caché del procesador, incurriendo en peor rendimiento que otras alternativas conscientes de esta opción; en segundo lugar, tiene una complejidad algortímica `O(n)` para el acceso siendo n el número de elementos que almacena, ya que partiendo del primer nodo se deben recorrer todos los siguientes hasta dar con el objetivo.

Una variación interesante para evaluar los cambios en las propiedades de la tabla es modificar qué estructura subyacente utilizan los buckets, seleccionado estructuras que ofrezcan mejores prestaciones que la lista enlazada para cada caso. Una alternativa interesante son los árboles binarios. Se trata de estructuras que ofrecen una complejidad de acceso `O(log(n))`, una mejora con respecto a la de las listas enlazadas. No obstante, los árboles binarios estándares pueden degradarse a listas enlazadas dependiendo de las claves que se inserten y el orden. Esto es, un árbol binario con una única rama, siendo virtualmente una lista enlazada con complejidad de acceso `O(n)`. Para explotar la mejora de complejidad de acceso es necesario que el *bucket* en cuestión sufra un número suficiente de colisiones tal que los árboles tengan un tamaño que aproveche la mejora de complejidad, pero hay que tener cuidado de no provocar un desbalanceo del mismo, degradándolos hasta el punto de simular listas enlazadas.



Para casos de uso donde se produzcan numerosas colisiones y se tienda al desbalanceo de los BST, puede ser interesante también utilizar árboles binarios auto-balanceables como estructura subyacente, tales como árboles AVL o *scapegoats*. Sin embargo, estos árboles, a cambio de garantizar el equilibrio de los nodos y con ello complejidad de acceso `O(long(n))`, requieren de un coste computacional superior en cada operación de inserción o borrado. Este *overhead* puede machacar el incremento de rendimiento derivado de disponer complejidad `O(long(n))`.

La figura muestra la estructura de un árbol BST y otro auto-balanceado AVL tras haber realizado tres inserciones en orden: `000000`, `000001` y `001010`. Como puede observarse, al ser una secuencia creciente (cada clave es menor a la siguiente), un árbol BST sufre una degradación a lista enlazada. Mientras tanto, el árbol AVL ha mantenido balanceados sus nodos. Este tipo de árboles garantizan que la altura de dos subárboles hijo no será superior a 1, lo cual consigue a través de rotaciones de nodos. De esta manera al insertar la tercera clave `001010`, el algoritmo de balanceo ha detectado que el nodo raíz tiene a su izquierda un subárbol hijo de altura 0 (no tiene hijo izquierdo), mientras que el de su derecha tiene altura 2, superando el límite de 1 establecido. Para solucionarlo aplica una rotación de nodos convirtiendo el hijo derecho en el nuevo nodo raíz, convirtiendo la antigua raíz en su hijo izquierdo. Existen cuatro operaciones de rotación para tratar todos los casos que se puedan dar de desbalanceo.



Además de lo anterior, para un aspecto fundamental en las tablas *hash* es disponer de funciones *hash* de calidad, que generen el menor número de colisiones posibles, por lo que parece razonable que si se dan demasiadas colisiones se evalúe primero la calidad de dicha función antes que la estructura subyacente.

4.2. Implementación propia

La implementación sigue el mismo convenio que el HAMT propuesto anteriormente: programación con genéricos mediante templates e interfaz consistente con la de `std::unordered_map` para facilitar el reemplazo y prueba de estas estructuras. Utiliza la función hash que propociona `std::hash` para la clase que se emplee para las claves. Internamente implementa los *buckets* como árboles, siguiendo los algoritmos y criterios propuestos por Ronald Rivest, Thomas Cormen, Charles Leiserson y Clifford Stein en su libro *Introduction to Algorithms* [12].

