Apunte 4 - Estructuras Conjuntistas

Índice general

Índice de figuras

Algoritmos

Índice de ejercicios

Apunte 4

Estructuras conjuntistas

Actualizado: 31 de mayo de 2022

En esta unidad trabajaremos sobre la siguiente pregunta principal:

> *¿De qué manera se pueden organizar los datos para encontrar eficientemente un elemento particular dentro de un conjunto?*

A lo largo de este capítulo se presentarán estructuras de datos que tienen ese objetivo (árbol Heap, árbol binario de búsqueda, árbol AVL y tabla Hash); exponiendo los conceptos básicos, aplicaciones y operaciones más importantes de cada una. También se presentarán sus implementaciones, analizando en cada caso el uso de memoria y el tiempo de ejecución de las distintas operaciones.

4.1. Árboles heap

4.1.1. Descripción

Un árbol heap (también conocido como montículo) es una estructura que permite mantener sus elementos parcialmente ordenados. Su propósito es permitir encontrar rápidamente el menor o mayor elemento (según la implementación) de todos los elementos guardados en la estructura.

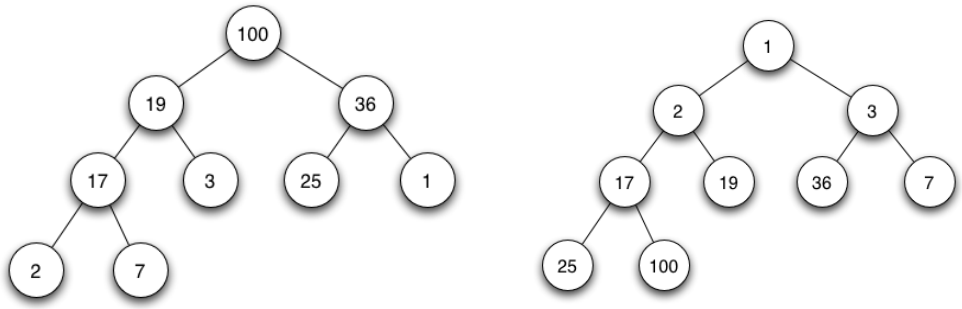
Se basa en una estructura de árbol binario, con dos restricciones adicionales:

- *Propiedad de heap*: Cada nodo del árbol contiene un valor mayor que el de sus hijos (para un heap máximo) o menor que el de sus hijos (para un heap mínimo).
- *Árbol semi completo*: El árbol está siempre balanceado[1] y las inserciones se realizan en el último nivel del árbol, de izquierda a derecha.

Se llaman parcialmente ordenados porque el orden entre nodos hermanos no está especificado en la propiedad de árbol heap, de manera que los subárboles de cualquier nodo son intercambiables.

En la Figura 4.1A se puede ver un ejemplo de árbol heap máximo, donde se observa que el elemento mayor del conjunto se encuentra en la raíz y que todos los nodos cumplen con la condición de ser mayor que sus hijos, aunque no hay orden establecido entre los hermanos. De manera similar, en la Figura 4.1B se observa un ejemplo de árbol heap mínimo donde el elemento menor del conjunto es el que se encuentra en la raíz.

[1] Un *árbol balanceado* es aquel donde todos los caminos desde la raíz hasta una hoja tienen la misma longitud o difieren, como máximo, en uno.



(A) Heap máximo (B) Heap mínimo

Figura 4.1: Ejemplos de árboles heap

Como se mencionó anteriormente, un árbol heap siempre mantiene una forma particular: se encuentra balanceado y los elementos se van agregando en el último nivel del árbol completándolo de izquierda a derecha, como se ve en la Figura 4.2. Las operaciones de inserción y eliminación deben mantener las propiedades de ordenamiento parcial y completitud del árbol. Más adelante en este apunte se explicará el algoritmo de estas operaciones.

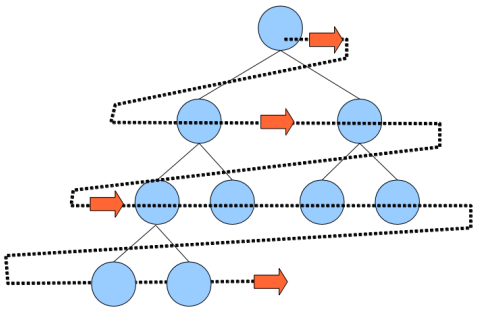


Figura 4.2: Esquema de orden de carga de elementos en el heap

4.1.2. Operaciones del TDA Árbol Heap

Como se explicó anteriormente, el propósito principal del árbol heap es facilitar la búsqueda y eliminación del elemento de la cima del árbol (que será el mínimo o máximo según el orden que se haya utilizado al implementar la inserción). Las operaciones básicas de esta estructura son:

- *constructor vacío*:
  // crea un árbol sin elementos.
- *insertar* (elemento): *boolean*
  // recibe un elemento y lo inserta en el árbol según el algoritmo que se explicará en la siguiente sección. Si la operación termina con éxito devuelve *verdadero* y *falso* en caso contrario.
  Nota: Los árboles heap aceptan elementos repetidos.
- *eliminarCima*(): *boolean*
  // elimina el elemento de la raíz (o cima del montículo) según el algoritmo que se explicará en la siguiente sección. Si la operación termina con éxito devuelve *verdadero* y *falso* si el árbol estaba vacío.
- *recuperarCima*(): elemento
  // devuelve el elemento que está en la raíz (cima del montículo). Precondición: el árbol no está vacío (si está vacío no se puede asegurar el funcionamiento de la operación).
- *esVacio*(): boolean
  // devuelve *falso* si hay al menos un elemento cargado en la tabla y *verdadero* en caso contrario.

Se pueden incluir otras operaciones útiles como *clone* y *vaciar*, como se ha hecho en los TDA anteriores. También es conveniente agregar la operación *toString* para permitir ver la estructura durante la etapa de debugging (esta operación deberá ser comentada o redefinida como privada antes de publicar la clase).

4.1.3. Algoritmos de las operaciones principales

El mecanismo para agregar un nuevo elemento en el árbol debe hacer que el árbol resultante mantenga tanto el orden entre los elementos como la forma balanceada y completa.

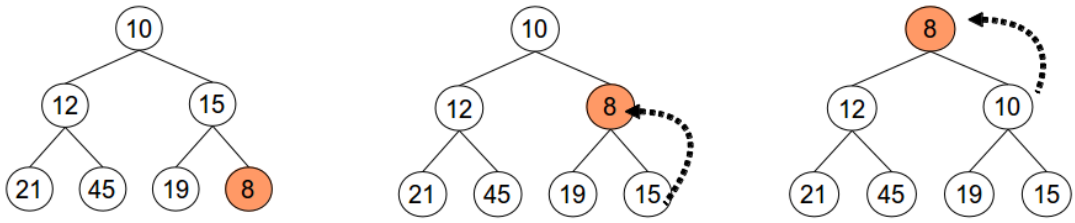
Algoritmo 4.1 Algoritmo de inserción en árbol heap mínimo

Algoritmo *insertar* (elemento)

1. Agregar el nuevo elemento como hoja. Se agrega en el último nivel. El nivel se completa de izquierda a derecha. Si el nivel esta completo se agrega el nuevo elemento en un nuevo nivel como primer elemento a la izquierda.
2. Si el nuevo elemento tiene menor valor que su padre lo hacemos subir (intercambiando sus valores)
3. Repetir el paso 2 hasta que el nuevo elemento cumpla con alguna de las siguientes condiciones:
   - *a*) llegue a una posición en que sea mayor que su padre
   - *b*) llegue a la raíz

Fin algoritmo

En la Figura 4.3 se muestra la inserción en un árbol heap mínimo. En este caso, el elemento nuevo (8) se inserta en la posición correspondiente para mantener la forma del árbol, es decir, en el último nivel y siguiendo el orden de izquierda a derecha. Luego se lo compara con su padre (15) y como es menor, se intercambian sus valores. A continuación vuelve a comparar con su padre (10) y como es menor, ocupa su lugar. En este caso, el algoritmo se detiene porque se llega a la raíz.



(1) se inserta 8 tan abajo a la izquierda como sea posible

(2) se lo compara e intercambia con su padre

(3) se repite la comparación e intercambio con su padre

Figura 4.3: Ejemplo de inserción de un elemento en un heap mínimo

En lo que respecta a la operación de eliminación en árboles heap, sólo interesa eliminar el elemento de la raíz, denominado cima. A continuación se explica dicho algoritmo:

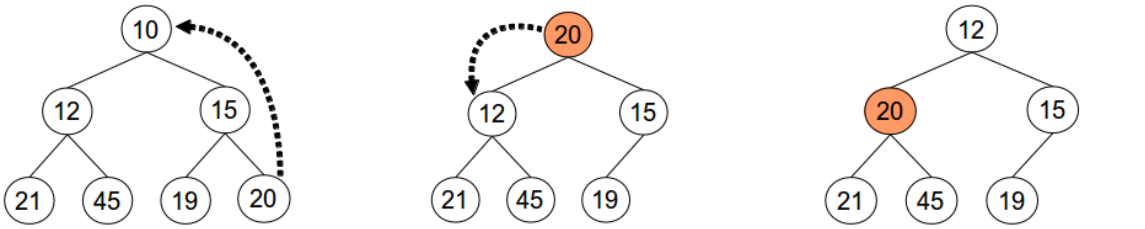
Algoritmo 4.2 Algoritmo de eliminación en árbol heap mínimo

Algoritmo *eliminarCima*()

1. Poner en la raíz el valor de la hoja más a la derecha del último nivel y eliminar dicha hoja.
2. "Empujar" el elemento hacia abajo, intercambiándo su valor con el del hijo de menor valor.
3. Repetir el paso 2 hasta que el elemento quede en una hoja o tenga menor valor que sus hijos.

Fin algoritmo

En la Figura 4.4 se muestra un ejemplo de eliminación en un árbol heap mínimo. Se comienza por poner en la raíz el valor de la hoja que esté en el último nivel, lo más a la derecha posible (20). Luego, se compara este elemento con sus hijos (12 y 15) y se intercambia por el menor de los dos (12). Se repite la comparación con los hijos (21 y 45). Como 20 es menor que ambos, se deja en ese lugar.



(1) se pone en la raíz el valor de la hoja más a la derecha del último nivel

(2) compara con sus hijos y se intercambia con el menor

(3) se repite la comparación (2) con sus nuevos hijos

Figura 4.4: Ejemplo de eliminación de un elemento en un heap mínimo

El algoritmo de búsqueda en árbol heap se reduce a recuperar el elemento *cima* (minimo o máximo según la implementación), por lo que sólo requiere realizar un acceso a la raíz.

4.1.4. Implementación utilizando arreglos

Si bien es posible representar el heap como un árbol binario implementado con nodos enlazados (como los vistos en el capítulo anterior), la condición de árbol semi-completo permite ser representado fácilmente utilizando un arreglo, colocando los elementos por niveles y, en cada nivel, los elementos de izquierda a derecha.

Dado que el árbol es completo, no es necesario almacenar los enlaces en el árbol. Por el contrario, siempre se puede calcular la posición de los hijos o la del padre a partir de la posición de un nodo en el arreglo (contando las posiciones del arreglo a partir de cero), considerando lo siguiente:

- El nodo raíz se almacena en la posición 1 del arreglo [2]
- Los hijos de un nodo almacenado en la posición n se almacenan:
  - Hijo izquierdo en la posición *2n*
  - Hijo derecho en la posición *2n+1*
- De las fórmulas anteriores se deduce que el padre de un nodo que está en la posición k ($k>0$) estará almacenado en la posición *k div 2*.

En la Figura 4.5 se muestra un heap máximo almacenado en un arreglo, donde la raíz (16) está en la posición i=1, luego su hijo izquierdo (14) en la posición 2i=2 y su hijo derecho en la posición 2i+1=3. De manera similar, se puede obtener fácilmente que el padre del elemento en la posición 5 del arreglo (7), es (14) que está en la posición 5 div 2=2.

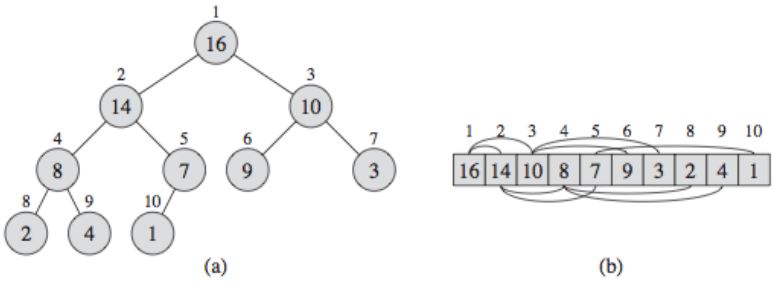


Figura 4.5: Ejemplo de representación de heap máximo almacenado en un arreglo

En la Figura 4.6 se presenta el UML de las clases necesarias para la implementación de árbol heap con arreglos. El atributo *ultimo* de tipo entero se utiliza para identificar cuál es la última celda utilizada (con elemento). Se debe recordar que el arreglo se utiliza desde la posición 1 en adelante, es decir que cuando el heap está vacío, *ultimo* es igual a 0.