4.2.1. Código fuente

```
#include <memory>
#include <vector>

#define INITIAL_TABLE_SIZE 10
#define MAX_LOAD_FACTOR 1

template<
                class Key,
                class T,
                class Hash = std::hash<Key>,
                class Pred = std::equal_to<Key>,
                class Alloc = std::allocator<std::pair<const Key, T>>>
class bst {
public:
        typedef Key key_type;
        typedef T mapped_type;
        typedef Hash hasher;
```

```
typedef Pred key_equal;
typedef Alloc allocator_type;
typedef std::pair<const key_type, mapped_type> value_type;

bst() {
        root_ptr = nullptr;
}

mapped_type &get(const key_type &key) {
        bst_node *prev_node_ptr = nullptr;
        bst_node *node_ptr = root_ptr;
        while (node_ptr != nullptr) {
                if (is_equals(key, node_ptr->key)) {
                        return node_ptr->value;
                } else if (key < node_ptr->key) {
                        node_ptr = node_ptr->left_child;
                } else {
                        node_ptr = node_ptr->right_child;
                }
        }
        throw std::out_of_range("Not_found");
}

/*
 * Returns true if the insertion succeeded and false if the key is already inserted (no duplicated allowed).
 */
bool insert(const value_type &pair) {
        key_type key = pair.first;
        bst_node *prev_node_ptr;
        bst_node *node_ptr = root_ptr;

        while (node_ptr != nullptr) {
                prev_node_ptr = node_ptr;
                if (is_equals(key, node_ptr->key)) {
                        return false;
                } else if (key < node_ptr->key) {
                        node_ptr = node_ptr->left_child;
                } else {
                        node_ptr = node_ptr->right_child;
                }
        }
        if (node_ptr == root_ptr) {
                root_ptr = new bst_node{key, pair.second, nullptr, nullptr, nullptr};
        } else if (key < prev_node_ptr->key) {
                prev_node_ptr->left_child = new bst_node{key, pair.second, prev_node_ptr, nullptr, nullptr};
        } else {
                prev_node_ptr->right_child = new bst_node{key, pair.second, prev_node_ptr, nullptr, nullptr};
        }
        ++elements;
        return true;
}

/*
 * Returns true if the erase succeeded and false if the key does not exist.
 */
bool erase(const key_type &key) {
        bst_node *node_ptr = root_ptr;

        while (node_ptr != nullptr) {
                if (is_equals(key, node_ptr->key)) {
                        if (node_ptr->left_child == nullptr) {
                                transplant(node_ptr, node_ptr->right_child, true);
                        } else if (node_ptr->right_child == nullptr) {
                                transplant(node_ptr, node_ptr->left_child, true);
                        } else {
                                bst_node *min = minimum(node_ptr->right_child);
                                if (min->parent != node_ptr) {
                                        transplant(min, min->right_child, false);
                                        min->right_child = node_ptr->right_child;
                                        min->right_child->parent = min;
                                }
                                transplant(node_ptr, min, true);
                                min->left_child = node_ptr->left_child;
                                min->left_child->parent = min;
                        }
                        --elements;
                        return true;
                } else if (key < node_ptr->key) {
                        node_ptr = node_ptr->left_child;
                } else {
                        node_ptr = node_ptr->right_child;
                }
        }
        return false;
}

std::vector<value_type> *content_to_list() {
        auto *result = new std::vector<value_type>;
        traversal_mapping(result, root_ptr);
        return result;
}
```

```
private:
        struct bst_node {
                key_type key;
                mapped_type value;
                bst_node *parent;
                bst_node *left_child;
                bst_node *right_child;
        };

        bst_node *root_ptr;
        unsigned int elements;
        key_equal is_equals;

        bst_node *minimum(bst_node *root) {
                bst_node *node_ptr = root;
                while (node_ptr->left_child != nullptr) {
                        node_ptr = node_ptr->left_child;
                }
                return node_ptr;
        }

        void transplant(bst_node *original, bst_node *replacement, bool destroy) {
                if (original->parent == nullptr) {
                        root_ptr = replacement;
                } else {
                        if (original->parent->left_child != nullptr && is_equals(original->key, original->parent->
                              left_child->key)) {
                                original->parent->left_child = replacement;
                        } else {
                                original->parent->right_child = replacement;
                        }
                }
                if (replacement != nullptr) {
                        replacement->parent = original->parent;
                }
                if (destroy) {
                        delete original;
                };
        }

        // Maps the whole tree into a list using a recursive Postorder tree walk algorithm
        void traversal_mapping(std::vector<value_type> *list, const bst_node *node_ptr) {
                if (node_ptr != nullptr) {
                        traversal_mapping(list, node_ptr->left_child);
                        traversal_mapping(list, node_ptr->right_child);
                        value_type value(node_ptr->key, node_ptr->value);
                        list->push_back(value);
                }
        }
};

template<
                class Key,
                class T,
                class Hash = std::hash<Key>,
                class Pred = std::equal_to<Key>,
                class Alloc = std::allocator<std::pair<const Key, T>>>
class hashmap_tree {

public:
        typedef Key key_type;
        typedef T mapped_type;
        typedef Hash hasher;
        typedef Pred key_equal;
        typedef Alloc allocator_type;
        typedef size_t size_type;
        typedef std::pair<const key_type, mapped_type> value_type;

        typedef bst<key_type, mapped_type> tree_type;
        typedef std::vector<tree_type> table_type;

        hashmap_tree() {
                table_ptr = new table_type(INITIAL_TABLE_SIZE);
                elements = 0;
                bucket_count = INITIAL_TABLE_SIZE;
        }

        ~hashmap_tree() {
                delete table_ptr;
        }

        mapped_type &at(const key_type &key) {
                return table_ptr->at(hash_key(key)).get(key);
        }

        bool insert(const value_type &pair) {
                value_type v(pair.first, pair.second);
                if (table_ptr->at(hash_key(pair.first)).insert(v)) {
                        ++elements;
                        if (rehash_required()) {
```

```
                        rehash();
                }
                return true;
        }
        return false;
}

size_type erase(const key_type &key) {
        if (table_ptr->at(hash_key(key)).erase(key)) {
                --elements;
                return true;
        }
        return false;
}

size_type count(const key_type &key) {
        try {
                table_ptr->at(hash_key(key)).get(key);
                return true;
        } catch (std::out_of_range &e) {
                return false;
        }
}

size_type size() const {
        return elements;
}

protected:

private:
        table_type *table_ptr;
        unsigned int elements;
        unsigned int bucket_count;
        hasher hash;

        unsigned long hash_key(const key_type &key) {
                return hash(key) % bucket_count;
        }

        bool rehash_required() {
                return (double) elements / bucket_count >= MAX_LOAD_FACTOR;
        }

        void rehash() {
                auto old_table_ptr = table_ptr;
                elements = 0;
                bucket_count *= 2;
                table_ptr = new table_type(bucket_count);
                for (bst tree : *old_table_ptr) {
                        auto *a = tree.content_to_list();
                        for (value_type pair : *a) {
                                insert(pair);
                        }
                }
                delete old_table_ptr;
        }
};
```