[2] Algunos autores proponen almacenar la raíz en la posición 0, por lo que el hijo izquierdo del elemento en la posición n se almacenará en la posición 2n+1 y el hijo derecho en 2n+2, ajustándose la fórmula para encontrar al padre de manera similar.

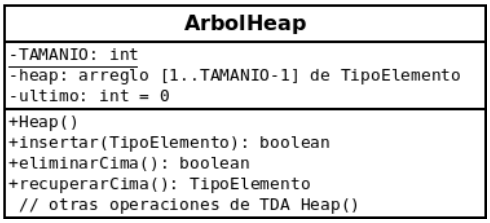


Figura 4.6: Diagrama UML de clases para árbol heap usando arreglo

En la Figura 4.3 se muestra la implementación de un árbol heap mínimo en una clase Java llamada HeapMin donde TipoElemento ha sido reemplazado por la interfaz *Comparable*. Esto es necesario porque *Object* de Java no tiene métodos que definan orden de mayor/menor entre instancias, entonces ese orden se impone de forma genérica cuando las clases implementan la interfaz *Comparable*. De esta manera cualquier clase que implemente dicha interfaz Java pueda ser almacenada como elemento del árbol heap[3].

Algoritmo 4.3 Definición de la clase HeapMin y método constructor en Java para elementos Comparable

```
package conjuntistas;

public class HeapMin {

    private Comparable[] heap;
    private int ultimo;
    private int TAMANIO = 20;

    public HeapMin() {
        this.heap = new Comparable[TAMANIO];
        this.ultimo = 0;  // la posición 0 nunca es utilizada
    }
```

A continuación, en la Figura 4.4, se muestra un ejemplo en Java del método *eliminarCima* para la implementación del diagrama presentado en la Figura 4.6, donde se reemplazó TipoElemento por `Comparable`. El método privado *hacerBajar(pos)* empuja el elemento que se colocó temporalmente en la raíz, intercambiándolo repetidamente por el menor de sus hijos, como se explicó en el algoritmo 4.2.

Algoritmo 4.4 Código Java del método *eliminarCima* de la clase HeapMin

```
public boolean eliminarCima() {
    boolean exito;
    if (this.ultimo == 0) {
        // estructura vacia
        exito = false;
    } else {
        // saca la raiz y pone la ultima hoja en su lugar
        this.heap[1] = this.heap[ultimo];
        this.ultimo--;
        // reestablece la propiedad de heap minimo
        hacerBajar(1);
        exito = true;
    }
    return exito;
}
```

[3] Para saber por qué usamos la interfaz Comparable y entender más de estructuras genéricas, ver el anexo «Claves para implementar estructuras generales»

Algoritmo 4.5 Código Java del método privado *hacerBajar* de clase HeapMin

```
private void hacerBajar(int posPadre) {
    int posH;
    Comparable temp = this.heap[posPadre];
    boolean salir = false;

    while (!salir) {
        posH = posPadre * 2;
        if (posH <= this.ultimo) {
            //temp tiene al menos un hijo (izq) y lo considera menor

            if (posH < this.ultimo) {
                //hijoMenor tiene hermano derecho

                if (this.heap[posH + 1].compareTo(this.heap[posH]) < 0) {
                    // el hijo derecho es el menor de los dos
                    posH++;
                }
            }

            // compara al hijo menor con el padre
            if (this.heap[posH].compareTo(temp) < 0) {
                // el hijo es menor que el padre, los intercambia
                this.heap[posPadre] = this.heap[posH];
                this.heap[posH] = temp;
                posPadre = posH;
            } else {
                // el padre es menor que sus hijos, está bien ubicado
                salir = true;
            }
        } else {
            // el temp es hoja, está bien ubicado
            salir = true;
        }
    }
}
```

Ejercicio 4.1: Implementación de árbol heap

- Crear una clase *ArbolHeap* que utilice un arreglo como base para almacenar un árbol heap mínimo e implementar todas las operaciones del TDA Árbol Heap para elementos que cumplan la interfaz *Comparable* de Java.
- Realizar una clase de testing para la clase anterior.

4.1.5. Análisis de eficiencia

Analicemos el costo de la inserción de un elemento en un árbol en el que ya se encuentran cargados n elementos.

Como se explicó al presentar el algoritmo de inserción (Algoritmo 4.1) el elemento se ubica en el último nivel, que se completa de izquierda a derecha (consideremos que esta tarea necesita un tiempo $T_{asigHoja}$), ese elemento se compara con su padre y se los intercambia en caso de ser necesario para mantener la propiedad de ordenamiento. Suponiendo que se realizaran i intercambios, podríamos escribir la función de tiempo para la inserción $T_i(n)$ de la siguiente manera:

$$T_i(n) = T_{asigHoja} + i * (T_{comp} + T_{intercambio}) \text{ donde } 0 \leq i \leq cantNiveles.$$

Notar que el tiempo máximo se obtendrá cuando ocurra la máxima cantidad de intercambios (es decir, cuando el elemento insertado "sube" hasta la raíz). Luego, podemos decir que el peor caso es cuando i es igual a la cantidad de niveles que tenga el árbol. Ahora bien, ¿cómo podemos saber cuantos niveles tiene el árbol?. Esto lo podemos inferir gracias a que sabemos que el árbol heap está siempre balanceado y casi completo. Como se mostró en el apunte de Árbol Binario, un *árbol binario completo* de altura h tiene $2^{h+1} - 1$ elementos. Como puede verse en la Figura 4.7, un árbol de altura 1 tiene 3 nodos, un árbol de altura 3 tiene 15 nodos, etc. La cantidad de niveles es uno más que la altura, por lo tanto $2^{h+1} - 1 = 2^i - 1$, donde i es la cantidad de niveles.

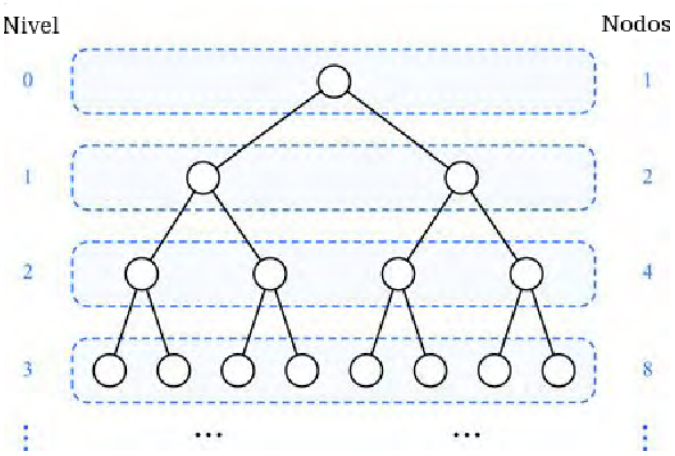


Figura 4.7: Árbol binario completo: Relación nivel y cantidad de nodos

Por lo tanto, como un árbol heap es un árbol casi completo, si sabemos la cantidad de elementos podemos aproximar la cantidad de niveles, despejando i de la fórmula anterior, de la siguiente manera:

- Sea n la cantidad de elementos en un árbol completo de i niveles, sabemos que $n = 2^i - 1$, haciendo pasaje de términos obtenemos que: $n+1 = 2^i$, luego aplicamos logaritmo en base 2 a ambos términos y obtenemos $log_2(n+1) = i$. Finalmente obtenemos que la cantidad de niveles en un árbol completo de n elementos es $i = log_2(n+1)$.

Considerando que, en el peor de los casos, el elemento será intercambiado por su padre tantas veces hasta llegar a la raíz, y que por ese motivo hace una visita por nivel, de la fórmula anterior podemos simplificar y decir que la cantidad de intercambios es del orden *O(log(n))*.

Volviendo a la fórmula del tiempo del algoritmo de inserción en el heap $T_i(n)$, analicemos el orden de velocidad de crecimiento de dicha función para la implementación con arreglos vista en la sección anterior:

- Sabemos que asignar el elemento en la hoja equivale a un acceso a una posición del arreglo $T_{asigHoja}$, luego el tiempo de comparación del hijo con el padre T_{comp} y el intercambio de ambos $T_{intercambio}$, son todas operaciones de O(1). A continuación, aplicamos las reglas de la suma y del producto, y obtenemos que:

$$O(T_i(n)) = O(1) + O(log(n)) * (O(1) + O(1)) = max(O(1), O(log(n)) = O(log(n))$$

- Por lo anterior, el orden de la operación de inserción en un árbol heap (implementado con arreglos) es de *O(log(n))*

De manera similar puede demostrarse que la eficiencia de la operación de eliminación es también de orden O(log(n)). Se debe destacar que ambos órdenes están calculados para la implementación con arreglos, porque se asegura que el tiempo de acceso desde el padre a los hijos y de estos a su padre es siempre O(1). Esto no puede asegurarse en una implementación de árbol binario con nodos enlazados de manera simple, a menos que existan enlaces tanto de padre a hijos como de los hijos al padre y se asegure un tiempo O(1) para agregar una hoja en la posición correspondiente del último nivel, lo cual hace la implementación dinámica bastante compleja.

La operación que devuelve el elemento en la cima del heap (mínimo o máximo) será de orden O(1), dado que sólo requiere acceder a la posición 1 del arreglo.

4.1.6. Ventajas y desventajas de árbol heap

En un árbol heap, el mayor elemento (o el menor, dependiendo de la relación de orden escogida) está siempre en el nodo raíz. Por esta razón, son generalmente elegidos para implementar Colas de Prioridad (tema que se tratará más adelante).

Una ventaja que poseen los heap es que, por ser árboles casi completos, se pueden implementar usando arreglos (arrays), lo cual simplifica su codificación contando con acceso directo a los elementos de la estructura, y libera al programador del uso de referencias a direcciones de memoria (enlaces o punteros).

Como desventaja se puede mencionar que la búsqueda de cualquier otro elemento que no sea el que se encuentra en la cima del montículo (la raíz) requiere recorrer la mayor parte del árbol (y en el peor de los casos, todos sus nodos).

4.1.7. Bibliografía sobre árboles heap

- A.V. Aho, J.E. Hopcroft y J.D. Ullman, Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988. (Capítulo 4, Sección 10)

4.2. Árboles Binarios de Búsqueda

4.2.1. Descripción

Un árbol binario de búsqueda (por sus siglas, ABB) es un árbol binario con las siguientes características:

- Sea N el nodo *raíz* de un *árbol binario de búsqueda* se cumple que:
  - En caso de tener subárbol izquierdo, la raíz N es mayor que el valor máximo almacenado en el subárbol izquierdo, y el subárbol izquierdo es a su vez un árbol binario de búsqueda.
  - En caso de tener subárbol derecho, la raíz N es menor que el valor mínimo almacenado en el subárbol derecho, y a su vez, el subárbol derecho es un árbol binario de búsqueda.

También podemos expresarlo de la siguiente manera:

- En un *árbol binario de búsqueda* todos los elementos del subárbol izquierdo de cualquier nodo (si no está vacío) son menores que el valor de dicho nodo, y todos los elementos del subárbol derecho (si no está vacío) son mayores que él.

Por este motivo se dice que un árbol binario de búsqueda es un árbol ordenado (a diferencia de árbol heap que corresponde a la clasificación de árboles parcialmente ordenados).

En la Figura 4.8 se grafica la relación de orden entre el nodo raíz N y sus subárboles izquierdo y derecho.

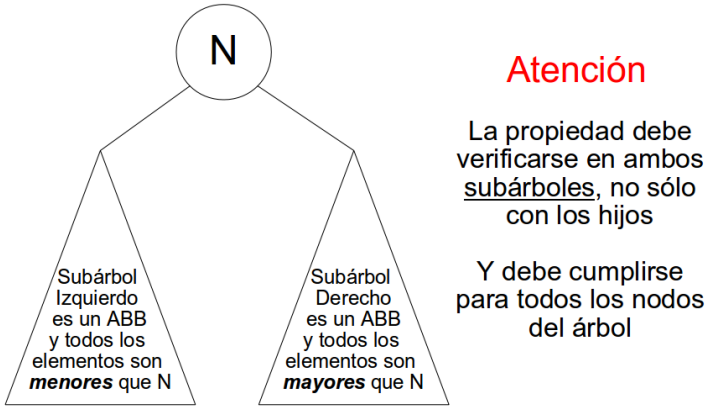


Figura 4.8: Definición de ABB

Como se puede observar en el ejemplo de la Figura 4.9, la propiedad de orden entre el nodo y los elementos de sus subárboles izquierdo y derecho, se cumple para todos los nodos. Otra característica importante es que su recorrido en *inorden* proporciona un listado de los elementos ordenados de forma ascendente.

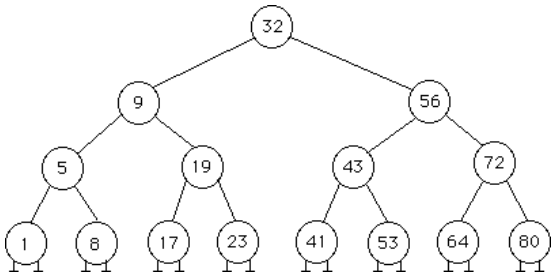


Figura 4.9: Ejemplo de ABB

La principal característica de los árboles ordenados, y en particular del ABB, es que la búsqueda de un elemento es eficiente en la mayoría de los casos, ya que se puede saber la posible ubicación de un elemento en el árbol sin necesidad de visitar todos los nodos del mismo. Para que el algoritmo de búsqueda se mantenga eficiente no se permiten insertar elementos con igual valor en un ABB, es decir, se considera un error tratar de insertar elementos repetidos.

4.2.2. Operaciones del TDA Árbol Binario de Búsqueda (ABB)

Los árboles ABB, como se explicó anteriormente, tienen el propósito de facilitar la búsqueda de cualquier elemento del conjunto, y de mantenerlos ordenados. Las operaciones básicas de esta estructura son:

- *constructor vacío*:
  // Crea un árbol sin elementos.
- *insertar* (elemento): *boolean*
  // Recibe un elemento y lo agrega en el árbol de manera ordenada. Si el elemento ya se encuentra en el árbol no se realiza la inserción. Devuelve *verdadero* si el elemento se agrega a la estructura y *falso* en caso contrario.
- *eliminar* (elemento): *boolean*
  // Recibe el elemento que se desea eliminar y se procede a removerlo del árbol. Si no se encuentra el elemento no se puede realizar la eliminación. Devuelve *verdadero* si el elemento se elimina de la estructura y *falso* en caso contrario.
- *pertenece* (elemento): *boolean*
  // Devuelve *verdadero* si el elemento recibido por parámetro está en el árbol y *falso* en caso contrario.
- *esVacio* (): *boolean*
  // Devuelve *falso* si hay al menos un elemento en el árbol y *verdadero* en caso contrario.
- *listar* (): Lista (de elemento)
  // recorre el árbol completo y devuelve una lista ordenada con los elementos que se encuentran almacenados en él.
- *listarRango* (elemMinimo, elemMaximo) : Lista (de elemento)
  // recorre parte del árbol (sólo lo necesario) y devuelve una lista ordenada con los elementos que se encuentran en el intervalo [elemMinimo, elemMaximo].
- *minimoElem* (): elemento
  // recorre la rama correspondiente y devuelve el elemento más pequeño almacenado en el árbol.
- *maximoElem* (): elemento
  // recorre la rama correspondiente y devuelve el elemento más grande almacenado en el árbol.

Se pueden incluir otras operaciones útiles como *clone, vaciar* y *toString* como en los TDA anteriores.

4.2.3. Algoritmos de las operaciones principales

Dada la propiedad de orden de los árboles ABB, es simple encontrar un elemento siguiendo el camino desde la raíz, bajando por una única rama del árbol. Por ejemplo, en la Figura 4.10, se muestra el recorrido necesario para encontrar el elemento D en el árbol. Dado que en la raíz se encuentra el elemento F, como D es menor que F se sabe que "si D está en el árbol" estará a la izquierda de F. Luego, se continúa el camino por la rama izquierda de F, es decir, por el subárbol con raíz B. Como D es mayor que B, sólo es necesario seguir la búsqueda por el subárbol derecho de B, ya que a la izquierda de B se encontrarán elementos menores que él. En el caso del ejemplo, D es el hijo derecho de B y la búsqueda termina ahí de forma exitosa. De no encontrarse, se continúa la búsqueda hasta llegar a un subárbol nulo, lo que indica que el elemento no estaba en el árbol, terminando la búsqueda sin éxito.

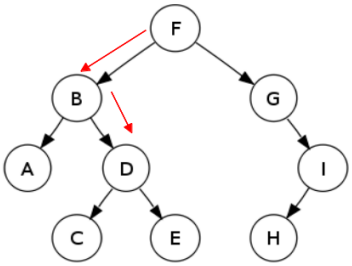


Figura 4.10: Ejemplo de búsqueda en ABB

A continuación se presenta el algoritmo de búsqueda en ABB.

Algoritmo 4.6 Algoritmo de búsqueda en ABB

Algoritmo *pertenece* (elemento)

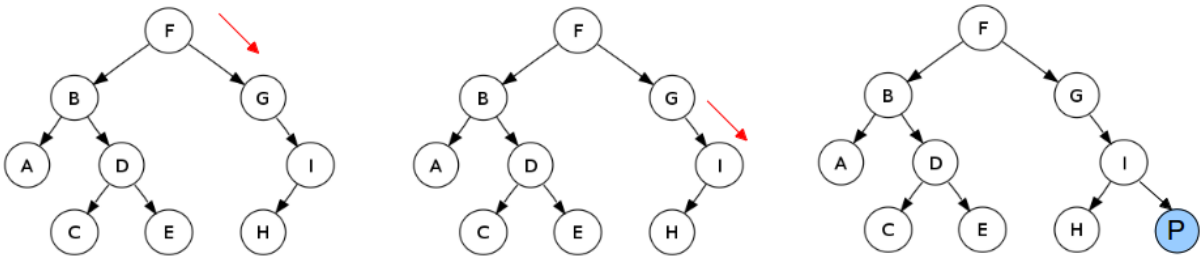
Comparar el elemento buscado con el del nodo raíz del subárbol
Si coinciden (son iguales) la búsqueda culmina con éxito
Sino

Si el elemento buscado es menor, la búsqueda continúa por el subárbol izquierdo
Si el elemento buscado es mayor, la búsqueda continúa por el subárbol derecho

Si se llega a un subárbol nulo la búsqueda culmina sin éxito

Fin algoritmo

En base al *algoritmo de búsqueda* se plantea el algoritmo de *inserción* de un elemento, como se ve en el ejemplo de la Figura 4.11. Para insertar el elemento P en el árbol ABB de la izquierda, se comienza comparando P con el elemento de la raíz (F). Como P es mayor se baja por la rama derecha de F, y se repite esta comparación con el hijo derecho de F (G) y con el hijo derecho de este (I). Finalmente, como el hijo derecho de I es nulo, se crea un nuevo nodo con P como elemento y se lo enlaza como hijo derecho de I.



(1) comienza la búsqueda de P en la raíz y baja a la derecha

(2) avanza buscando P, a la derecha de G y luego a la derecha de I

(3) como P > I y el hijo derecho de I es nulo, inserta P en un nuevo nodo

Figura 4.11: Ejemplo de inserción en ABB

Resumiendo, el algoritmo de inserción de un elemento en ABB es el siguiente:

Algoritmo 4.7 Algoritmo de inserción en ABB

Algoritmo *insertar* (elemento)

Comparar el elemento a insertar con el del nodo raíz del subárbol

Si el elemento buscado es menor, avanzar hacia el subárbol izquierdo
Si el elemento buscado es mayor, avanzar hacia el subárbol derecho
Si se llega a un subárbol vacío, agregar el elemento nuevo

Repetir el paso anterior hasta que se llegue a un subárbol vacío o se encuentre el elemento

Si el elemento se encuentra en el árbol, no se efectúa la operación y se devuelve error

Fin algoritmo

La operación de eliminación en un ABB es más compleja. Existen tres casos a considerar para eliminar un nodo N:

1. El nodo N es una hoja.
2. El nodo N tiene un solo hijo (izquierdo o derecho, es indistinto).
3. El nodo N tiene dos hijos.

Caso 1: El nodo a eliminar es una hoja

De los 3 casos, es el más sencillo de resolver. Si el nodo a eliminar no tiene hijos, simplemente lo desconectamos de su padre, dejando el enlace que los une en nulo. Por ejemplo, en la Figura 4.12 se muestra que al eliminar el elemento 96 del árbol de la izquierda, en el árbol resultante, simplemente se le quita el hijo derecho al nodo 73. El mismo caso ocurre si se intentara eliminar los elementos 55 o 13 que se encuentran en las hojas.

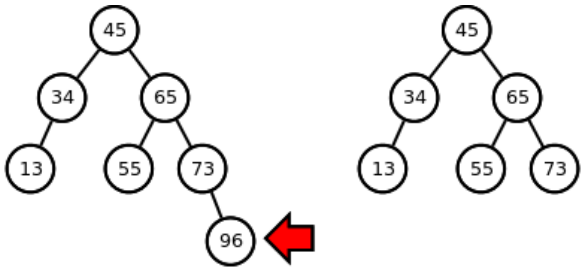


Figura 4.12: Eliminación en ABB (Caso 1: Nodo hoja)

Caso 2: El nodo tiene un solo hijo

Si el nodo a eliminar tiene un solo hijo, dicho hijo debe tomar el lugar del padre. Para ello, se conecta el padre de N con el hijo de N. Por ejemplo, en la Figura 4.13 se muestra que al eliminar el elemento 34 del árbol de la izquierda (que tiene como único hijo a 13), en el árbol resultante se conecta a 45 (padre de 34) con 13 (único hijo de 34). Luego el árbol resultante sigue cumpliendo con la condición de estar ordenado. Lo mismo vale si el nodo tiene únicamente un hijo derecho. Por ejemplo, si en la figura se intentara eliminar el nodo 73, basta con enlazar su hijo 96 como hijo derecho de 65.

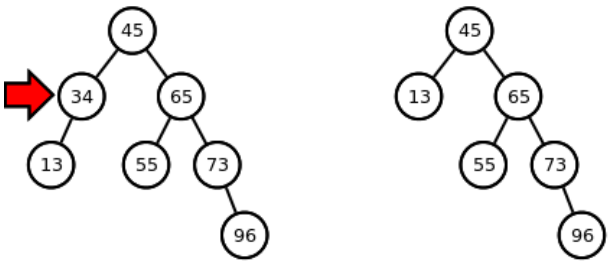


Figura 4.13: Eliminación en ABB (Caso 2: Nodo con un hijo)

Caso 3: El nodo tiene ambos hijos

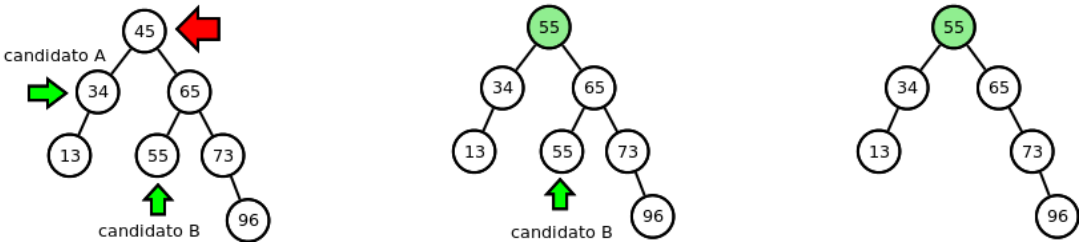
Este caso es de mayor complejidad que los anteriores. Si el nodo a eliminar tiene ambos hijos, se debe encontrar algún elemento entre los descendientes de N que "suba" a ocupar el lugar de N, de manera que se siga verificando la propiedad de orden del ABB.

Sea N el elemento a eliminar, cuyo nodo tiene dos hijos, existen dos posibles candidatos a ocupar su lugar. Por ejemplo, en la Figura 4.14 si se desea eliminar el elemento 45, los candidatos a ocupar su lugar son 34 y 55. ¿Cómo los identificamos?

- *Candidato* A: El mayor elemento del subárbol izquierdo de N.
  Para encontrarlo, se baja al hijo izquierdo de N y desde allí se repite el movimiento sólo por los enlaces hacia la derecha. El nodo que se encuentre en ese camino que no tiene hijo derecho será el máximo del subárbol izquierdo. En el caso de la Figura 4.14, ese nodo es el 34. A este candidato también se lo puede identificar como el predecesor de N en el recorrido en inorden del árbol

- *Candidato B*: El menor elemento del subárbol derecho de N.
  En este caso, se comienza bajando al hijo derecho de N y desde allí se repite el movimiento pero sólo por los enlaces hacia la izquierda. El nodo que no tenga hijo izquierdo contendrá al mínimo de ese subárbol. En el caso de la Figura 4.14, el nodo que no tiene hijo izquierdo (en el subárbol derecho de 45) es el 55. A este candidato también se lo puede identificar como el sucesor de N en el recorrido en inorden del árbol.

Una vez buscado el candidato (se debe optar por uno de los dos, no hace falta buscar ambos), su valor se utiliza para reemplazar el valor del nodo donde está N. A continuación, se procede a eliminar del subárbol correspondiente el elemento candidato elegido. Como se puede observar, los candidatos siempre son hojas o tienen un solo hijo, por lo tanto la eliminación del nodo candidato se resolverá utilizando los casos 1 o 2 (nunca caso 3). En la Figura 4.14 se muestra la eliminación eligiendo al candidato B.