5 Comparativa

De cara a evaluar el desempeño de la estructura principal del trabajo, el HAMT, se ha puesto a prueba con una batería de pruebas elaborada por el profesor Santiago Tapia. De los diferentes escenarios se ha seleccionado el más genérico para obtener una visión global de estas estructuras, comprobando si sus prestaciones a nivel práctico son capaces de batir al `std::unordered_map`.

El caso consiste en un número variable de inserciones de pares clave-valor, seguidas de una cantidad constante de búsquedas. Los pares clave-valor se componen de una cadena de caracteres como clave y un número entero como valor, ambos generado pseudoaleatoriamente. En el caso de la cadena de caracteres, primero se genera un entero pseudoaleatorio y a continuación se transforma a cadena de caracteres, por lo que la clave es la representación textual de dicho número.

Para representar el tiempo de búsqueda por operación se realiza una media, tomando la diferencia de tiempo entre el incio del bucle de búsquedas y su fin, y dividiendo dicho valor por y. Se utiliza una media porque de manera unitaria este tipo de operaciones tienen un tiempo muy cercano al límite de precisión del reloj que las mide, por lo que conviene tomar espacios de tiempo más amplios para evitar errores de medición. De esta manera se tienen unos datos más fiable pero como desventaja no es posible apreciar el efecto que tienen los *rehashing*, una de los problemas del `std::unordered_map` que pretende eliminar la estructura HAMT.