(1) árbol original

(2) se reemplaza el 45 por el candidato B (55)

(3) se elimina el candidato B del subárbol derecho (caso 1: nodo hoja)

Figura 4.14: Eliminación en ABB (Caso 3: Nodo con dos hijos)

Algoritmo 4.8 Algoritmo de eliminación en ABB

Algoritmo *eliminar* (elemento)

Comparar el *elemento* a eliminar con el elemento en el nodo actual del árbol

Si es menor avanzar hacia el subárbol izquierdo
Si es mayor avanzar hacia el subárbol derecho

Repetir el paso anterior hasta que se llegue a un subárbol vacío o se encuentre el elemento

Si encontró el elemento

Si es hoja: eliminar según el caso 1
Si tiene un hijo: eliminar según el caso 2
Si tiene ambos hijos: eliminar según el caso 3

Si encontró un subárbol vacío, la operación culmina sin éxito y se devuelve error

Fin algoritmo

4.2.4. Implementación dinámica

La implementación más eficiente de Árbol Binario de Búsqueda es mediante nodos enlazados, de la misma manera que se han implementado los árboles binarios simples. La clase ArbolBB tiene un único

atributo raíz (de tipo NodoArbol) y luego la interfaz que corresponde con las operaciones del TDA descriptas en la sección 4.2.2. En la Figura 4.15 se presenta el diagrama UML de ambas clases.

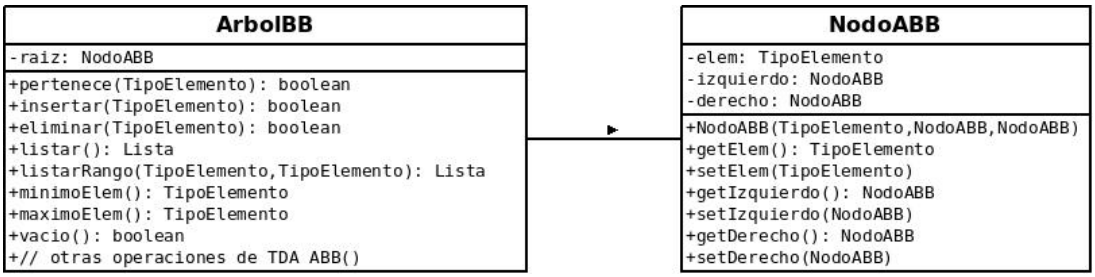


Figura 4.15: Diagrama UML de clases para Árbol Binario de Búsqueda (ABB)

A continuación se muestra un ejemplo en Java del método *insertar* para la implementación del diagrama presentado en la Figura 4.15, donde se reemplazó TipoElemento por la interfaz Comparable de Java[4] de igual manera que se usó en la implementación de árbol heap. El método *insertar* presentado está implementado de manera recursiva. También es posible hacerlo de manera iterativa, ya que en ABB no es necesario moverse a otras ramas pues el camino recorrido hasta encontrar la posición a insertar es único, y esto se debe a la propiedad de orden total de los ABB. Lo mismo ocurre con las operaciones de búsqueda y eliminar.

Algoritmo 4.9 Código Java del método *insertar* de clase ArbolBB (propuesta recursiva)

```
public boolean insertar(Comparable elemento) {
    boolean exito = true;
    if (this.raiz == null) {
        this.raiz = new NodoABB(elemento);
    } else {
        exito = insertarAux(this.raiz, elemento);
    }
    return exito;
}

private boolean insertarAux(NodoABB n, Comparable elemento) {
    // precondicion: n no es nulo
    boolean exito = true;

    if ((elemento.compareTo(n.getElem()) == 0)) {
        // Reportar error: Elemento repetido
        exito = false;

    } else if (elemento.compareTo(n.getElem()) < 0) {
        // elemento es menor que n.getElem()
        // si tiene HI baja a la izquierda, sino agrega elemento
        if (n.getIzquierdo() != null) {
            exito = insertarAux(n.getIzquierdo(), elemento);
        } else {
            n.setIzquierdo(new NodoABB(elemento));
        }
    } else // elemento es mayor que n.getElem()
    // si tiene HD baja a la derecha, sino agrega elemento
    if (n.getDerecho() != null) {
        exito = insertarAux(n.getDerecho(), elemento);
    } else {
        n.setDerecho(new NodoABB(elemento));
    }

    return exito;
}
```

[4] Para saber por qué usamos la interfaz Comparable y entender más de estructuras genéricas, ver el anexo "Claves para implementar estructuras generales"

Ejercicio 4.2: Implementación de árbol binario de búsqueda ABB

- Crear una clase ArbolBB e implementar todas las operaciones del TDA Árbol Binario de Búsqueda según lo visto en la secciones anteriores.
- Realizar una clase de testing para la clase anterior. Probar usando elementos tipo int, String y algún tipo definido por el usuario.

4.2.5. Análisis de eficiencia

Como se explicó en el análisis de eficiencia del Árbol Heap (Sección 4.1.5), se puede demostrar que en un árbol binario completo con n nodos, todos los caminos de la raíz a una hoja tendrá aproximadamente $log_2 n$ nodos. Si el árbol ABB se mantiene balanceado, las operaciones *pertenece* e *insertar* tomarán una cantidad de tiempo constante en cada nodo que visitan y la cantidad de nodos visitados es del O(log n), luego el tiempo total consumido por ambos algoritmos es también *O(log n)*. El mismo análisis corresponde a la operación *eliminar*, que a pesar de su complejidad mantiene la cantidad de nodos a visitar en uno por nivel, por lo que el orden es O(log n), aún cuando al buscar el candidato puede visitar algunos nodos 2 veces.

Sin embargo, el orden de eficiencia O(log n) sólo puede asegurarse cuando el árbol está balanceado (es decir cuando todos los caminos de la raíz a una hoja, mantienen una longitud similar a O(log n). Lamentablemente, esto no puede asegurarse en la práctica, dado que depende del orden de inserción de los elementos en el árbol. Por ejemplo, en la Figura 4.16 se muestran tres árboles ABB que tienen los mismos elementos. En el árbol de la izquierda se han ingresado aleatoriamente (40,35,38,65,10,89,49), obteniéndose un árbol completamente balanceado, donde las ramas tienen la longitud ideal ($log_2(7+1) = 3$). Este es el mejor caso posible. En el árbol del medio los elementos se ingresaron de la siguiente manera (35, 65, 38, 10, 49, 89, 40). El resultado es el caso más común, cuando algunas ramas quedan más llenas que otras. Finalmente, el árbol de la derecha muestra el peor caso posible, que se produce cuando los elementos son ingresados de manera ordenada (en este caso ascendente), generando un árbol con una sola rama de longitud 8, es decir orden *O(n)*.

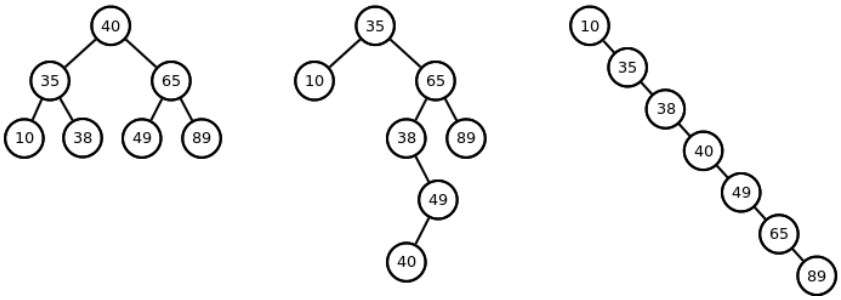


Figura 4.16: Análisis de eficiencia en ABB

4.2.6. Ventajas y desventajas de árbol binario de búsqueda

El ABB es una estructura que mantiene los elementos ordenados, por lo que está diseñado para localizar un elemento siguiendo un camino desde la raíz, visitando un solo nodo por nivel, y en el peor de los casos necesita hacer tantas visitas como niveles tenga el árbol para encontrar el elemento o determinar que este no se encuentra en la estructura. Dado que se realiza una visita por nivel, en general se dice que el ABB tiene un orden de eficiencia para sus operaciones de búsqueda, inserción y eliminación, de O(log n). Sin embargo, este orden de eficiencia se cumple sólo en el mejor de los casos, es decir mientras el árbol se mantenga balanceado o con un desbalanceo leve. La posibilidad de que el árbol se desbalancee, llegando a un orden *O(n)*, hace que en la realidad no se utilicen los árboles ABB puros, optando por alguna de las variantes de árboles auto-balanceados, como el AVL (que veremos en la siguiente sección) o los árboles rojo-negro, entre otros.

4.2.7. Bibliografía recomendada sobre ABB

- A.V. Aho, J.E. Hopcroft y J.D. Ullman, Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988. (Capítulo 5, Sección 1)
- Joaquín Fdez-Valdivia, Apuntes de la asignatura "Estructuras de Datos", Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de Granada http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/arb_BB.htm

4.3. Árboles auto-balanceados AVL

4.3.1. Descripción

El árbol AVL toma su nombre de las iniciales de los apellidos de sus inventores, *A*delson-*V*elskii y *L*andis, y fue publicado en 1962.

Los árboles AVL son *árboles binarios de búsqueda que están siempre equilibrados* de tal modo que, para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha, y viceversa. Gracias a esta forma de equilibrio (o balanceo), la complejidad de la búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad *O(log n)*, como se discutió en la sección anterior.

Para implementar este tipo de dato abstracto, se parte de un árbol binario de búsqueda en el que se modifican las operaciones de inserción y eliminación para asegurar que la diferencia de altura entre los subárboles derecho e izquierdo de cada nodo no sea mayor a uno en valor absoluto. A la diferencia de altura entre los subárboles de un nodo la llamaremos *balance* y lo calcularemos de la siguiente manera:

$$balance(N) = altura_{HijoIzq(N)} - altura_{HijoDer(N)}$$

Los valores aceptables de balance son[5]:

- *Balance 1*: implica que la altura subárbol izquierdo es mayor que la altura subárbol derecho (el árbol cae apenas hacia la izquierda).
- *Balance 0*: implica que las alturas de los dos subárboles son iguales (el árbol está bien balanceado).
- *Balance -1*: implica que la altura del subárbol derecho es mayor que la altura del subárbol izquierdo (el árbol cae apenas hacia la derecha)

Cuando el balance de un nodo es 2 o -2 es el momento de "reorganizar" el árbol para recuperar el balance deseado, que puede ser -1, 0 o 1. Para ello, después de agregar o extraer un nodo que provoque un desbalance mayor a 1 en alguno de los subárboles, se aplica un mecanismo de reorganización mediante rotaciones. Estas rotaciones pueden ser simples o dobles y serán explicadas en la siguiente sección.

En la Figura 4.17 se presentan dos árboles binarios de búsqueda. El de la izquierda cumple la condición de AVL. Para ello se observa cada nodo del árbol y se calcula su balance con la fórmula anterior. A los subárboles *nulos* (vacíos) los consideraremos de altura -1. En el caso de las *hojas,* su balance es 0. Veamos el cálculo para alguno de los otros nodos:

- b*alance(B) = altura(nulo) - altura(D) = -1 - 0 = -1*
  Se puede observar en la Figura 4.17, en el árbol de la izquierda, el subárbol con raíz en el nodo B está un poco caído hacia la derecha, pero dentro del límite aceptable.
- *balance(M) = altura(H) - altura(Q) = 1 - 0 = 1*
  Se observa un ejemplo en la Figura 4.17, en el árbol de la izquierda, el subárbol con raíz en el nodo M está un poco caído hacia la izquierda, pero dentro del límite aceptable.
- *balance(E) = altura(B) - altura(M) = 1 - 2 = -1*
  En el árbol de la izquierda de la Figura 4.17, el subárbol con raíz en el nodo E está un poco caído hacia la derecha, pero dentro del límite aceptable.

[5] Algunos autores calculan el balance de manera inversa, $balance(N) = altura(HI) - altura(HD)$), por lo que el signo del balance se invierte.

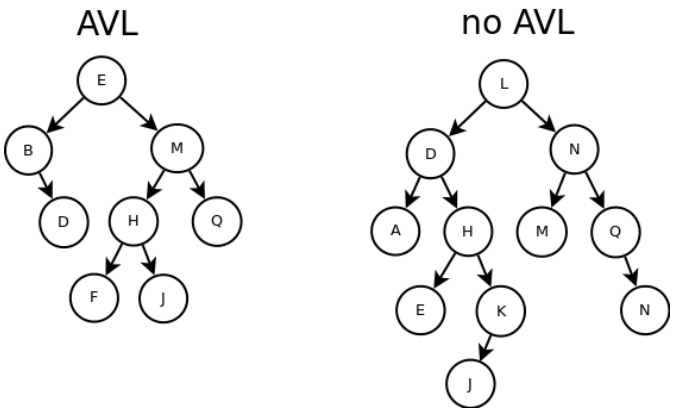


Figura 4.17: Ejemplo de un árbol balanceado y un árbol no balanceado

Al analizar el árbol de la derecha en la Figura 4.17, podemos ver que no cumple con la condición AVL. El nodo que está produciendo el desbalance es D:

- *balance(D) = altura(A) - altura(H) = 0 - 2 = -2*
  Esto significa que en el nodo D el árbol está caído hacia la derecha 2 niveles, es decir fuera del rango permitido. Acá habrá que realizar una rotación.