En las figuras que se muestran a continuación, la correspondencia entre código y estructura de datos es la siguiente:

- `u` representa al `std::unordered_map`.
- `m` representa al `std::map`.
- `h` represental al HAMT implementado.

La Figura 5.1 muestra el rendimiento de las operaciones de búsqueda. Como puede observarse, todas se degradan conforme incrementa el número de elementos almacenados. La estructura `std::map` es la que peor rendimiento presenta a lo largo de toda la prueba, con amplia diferencia. El HAMT supera al `std::unordered_map` desde el comienzo de la ejecución, con cierta ventaja, y dicha ventaja comienza a reducirse hasta el punto en que la estructura queda superada por `std::unordered_map`. Esto muestra como, aunque el HAMT presente mejor rendimiento para este caso, su rendimiento se degrada a mayor velocidad que el resto.

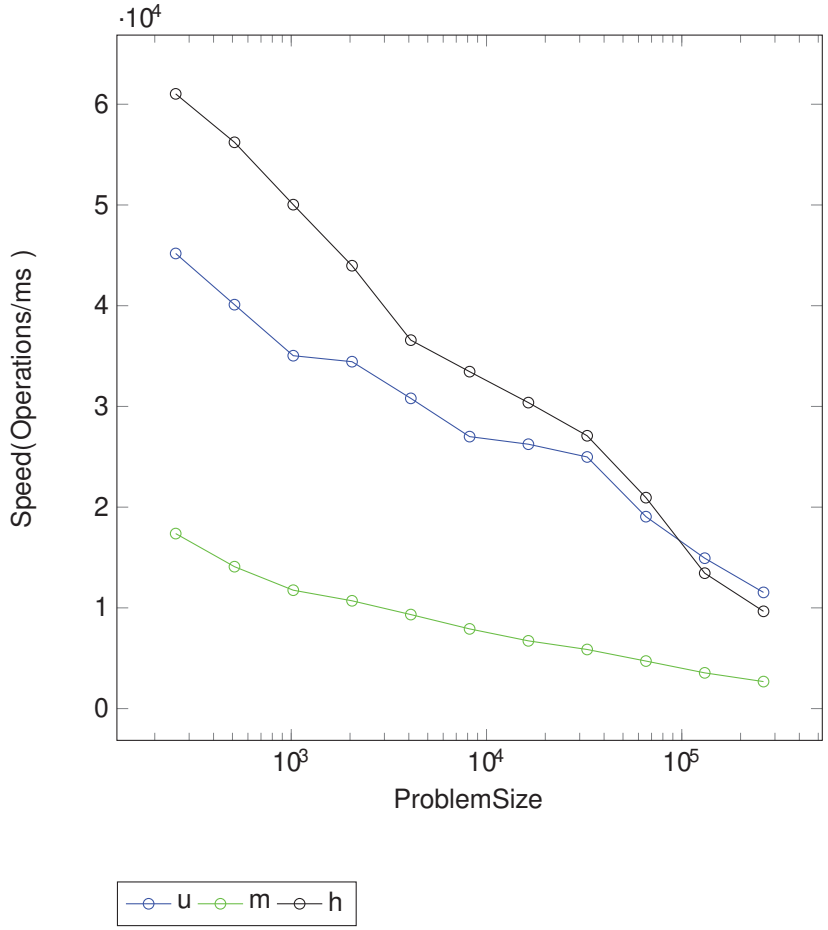


Figura 5.1: Rendimientos de la operación de búsqueda. Fuente: elaboración propia

Con respecto al rendimiento de las inserciones, la Figura 5.2 arroja unos resultados interesantes. El rendimiento de las estructuras sufren altibajos, incrementando y decrementando puntualmente mientras se amplia el número de elementos que almacenan. En total, la tendencia es una degradación, lo cual era previsible. Sin embargo, en este caso `std::unordered_map` y el HAMT se mantienen muy cercanos, sin una diferencia destacable, manteniendo la segunda una ventaja con respecto a la primera.