La detección del desbalance se hace siempre después de la inserción o eliminación de un elemento, y es un chequeo que se realiza en todos los niveles a la vuelta de la recursión. En cuanto se detecta que el balance no respeta el balance permitido se aplican una o dos rotaciones para solucionarlo, como se explicará en la sección siguiente.

4.3.2. Rotaciones en el árbol AVL

Existen 4 tipos de rotaciones. Para darnos cuenta cuál debemos utilizar, nos fijamos en el signo del balance del nodo y del hijo correspondiente. Si tienen igual signo se aplica una rotación simple. Si tienen distinto signo, se aplica una rotación doble. En la tabla a continuación se presenta la rotación necesaria para cada tipo de desbalance encontrado:

Balance padre	*Balance hijo*	*Signo*	*Rotación*
2	1 ó 0	igual	Simple a derecha
2	-1	distinto	Doble izquierda-derecha
-2	-1 ó 0	igual	Simple a izquierda
-2	1	distinto	Doble derecha-izquierda

Figura 4.18: Tabla de rotaciones en AVL, según el balance de padres e hijos

Por ejemplo, en el árbol no AVL de la Figura 4.17, al detectar que el nodo k está caído hacia la derecha (balance -2), miramos el balance de su hijo derecho, n, (balance -1). Esto quiere decir que tanto el padre como el hijo están caídos hacia la derecha, luego el tipo de rotación a aplicar es *simple a izquierda.*

4.3.2.1. Rotación simple a la izquierda

Esta rotación se aplica cuando el nodo padre está caído a la derecha (balance -2) y su hijo derecho está caído hacia el mismo lado (balance -1).

Sea un subárbol de raíz (r) y de hijos izquierdo (i) y derecho (d). La rotación consiste en "girar" el árbol para que la raíz sea el hijo derecho (d). A la antigua raíz (r) se la coloca como hijo izquierdo de (d) y al antiguo hijo izquierdo de (d) se lo enlaza como hijo derecho de (r). Tanto el hijo derecho de d (hd), como el hijo izquierdo de r (i) se dejan en el mismo lugar. (Ver Figura 4.19).

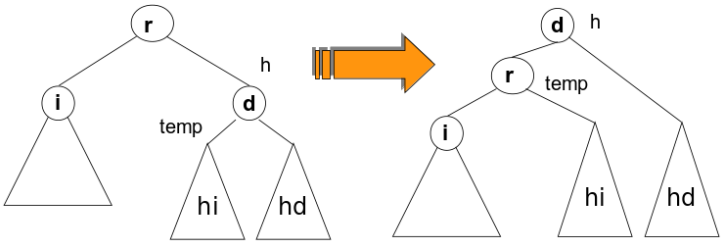


Figura 4.19: Esquema de rotación simple a izquierda

A continuación se presenta el algoritmo de rotación explicado anteriormente, donde el parámetro r representa al pivote. Una vez efectuada la rotación, el algoritmo devuelve la nueva raíz del subárbol.

Algoritmo 4.10 Algoritmo de rotación simple a izquierda sobre pivote r

```
Algoritmo rotarIzquierda(r)

    h = hijo_der(r)
    temp = hijo_izq(h)
    hijo_izq(h) = r
    hijo_der(r) = temp
    Retornar h          // retornar la nueva raíz del subárbol

Fin algoritmo
```

En la Figura 4.20 se presenta un ejemplo de un árbol donde el nodo de la raíz (8) se encuentra desbalanceado hacia la derecha ($balance(8) = altura(5) - altura(15) = 0 - 2 = -2$). Como el desbalance es negativo, se mira el balance de su hijo derecho 15 ($balance(15) = altura(13) - altura(20) = 0 - 1 = -1$). Al ser ambos del mismo signo, se aplica rotación simple. Al estar "caído" hacia la derecha, la rotación que se aplica es a izquierda, obteniéndose el 15 en la raíz, la antigua raíz a su izquierda (8), y el antiguo hijo izquierdo de 15 (13) como hijo derecho de 8. Gracias a esta rotación el subárbol ahora se encuentra balanceado ($balance(15) = 1 - 1 = 0$) y cumpliendo con la condición de orden de árbol binario de búsqueda.

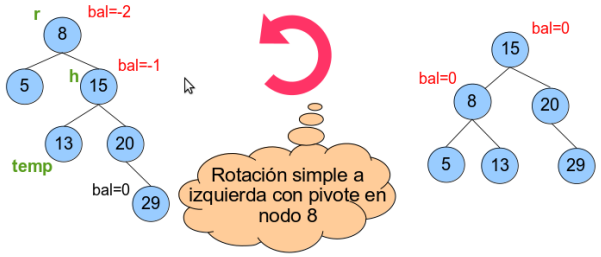


Figura 4.20: Ejemplo de rotación simple a izquierda

4.3.2.2. Rotación simple a la derecha

Esta rotación se aplica cuando el nodo padre está caído a la izquierda (balance 2) y su hijo izquierdo está caído hacia el mismo lado (balance 1).

Sea un subárbol de raíz (r) y de hijos izquierdo (i) y derecho (d). La rotación simple a derecha consiste en "girar" el árbol para que la raíz sea el hijo izquierdo (i). Como su hijo derecho se coloca a la antigua raíz (r) y al antiguo hijo derecho de (i) se lo enlaza como hijo izquierdo de (r). Tanto el hijo izquierdo de i (hi) como el hijo derecho de r (d) se dejan en el mismo lugar. (Ver Figura 4.21).

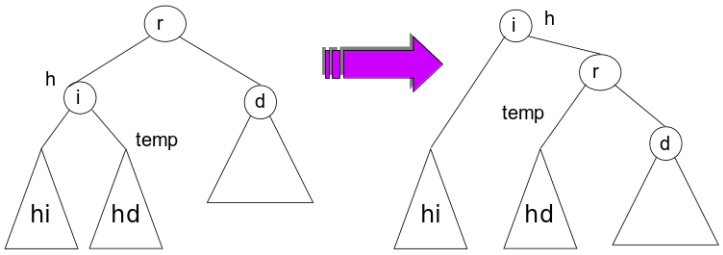


Figura 4.21: Esquema de rotación simple a derecha

A continuación se presenta el algoritmo de rotación explicado anteriormente, donde el parámetro r representa al pivote. Una vez efectuada la rotación, el algoritmo devuelve la nueva raíz del subárbol.

Algoritmo 4.11 Algoritmo de rotación simple a derecha sobre pivote r

```
Algoritmo rotarDerecha(r)

    h = hijo_izq(r)
    temp = hijo_der(h)
    hijo_der(h) = r
    hijo_izq(r) = temp
    Retornar h          // retornar la nueva raíz del subárbol

Fin algoritmo
```

En la Figura 4.20 se presenta un ejemplo de un árbol donde el nodo de la raíz (10) se encuentra desbalanceado hacia la izquierda ($balance(10) = altura(5) - altura(15) = 2 - 0 = 2$). Como el desbalance es positivo, se mira el balance de su hijo izquierdo 5 ($balance(5) = altura(3) - altura(7) = 1 - 0 = 1$). Al ser ambos del mismo signo, se aplica rotación simple. Al estar "caído" hacia la izquierda, la rotación que se aplica es a derecha, obteniéndose el 5 en la raíz, la antigua raíz a su derecha (3), y el antiguo hijo derecho de 5 (3) como hijo izquierdo de 10. Gracias a esta rotación el subárbol ahora se encuentra balanceado ($balance(5) = 1 - 1 = 0$) y a la vez cumple con la condición de árbol binario de búsqueda.

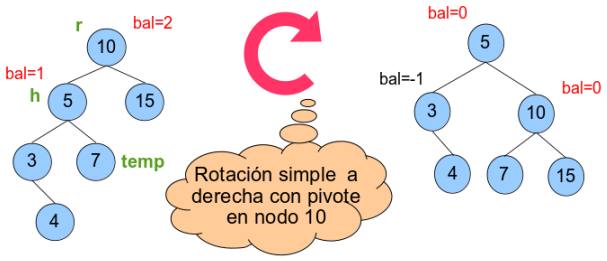


Figura 4.22: Ejemplo de rotación simple a derecha

4.3.2.3. Rotación doble derecha-izquierda

Esta rotación se aplica cuando el nodo padre está caído a la derecha (balance -2) y su hijo derecho está caído hacia el lado contrario (balance 1).

Cuando el signo del desbalance de la raíz y del hijo correspondiente a su desbalance son distintos, la rotación simple no alcanza para balancear el árbol. Un ejemplo de esto se muestra en la Figura 4.23, donde el árbol con raíz 10 está desbalanceado hacia la derecha (balance -2) y su hijo derecho (15) está desbalanceado hacia la izquierda (balance 1). Al intentar aplicar la rotación simple a izquierda, el árbol resultante queda desbalanceado de la manera contraria (su raíz hacia la izquierda y el hijo izquierdo de esta hacia la derecha) por lo tanto no se resuelve el desbalance.

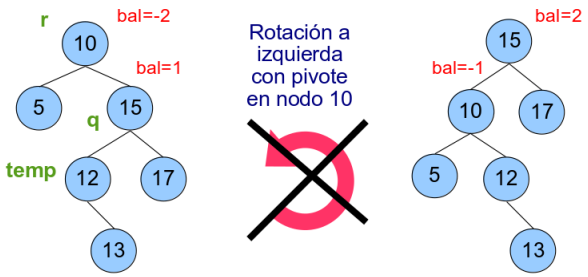


Figura 4.23: Ejemplo de rotación simple errónea

En este caso, debe aplicarse una rotación doble, que en realidad son dos rotaciones simples a diferente altura:

1. Se hace una rotación simple a derecha con pivote en el hijo 15 (para invertir el signo del desbalance)
2. A continuación se aplica la rotación a izquierda con pivote en el padre 10, que al tener ambos desbalances con igual signo, funciona perfectamente.

En la Figura 4.24 se muestra la rotación doble paso a paso.

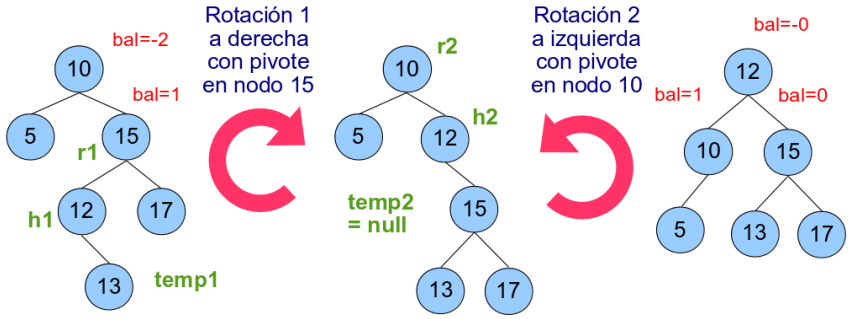


Figura 4.24: Ejemplo de rotación doble derecha-izquierda

4.3.2.4. Rotación doble izquierda-derecha

Esta rotación se aplica cuando el nodo padre está caído hacia la izquierda (balance 2) y su hijo izquierdo está caído hacia el lado contrario (balance -1).

De manera contrapuesta al ejemplo anterior, en este caso debe aplicarse una rotación doble izquierda-derecha:

1. Se hace una rotación simple a izquierda con pivote en el hijo 5 (para invertir el signo del desbalance)
2. A continuación se aplica la rotación a derecha con pivote en el padre 12.

En la Figura 4.25 se muestra dicha rotación doble, paso a paso.

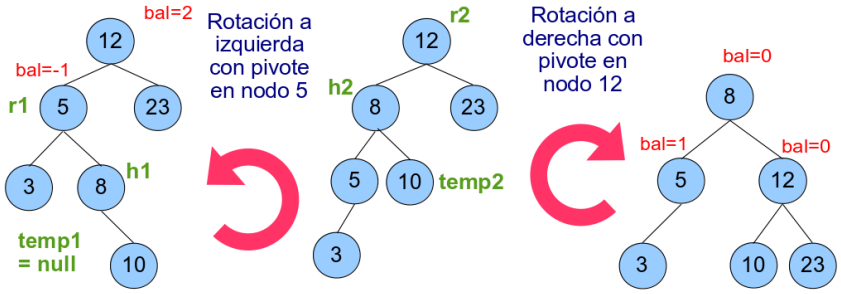
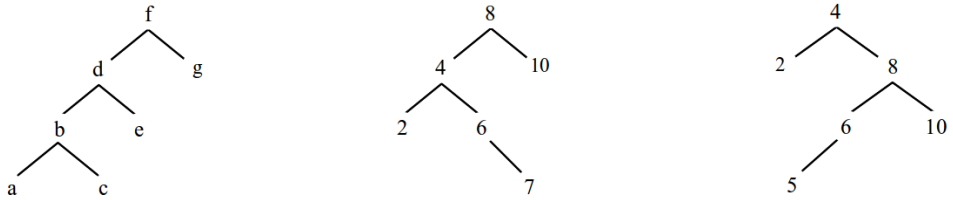


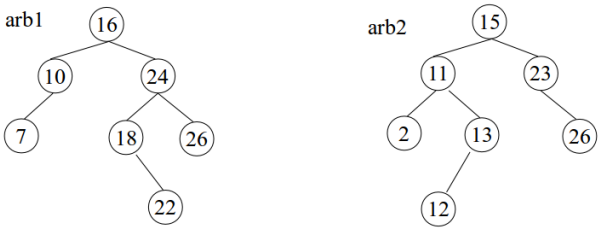
Figura 4.25: Ejemplo de rotación doble izquierda-derecha

Ejercicio 4.3: Rotaciones en árboles AVL

1. En cada uno de los siguientes árboles, indique cuál es el nodo que está desbalanceado y aplique las rotaciones necesarias para lograr balancearlos. Indique el pivote de cada rotación y muestre el resultado y los estados intermedios.



2. Dados los siguientes árboles arb1 y arb2



   a) En el árbol AVL *arb1* ¿qué debe hacer para balancearlo después de eliminar el nodo 26? Explique cuál es el tipo de desbalance que se produce, qué tipo de rotación se debe aplicar y cuál es el pivote correspondiente en cada rotación. Dibuje el árbol resultante y los pasos intermedios.

   b) Idem inciso (1) para el árbol *arb2*

   c) ¿Qué elemento debería insertar en *arb1* para que a continuación sea necesaria una rotación simple a izquierda? Indique con qué pivote es necesario realizarla.

   d) ¿Qué elemento debería insertar en *arb1* para que sea necesaria una rotación doble derecha-izquierda?

3. Mostrar un ejemplo de árbol AVL donde al eliminar un elemento dado el rebalance requiera una rotación doble izquierda-derecha. Indicar con qué pivote debe hacerse cada rotación y dibujar las rotaciones paso a paso.

4.3.3. Implementación dinámica

La implementación más eficiente de Árbol AVL es mediante nodos enlazados. Para facilitar los algoritmos de rotación se puede almacenar el balance o la altura en el nodo. Para la implementación de las operaciones se pueden copiar las operaciones de la clase ArbolBB, reescribiendo los métodos insertar y eliminar para que chequeen el balance y roten si es necesario, a la vuelta de la recursión.

En la Figura 4.26 se presenta el diagrama UML de ambas clases, almacenando la altura del nodo, que deberá mantenerse actualizada al realizar las rotaciones, y que puede re-calcularse en tiempo constante a partir de la altura de los nodos hijos.

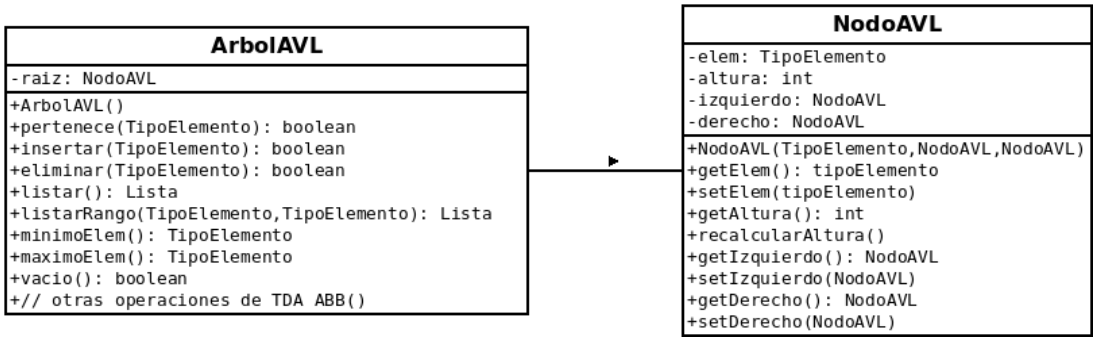


Figura 4.26: Diagrama UML de clases para árbol binario balanceado AVL

Ejercicio 4.4: Implementación de Árbol AVL

- Crear las clases ArbolAVL y NodoAVL e implementar todas las operaciones del TDA Arbol AVL presentado anteriormente.
- Realizar una clase que realice el test exhaustivo del TDA implementado.

4.3.4. Análisis de eficiencia

En cuanto a la eficiencia de las operaciones en el árbol AVL, dado que se mantienen balanceados (con una diferencia de máximo un nivel entre cualquiera de sus ramas), puede asegurarse que son, en el peor de los casos, de orden O(log n). Aunque las operaciones de inserción y eliminación tienen un costo extra (el balanceo), si el cálculo de la altura de cada nodo se hace de manera constante, el orden de insertar o eliminar se mantiene en O(log n), dado que la tarea de balanceo se hace a la vuelta de la recursión.

4.3.5. Ventajas y desventajas del árbol AVL

Como ventaja, podemos decir que el árbol AVL asegura una eficiencia logarítmica para las operaciones de búsqueda, inserción y eliminación. Como desventaja se puede mencionar la complejidad de la implementación de las rotaciones, aunque es un costo menor, comparado con la ganancia en eficiencia al almacenar conjuntos de tamaño considerable. Otra ventaja es que al ser una estructura dinámica puede crecer tanto como sea necesario. Sin embargo, en conjuntos de muchos elementos, la cantidad de niveles puede hacer un poco más lento el acceso en comparación con otras estructuras más avanzadas (árboles B, 2-3, Trie, etc.), pero estas escapan al alcance de esta materia.

4.3.6. Bibliografía recomendada sobre árbol AVL

- Joaquín Fdez-Valdivia, Apuntes de la asignatura "Estructuras de Datos", Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de Granada http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/avl.html
- Frank Pfenning, Lecture Notes on AVL Trees, Principles of Imperative Computation Lecture. Carnegie Mellon University, 2011. http://www.cs.cmu.edu/~wlovas/15122-r11/lectures/18-avl.pdf

4.4. Tabla Hash

4.4.1. Descripción

Un enfoque completamente diferente de los anteriores consiste en realizar la búsqueda de un elemento, no mediante comparaciones entre valores y manteniéndolos en orden, sino mediante una función *h(k)* que nos indique directamente la ubicación del elemento k en una tabla. Como se ve en la Figura 4.27, la función h se aplica a cada elemento del conjunto U y se obtiene su posición en la tabla T. Por ejemplo,

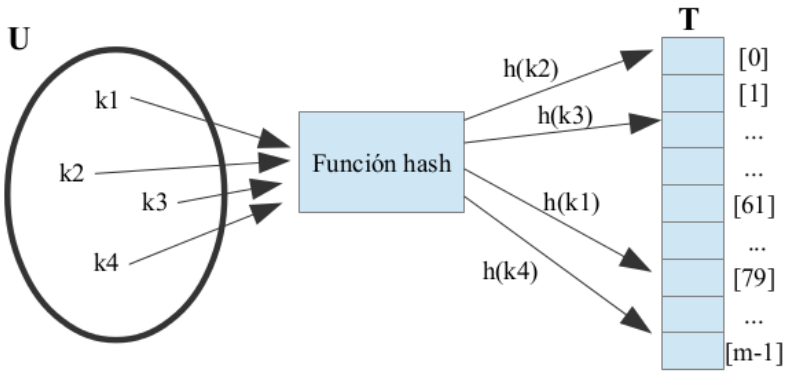


Figura 4.27: Transformación de claves mediante una función hash

la función h para el elemento con clave k1, h(k1), devuelve el resultado 79, es decir, se lo ubica en la posición 79 de la tabla; luego el elemento con clave k2 se almacena en la posición h(k2)=0; etc.

Una tabla hash se construye en base a un arreglo de celdas (*buckets*, en ingles), que será la estructura que almacene la información; y una función de dispersión o función hash. La función hash permitirá asociar cada elemento con una celda donde dicho elemento debe almacenarse o buscarse. Lo deseado es que dadas dos claves distintas, el resultado de la función hash (es decir, su posición en la tabla) sea distinto. Por el contrario, cuando dos claves distintas obtienen el mismo valor de la función hash, se dice que ocurre una *colisión*.

La función hash perfecta es aquella que es inyectiva, es decir cuando no ocurren colisiones porque $k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$. Sin embargo, las funciones que evitan las colisiones son sorprendentemente difíciles de encontrar, incluso para cantidades de datos pequeñas. Por ejemplo, la famosa *"paradoja del cumpleaños"* asegura que si en una reunión hay 23 o más personas, hay una probabilidad alta de que dos de ellas hayan nacido el mismo día del mismo mes. En otras palabras, si seleccionamos una función aleatoria que aplique 23 claves a una tabla de tamaño 365, la probabilidad de que dos claves no caigan en la misma localización es de sólo 0.4927. El objetivo entonces, será encontrar funciones hash que provoquen el menor número posible de colisiones.

Otra cosa importante es que la implementación de la tabla hash deberá estar preparada para poder almacenar elementos aunque colisionen entre sí. Con este objetivo, existen dos tipos de tablas hash, en función de cómo se resuelven las colisiones:

- *Hash abierto*: Las colisiones se resuelven insertando los elementos que colisionan en una lista. De esa forma se tiene como estructura un arreglo de listas. Al número medio de elementos por lista se le llama factor de carga y se intenta que sea lo más pequeño posible.
- *Hash cerrado*: Se utiliza un arreglo de elementos como representación, y cuando se produce una colisión se la resuelve reasignándole otro valor hash al elemento hasta que se encuentre un hueco libre.

Dichas implementaciones serán analizadas más adelante, en la Sección 4.4.4

4.4.2. Funciones *hash*

Como se ha dicho, las tablas hash dependen de las funciones que transforman valores en posiciones dentro de la tabla. Concretamente, se necesita una función que transforme los elementos (sin importar su tipo) en enteros en un rango [0..M-1], donde M es el tamaño de la tabla o arreglo. Como ya se dijo, la función debe minimizar las colisiones y debe ser relativamente rápida y fácil de calcular.

A continuación se presentan algunos ejemplos de dichas funciones:

Doblamiento. Se utiliza cuando la clave es un número entero y tiene muchos dígitos. El proceso consiste en separar la clave en varios grupos de dígitos y sumarlos entre sí. La suma obtenida es la posición asignada en la tabla.

Ejemplo: Dada la clave 163456789, se separa la clave en tres grupos: 16, 3456 y 789. Luego se suman dichos grupos y se obtiene 4261, que es el resultado de la función. Si este valor es mayor que M, para ajustarlo a la tabla se tomará el resto del resultado dividido por M (valor MOD M).

Función Cuadrado. Consiste en elevar al cuadrado la clave y luego tomar los dígitos centrales como dirección. El número de dígitos a tomar queda determinado por el tamaño M. Sea k la clave del dato buscado, la función hash queda definida por la siguiente fórmula:

$$h(k) = \text{dígitos\_centrales } (k^2)$$

Ejemplo: Sea M=100 y sea k1 = 7259, se aplica la fórmula de la siguiente forma: $k1^2 = 52.693.081$ h(k1) = dígitos_centrales(52.693.081) = 93.

Si la tabla tiene 100 posiciones, el rango de posiciones variará entre 00 y 99, luego se pueden tomar los dos dígitos centrales del cuadrado de la clave como posición del elemento en la tabla. En el ejemplo, esta posición es la 93.

División por un número primo. Se toma el mayor número primo P que sea menor a M, donde M es la cantidad de celdas de la tabla. Luego se divide el número M por P y a al resto obtenido se lo toma como resultado de la función:

$$h(k) = \text{k MOD P (donde P es el mayor primo} < \text{M y M es el tamaño de la tabla)}$$

Ejemplo: Supongamos que se tiene una tabla de 7965 posiciones. El mayor número primo menor que 7965 es 7963. Luego, sea la clave 28361, entonces 28361 = 7963 * 3 + 4472, es decir 4472 es el resto de dividir 28361 por 7963. La dirección asignada al elemento con clave 28361 será 4472. Luego puede ajustarse con la función MOD al tamaño de la tabla.

Operando con clave tipo cadena. En los casos de claves de tipo cadena de texto, la estrategia elegida es realizar la suma de los valores ASCII para cada carácter de la cadena. A dicha suma se la puede utilizar así mismo o se le puede aplicar alguna de las funciones anteriores.

```
Función h (s: cadena) : entero
    suma = 0
    Para cada carácter de la cadena de texto s
        suma = suma + ASCII(s[i])
    Sale suma
Fin función h
```

Ejercicio 4.5: Implementación de funciones hash

- Crear una clase *funciones* de tipo librería con al menos dos funciones hash distintas, una para elementos de tipo Entero y otra para elementos de tipo String.