Finalmente la Figura 5.3 muestra el consumo de memoria en función del número de elementos almacenados. Aceptuando el caso de 2^8 y 2^{14}, `std::unordered_map` presenta mayor optimización en el uso de memoria. No obstante, la diferencia con el resto de estructuras no es muy significativa considerando el número de elementos que almacenan.

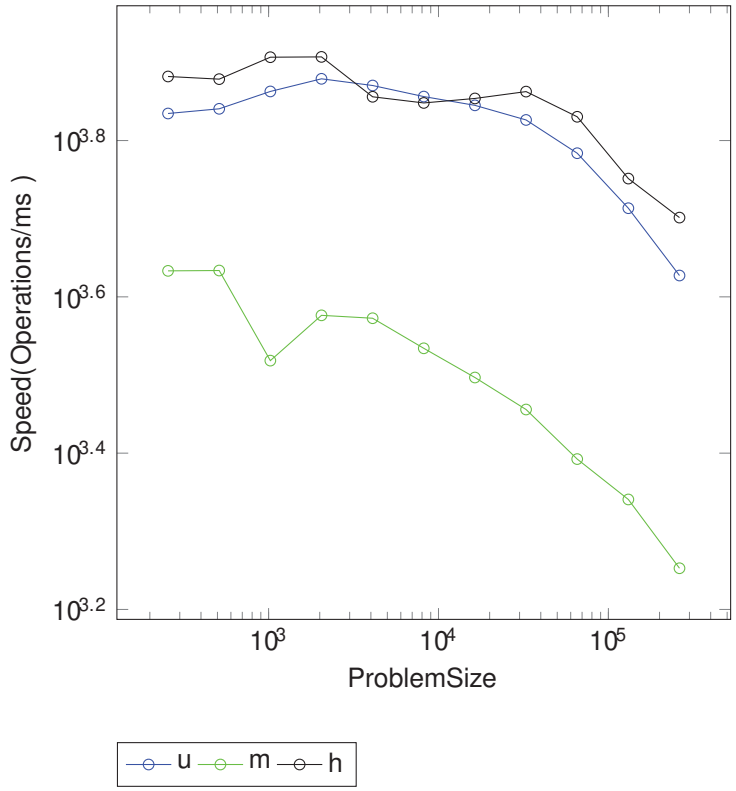


Figura 5.2: Rendimientos de la operación de inserción. Fuente: elaboración propia