4.4.3. Operaciones del TDA Tabla Hash

La tabla hash tiene, como se explicó antes, el propósito de facilitar la búsqueda de un elemento. Las operaciones básicas de esta estructura son:

- *constructor vacío*
  // crea una tabla sin elementos (el tamaño puede asignarse en el constructor o estar definido por defecto)
- *insertar* (elemento): *boolean*
  // recibe un elemento e intenta insertarlo en la tabla. Si todo funciona OK (no está repetido y hay lugar suficiente en la tabla) devuelve *verdadero*, si hay algún problema devuelve *falso*
- *eliminar* (elemento): *boolean*
  // recibe el elemento que se desea eliminar y se procede a quitarlo de la tabla. Si todo funciona OK (el elemento estaba cargado previamente en la tabla) devuelve *verdadero*, si hay algún problema devuelve *falso*
- *pertenece* (elemento): *boolean*
  // recibe el elemento y devuelve *verdadero* si ya está cargado en la tabla y *falso* en caso contrario
- *esVacia*(): boolean
  // devuelve *falso* si hay al menos un elemento cargado en la tabla y *verdadero* en caso contrario

- *listar* (): lista (de elementos)
  // recorre la tabla completa y devuelve una lista con los elementos que se encuentran almacenados en la tabla. Es útil para mostrar los datos sin depender del dispositivo de salida (consola, ventana, etc)

Se pueden incluir otras operaciones útiles como *clone* y *vaciar*.

4.4.4. Implementaciones de Tabla Hash

Como se ha mencionado antes, la base de la implementación de un tabla hash es siempre un arreglo de tamaño M, pero la diferencia entre las diferentes implementaciones se refieren a la estrategia utilizada para resolver las colisiones (es decir, de qué manera lidiar con los elementos cuya función hash los envía a guardar en la misma posición).

4.4.4.1. Hash abierto

La manera más simple de resolver una colisión es permitir que cada posición en la tabla almacene una lista enlazada de nodos, de manera que se guarden juntos todos los elementos cuya función hash indique esa dirección. Hay que observar que el tiempo requerido para una búsqueda dependerá de la longitud de las listas y de las posiciones de los elementos dentro de ellas. Se pueden realizar variantes dependiendo de la estrategia utilizada para almacenar los elementos que colisionan (al principio o al final de los que ya están o bien ordenados), aunque dado que las listas no deberían ser demasiado largas (se sugiere 3 elementos como máximo), se suele optar por la alternativa más simple, insertando al inicio.

En la Figura 4.28 se observa una tabla de tamaño 13 donde se han insertado los elementos 91(cuya función hash h(91) MOD M ha dado como resultado 0); 119 (donde h(119) MOD M dio como resultado 2), etc.

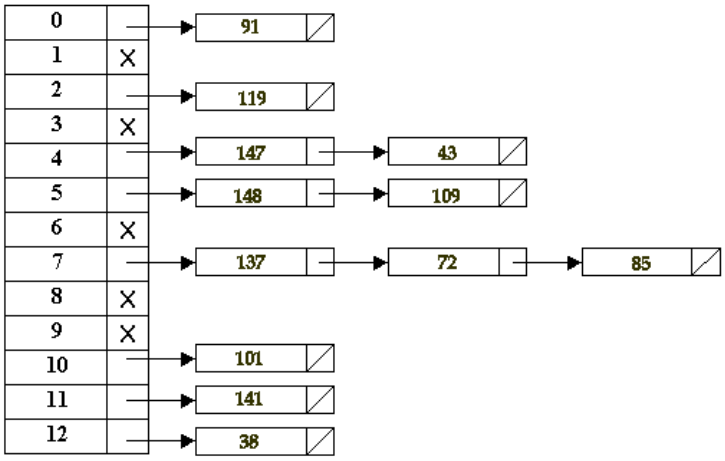


Figura 4.28: Ejemplo de tabla implementada con hash abierto

En la Figura 4.29 se presenta el diagrama UML de las clases necesarias para la implementación de hash abierto. Aunque no es parte de la implementación básica, se ha agregado un atributo *cant* en la clase TablaHash para facilitar la operación *esVacia* (y evitar recorrer toda la tabla para determinarlo). También puede aprovecharse para hacer más eficientes otras operaciones que requieren recorrer toda la tabla (como listar), ya que permite preguntar si ya se han visitado todos los elementos de la tabla y se puede parar o si es necesario seguir recorriendo.

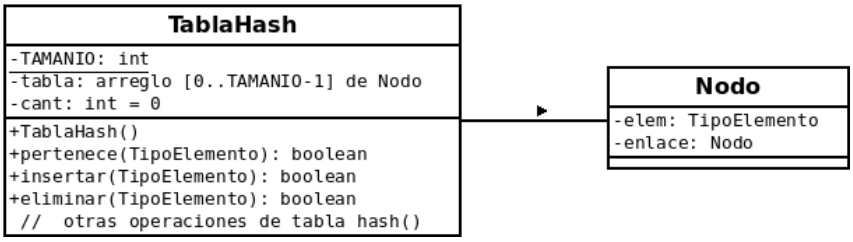


Figura 4.29: Diagrama UML de clases para hash abierto

Operaciones en hash abierto

Las operaciones de la tabla hash en esta implementación son muy sencillas, debiendo contemplar los casos de inserción en primera posición de una lista enlazada y de eliminación en cualquier posición. A modo de ejemplo se presenta el algoritmo de inserción.

Algoritmo 4.12 *Algoritmo de inserción en hash abierto*

Algoritmo *insertar* (elemento)

1. calcular posición en la tabla pos = hash(elemento) MOD tamaño de la tabla
2. buscar el elemento en la lista con cabecera tabla[pos]
3. si no lo encuentra, crear el nodo nuevo con (elemento) y enlazarlo a la lista de tabla[pos]
   sino, devolver *error (elemento repetido)*

Fin algoritmo

A continuación se muestra un ejemplo en Java del método *insertar* para la implementación del diagrama presentado en la Figura 4.29, donde se reemplazó TipoElemento por *Object,* y se utiliza la función hashCode predefinida en Java.

Algoritmo 4.13 Código Java del método *insertar* de clase TablaHash (hash abierto)

```
public boolean insertar(Object nuevoElem) {
    // primero verifica si el elemento ya esta cargado
    // si no lo encuentra, lo pone adelante del resto

    int pos = nuevoElem.hashCode() % this.TAMANIO;
    Nodo aux = this.hash[pos];
    boolean encontrado = false;

    while (!encontrado && aux != null) {
        encontrado = aux.getElem().equals(nuevoElem);
        aux = aux.getEnlace();
    }

    if (!encontrado){
        this.hash[pos] = new Nodo(nuevoElem, this.hash[pos]);
        this.cant++;
    }

    return !encontrado;
}
```

Ejercicio 4.6: Ejercicios de hash abierto

En forma gráfica, resolver:

- Sea una tabla de tamaño 10 vacía, considerar la función hash(x) h=x*7 y realizar las siguientes operaciones considerando que el elemento a insertar tiene una clave de tipo número entero:
  - Insertar 3, 5, 13, 6, 25, 35
  - Eliminar 13, 25

Crear las clases Java necesarias para:

- Implementar la clase TablaHash con *hash abierto* para elementos de un tipo dado (int o String), que utilice alguna de las funciones hash implementadas en la librería «funciones» creada en el Ejercicio 4.5. Hacer un test que pruebe todas las operaciones, forzando colisiones, insertando elementos repetidos, eliminando y buscando elementos que existan o no.
- Crear otra versión de Tabla Hash con *hash abierto* para elementos de tipo Object y que como función use el método *hashCode* de la clase Object. Implementar un test que pruebe esta versión con elementos String e Integer y también con una clase propia que tenga implementados los métodos hashCode e equals para su clave (Ej: Alumno con clave legajo, Automóvil con clave patente, etc).

4.4.4.2. Hash cerrado

Otra alternativa, muy utilizada, para implementar hash se llama de "hash cerrado". En este tipo de implementación, se utiliza un arreglo que guarda directamente un elemento en cada celda (en lugar de un enlace a un nodo como el hash abierto). Como se puede ver en la figura a continuación, usando el mismo ejemplo con los elementos 91 (que se debe almacenar en la posición 0) y 119 (que se debe almacenar en la posición 2) de la figura anterior, ahora se almacenan directamente en la celda correspondiente a la aplicación de la función hash.

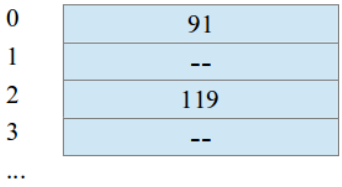


Figura 4.30: Ejemplo de tabla implementada con hash cerrado

Sin embargo, ante una colisión no se puede tener dos elementos en la misma celda, por lo tanto se debe usar una estrategia de *rehashing*, que consiste en utilizar una función adicional hasta encontrar una casilla que esté desocupada y acomodar el elemento ahí.

Hay varias estrategias de rehashing. Como ejemplo, vamos a analizar dos de ellas:

Rehashing lineal. La estrategia más simple consiste en utilizar una función que si la posición i está ocupada, la función de rehashing le agregue uno más, y siga haciéndolo hasta que encuentre una casilla vacía.

$$rh_i(x) = (h(x) + (i-1))$$

donde i es el número de intento de rehashing; $i=2,3,...$

Al utilizar el MOD para ajustar la posición al tamaño de la tabla, cuando se alcanzó la última posición de la tabla, se vuelve a la posición 0, recorriéndolo de manera circular.

Ejemplo: Consideremos una tabla de tamaño 10 vacía y una función hash muy simple que devuelve h(x)=x. Consideremos que los elementos llegan en el siguiente orden: 83, 45, 376, 25, 85.

El primero en llegar es 83, h(83)= 83, luego se hace 83 mod 10 = 3, como la celda 3 está vacía ocupa su posición. El segundo en llegar es 45, h(45)=45, luego 45 MOD 10 =5; como la celda 5 está vacía ocupa su posición. De la misma manera llega el 376 y se ubica en la posición 6.

A continuación llega el elemento 25:

- Intento 1: calcula la función hash h(25)=25, luego 25 MOD 10 = 5 y la encuentra ocupada, por lo tanto es preciso realizar un rehashing.
  - Intento 2: calcula la función de rehash rh2(25)=5+(2-1)=6. Pero la celda 6 también está ocupada, por lo tanto debe intentar de nuevo
  - Intento 3: rh3(25)=5+(3-1)=7. Como está libre, el elemento 25 se almacena en la posición 7 (Figura 4.31A).

¿Cómo es el procedimiento de rehashing para el elemento 85 que llega a continuación? Ocurrirá que 85 colisionará 3 veces: primero con 45, luego con 376, y también con 25, sumando de a uno cada vez, hasta que pueda encontrar una celda vacía en la posición 8 (Figura 4.31B).

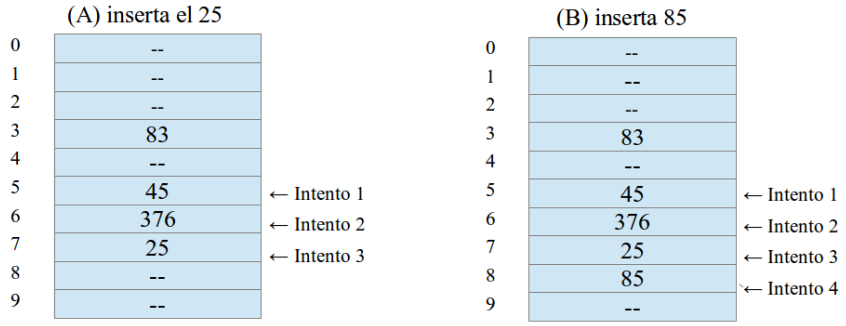


Figura 4.31: Ejemplo de inserción utilizando rehashing lineal

Rehashing doble. Como vimos en el ejemplo anterior, la estrategia lineal tiene el problema de que los elementos que colisionan quedan agrupados alrededor de esa área y esto suele generar más colisiones a futuro. Para evitarlo, se suele utilizar otra estrategia de rehashing que en lugar de incrementar siempre en 1, incrementa un número distinto de posiciones para cada elemento. Para ello necesitamos dos funciones, la función hash habitual *h(x)*, que devuelve la posición inicial, y una función h2(k) que devuelve un incremento fijo para cada elemento. En caso de múltiples colisiones el valor se multiplica por el número de intento. La función queda de la siguiente manera:

$$rh_i(x) = (h(x) + (i-1) * h2(x))$$

donde nuevamente, i es el número de intento de rehashing; $i=2,3,...$

Veamos un ejemplo al cambiar la estrategia de rehashing: Consideremos la tabla de tamaño 10 vacía y la función hash $h(x)=x$ del ejemplo anterior. Luego agregamos la función de rehashing doble $h2(x)=x \ div \ 10$ (donde div representa la división entera). Veamos la inserción de los mismos elementos del ejemplo anterior: 83, 45, 376, 25, 85.

El primero en llegar es 83, h(83)=83, luego 83 mod 10=3, como la celda 3 está vacía ocupa su posición. El segundo en llegar es 45, se ubica en la celda 5 que está vacía. Luego el 376 se ubica en la posición 6.

Analicemos que pasa cuando llega el elemento 25 (Ver Figura 4.32A):

- Intento 1: aplica la función h(25), ajusta con MOD 10 y encuentra ocupada la posición 5.