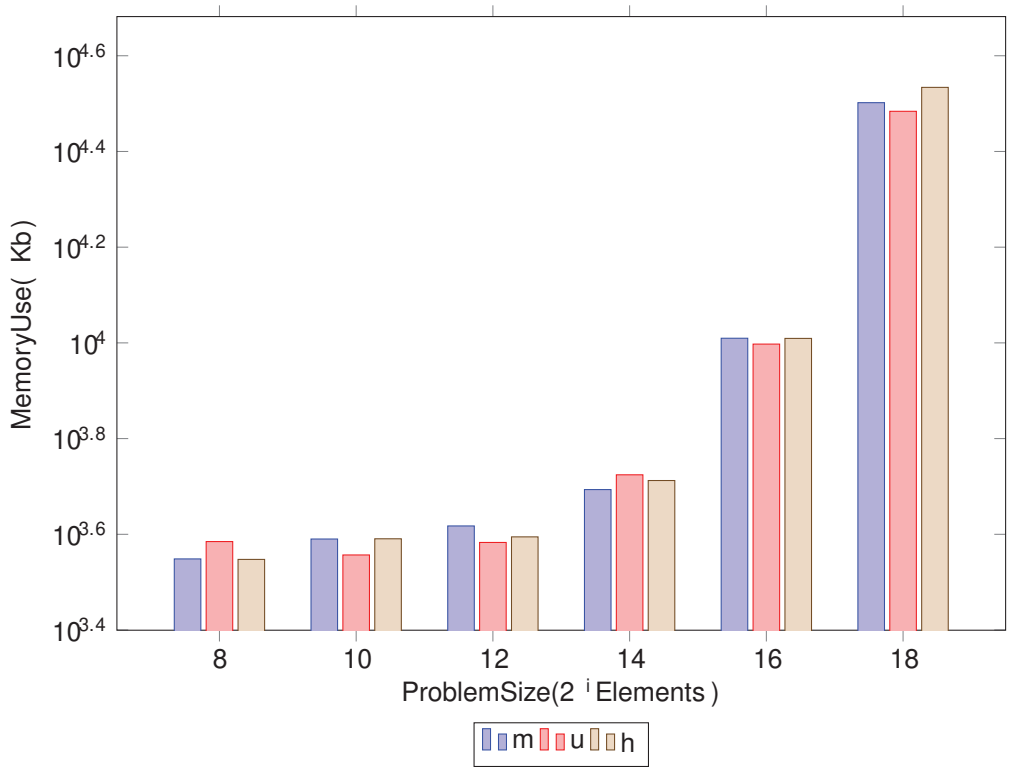


Figura 5.3: Consumo de memoria. Fuente: elaboración propia

6 Conclusiones

El trabajo propone la implementación de dos estructuras de datos alternativas al `std::unordered_map`, una ellas con un mayor peso de complejidad en su diseño (HAMT). Al tratarse de un trabajo de implementación e investigación de estructuras ha quedado patente la inmensa cantidad de posibilidades y combinaciones que existen de cara a buscar una alternativa de dicha estructura estándar. Cada decisión de diseño tiene un impacto sobre el funcionamiento de la estructura en diferentes escenarios, por lo que es un gran reto de ingeniería crear una estructura capaz de ofrecer un rendimiento excelente en todos los posibles casos de uso. Por ello, una de las conclusiones más importantes es la necesidad de utilizar para cada caso de uso específico una estructura que se adecúe a las particularidades del problema.

7 Bibliografía

[1] “Working Draft, Standard for Programming Language C++,” Feb. 2011.

[2] Cplusplus.com, “Standard Containers.” [Online]. Available: https://www.cplusplus.com/reference/stl

[3] J. G. de Jalón, *Aprenda C++ como si estuviera en primero*. Escuela Superior de Ingenieros Industriales de San Sebastián, Universidad de Navarra, 1998.

[4] Cplusplus.com, “Unordered_map interface.” [Online]. Available: https://www.cplusplus.com/reference/unordered_map/unordered_map/max_load_factor

[5] M. Thatte, “A deep dive into cojure’s data structures,” in *Euroclojure Conference*, Jun. 2015 [Online]. Available: https://www.youtube.com/watch?v=7BFF50BHPPo

[6] P. Bagwell, “Ideal Hash Trees,” LAMP – *Programming Methods Laboratory*, 2001 [Online]. Available: https://lampwww.epfl.ch/papers/idealhashtrees.pdf

[7] R. Hickey, “PersistentHashMap.java code,” Dec. 2017. [Online]. Available: https:// github.com/clojure/ clojure/blob/master/src/jvm/clojure/lang/PersistentHashMap.java

[8] H. Nord, “Erlang/OTP 18.0 has been released,” Jun. 2015. [Online]. Available: https://www.erlang.org/news/86

[9] P. Bagwell, “Fast And Space Efficient Trie Searches,” *Core.ac*, Jan. 2006 [Online]. Available: https://core.ac.uk/download/pdf/147909579.pdf

[10] Intel, “Intel® 64 and IA-32 Architectures Software Developer Manuals.” [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[11] Cplusplus.com, “Vector.” [Online]. Available: https://www.cplusplus.com/reference/vector/vector

[12] R. Rivest, *Introduction to Algorithms*. Massachusetts Institute of Technology, 2009 [Online]. Available: https://edutechlearners.com/download/Introduction_to_algorithms-3rd Edition.pdf

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
Fecha/Hora	Wed Jan 26 17:50:45 CET 2022
Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
Numero de Serie	561
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)