Entonces, debe aplicar rehashing doble, por lo tanto calcula el incremento para 25: h2(25)=25 div 10=2.

- Intento 2: obtiene la próxima posición calculando rh2(25)=5+(2-1)*2=5+2=7, que es el equivalente a sumarle el incremento (2) a la posición inicial. Como la celda 7 está libre, el elemento 25 se almacena allí.

¿Qué pasa con el elemento 85?. En este caso se deben realizar tres intentos (Ver Figura 4.32B). Primero, 85 colisiona con 45, por lo tanto es preciso realizar rehashing. Se calcula la función incremento con h2(85)=85 div 10=8. Luego se intenta el rehashing rh2(85)=5+(2-1)*8=5+8=13. Como 13 excede el tamaño de la tabla, se calcula 13 mod 10 y da la posición 3, que está ocupada por 83. Finalmente, se vuelve a aplicar el rehash para el intento 3, rh3(85)=5+(3-1)*8=5+16=21 (o su equivalente, que es sumar 8 a la posición anterior), y luego se ajusta al tamaño de la tabla haciendo 21 mod 10, que da la posición 1 que está libre.

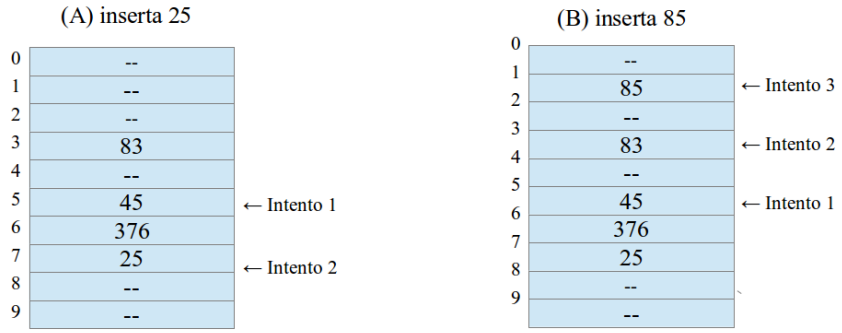


Figura 4.32: Ejemplo de inserción utilizando rehashing doble

Algoritmos de las operaciones principales

A continuación se presentan los algoritmos de inserción, búsqueda y eliminación en hash cerrado. En estos algoritmos no se establece la estrategia de rehashing, por lo que al momento de implementarlos puede utilizarse tanto lineal como doble.

Algoritmo 4.14 Algoritmo de búsqueda en hash cerrado

Algoritmo *pertenece* (elemento):boolean

1. incremento = {inicializar según la estrategia de rehash}
2. pos = h(elemento) MOD tamaño de la tabla
3. Verificar el elemento en tabla[pos]
   - si no es el elemento buscado y la celda no está vacía,
     calcular la próxima dirección pos = (pos + incremento*nro intento) MOD tamaño de la tabla
   - si es el elemento buscado cortar la búsqueda
4. Repetir el paso 3 hasta que encuentre el elemento o una celda vacía

Fin algoritmo

Algoritmo 4.15 Algoritmo de inserción en hash cerrado

Algoritmo *insertar* (elemento)

- Si el elemento ya está cargado en la tabla devolver *error*
  sino
  1. inicializar variables
     - pos = h(elemento) MOD tamaño de la tabla
     - incremento = {inicializar según estrategia de rehash}
  2. si la celda tabla[pos] está disponible colocar el elemento allí
     sino calcular la próxima dirección pos = (pos + incremento*nro intento) MOD tamaño de la tabla
  3. repetir el paso 2 hasta que consiga una celda libre

Fin algoritmo

Algoritmo de eliminación

Al analizar la eliminación de un elemento nos encontramos con algunos casos especiales, que es necesario observar mediante ejemplos. Supongamos que en la tabla que teníamos antes se decide borrar el elemento 83. Utilizando el mismo mecanismo que al momento de insertar, aplicamos la función hash h(83)=83 mod 10=3, y como es el elemento buscado marca la celda como vacía (Ver Figura 4.33A). Al intentar más adelante buscar el elemento 85, se encontrará primero que en la posición 5 está el elemento 25, pero como no es el 85 sigue buscando con la función de rehash y llega a la posición 3. Como la celda está vacía, no lo encuentra y retorna falso es decir, que el elemento no fue encontrado (Ver Figura 4.33B).

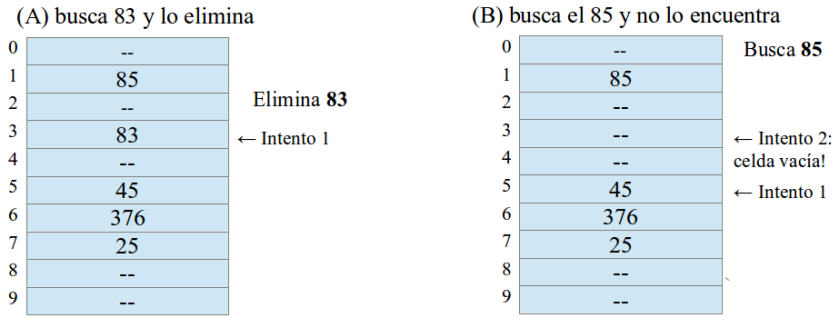


Figura 4.33: Ejemplo de eliminación sin usar marca de borrado (produce error)

Sin embargo, no es correcto que responda que el elemento no exista en la tabla, sino que está más allá del lugar donde se está viendo (ya que al insertar hubo 2 colisiones). Por este motivo es necesario identificar estados distintos cuando una celda está vacía porque nunca fue utilizada o porque hubo algo que luego fue borrado. Por lo tanto es muy importante que la implementación tenga en cuenta esta diferenciación. En la Figura 4.34A se ve el caso de búsqueda de 85 tras la eliminación de 83 donde la búsqueda arroja el resultado correcto. Luego, en la Figura 4.34B, se muestra el caso que busca un elemento 35, que nunca fue insertado. Al buscarlo primero en la posición 5 encuentra un elemento que no es el buscado, pero al hacer rehashing y encontrar la celda 8 vacía, puede determinar que el 35 no está en la tabla.

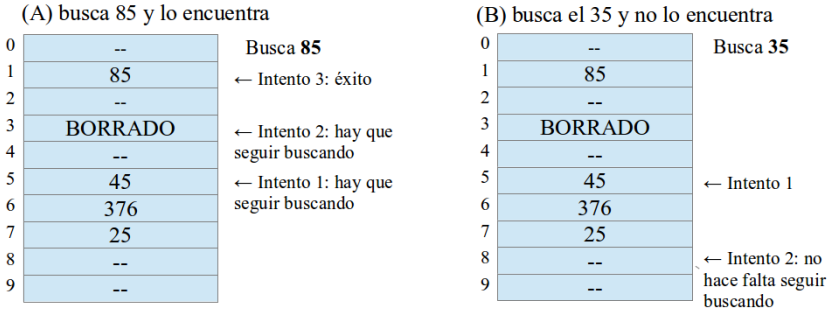


Figura 4.34: Ejemplo de eliminación utilizando marca de borrado

A continuación se presenta el algoritmo de eliminación utilizando la marca de borrado.

Algoritmo 4.16 Algoritmo de eliminación en hash cerrado

Algoritmo *eliminar* (elemento):boolean

1. Inicializar variables
   - incremento = {inicializar según la estrategia de rehash}
   - pos = h(elemento) MOD tamaño de la tabla
   - intento = 1
2. Verificar el elemento en tabla[pos]
   - si no es el elemento buscado y la celda no está vacía, calcular la próxima dirección pos = pos + incremento*intento
   - si es el elemento buscado cortar la búsqueda y marcar la celda como *borrada*
3. Repetir el paso 2 hasta que encuentre el elemento o una celda vacía y retorne el resultado correspondiente

Fin algoritmo

Respecto a la implementación en Java, se pueden elegir distintas estrategias para marcar la celda como borrada o vacía. Una estrategia puede ser elegir un elemento que no pueda ser parte del conjunto de elementos a cargar y utilizarlo como representante de borrado. Ejemplo: si los elementos a guardar en la tabla son siempre enteros positivos, se puede reservar el 0 (cero) para marcar que la celda está vacía (nunca usada) y -1 para indicar que fue borrada. Para casos donde no se puede saber el universo de los elementos a guardar, conviene guardar en el arreglo un tipo celda que almacena el elemento y un atributo estado que por defecto indicará que la celda está libre. Para definir los estados, se pueden utilizar

constantes (VACIO=0, BORRADO=-1, OCUPADO=1). En la Figura 4.35 se presenta el UML de las clases necesarias para la implementación de hash cerrado de acuerdo a esta consigna. En el constructor de la clase será necesario llenar el arreglo con objetos de tipo Celda marcados todos como *vacíos*. Como en la implementación de hash abierto, aunque no es parte de la implementación básica, se ha agregado un atributo *cant* en la clase TablaHash para hacer más eficientes algunas operaciones.

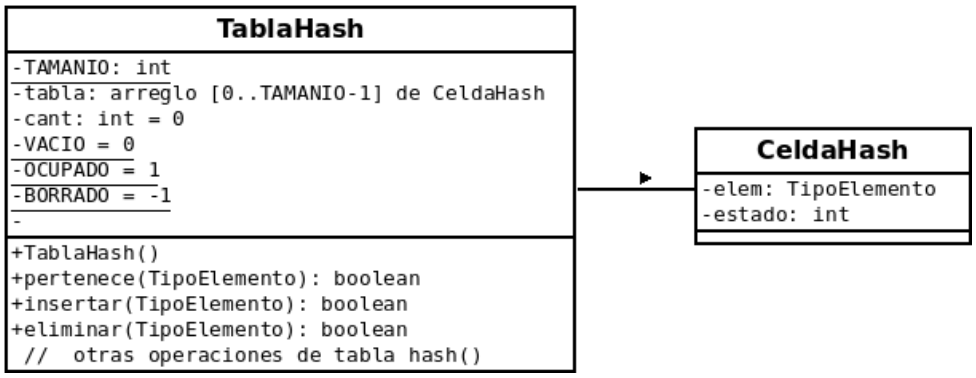


Figura 4.35: Diagrama UML de clases para hash cerrado

A continuación se muestra un ejemplo en Java del método *eliminar* para la implementación del diagrama presentado en la Figura 4.35, donde se reemplazó TipoElemento por *Object,* de manera similar a las implementaciones vistas anteriormente de lista, pila, cola, árboles binarios y genéricos, ya que en la Tabla Hash no es necesario que los elementos mantengan un orden, sino que solo deben cumplir con la condición de tener la función *hashCode* de Object implementada. Dicha función hashCode está implementada por defecto en la clase String y wrappers de Java y para clases propias se puede generar automáticamente usando los generadores automáticos que los IDE suelen incluir habitualmente.

Algoritmo 4.17 Código Java del método *eliminar* de clase TablaHash (hash cerrado)

```
public boolean eliminar(Object buscado) {
    // calcula posicion inicial e incremento
    int pos = buscado.hashCode() % this.TAMANIO;
    int incremento = funciones.rehash(buscado) % this.TAMANIO;

    boolean encontrado = false;
    int intento = 1;

    // busca el elemento hasta encontrarlo o encontrar una celda vacia
    // o para despues de TAM intentos
    while (!encontrado && intento < this.TAMANIO
            && this.hash[pos].getEstado() != VACIO) {

        if (this.hash[pos].getEstado() == OCUPADO) {
            encontrado = this.hash[pos].getElem() == buscado;
            if (encontrado) {
                // si lo encuentra lo marca y para el ciclo
                this.hash[pos].setEstado(BORRADO);
                this.cant--;
            }
        }

        pos = (pos + intento * incremento) % this.TAMANIO;
        intento++;
    }

    return encontrado;
}
```

Ejercicio 4.7: Ejercicios de hash cerrado

En forma gráfica, resolver:

- Sea una tabla de tamaño 10 vacía, usando la función hash *h(x)=x*3* y la función de rehash doble *rh(x)=x MOD 7 + 1*, realizar las siguientes operaciones considerando que el elemento a insertar tiene una clave de tipo número entero:
  - insertar(4); insertar(15); insertar(14); insertar(23); insertar(13); insertar(35);
  - buscar(14); buscar(35); buscar(65);
  - eliminar(14); eliminar(35);
  - buscar(14); buscar(35); buscar(65);
  - insertar(24); insertar(16);
  - buscar(14); buscar(35);

Implementar en Java:

- Crear una clase TablaHash que utilice la estrategia de hash cerrado y rehashing doble.
- Realizar una clase de test para la clase anterior.
- El algoritmo básico de inserción (4.15) no considera que la tabla puede estar llena o casi llena, y que puede no alcanzarse nunca una posición vacía, por lo que, si la variable éxito es siempre falsa, la búsqueda se convertirá en un bucle infinito. Proponga una manera eficiente para evitarlo y mejore la implementación anterior.

4.4.5. Análisis de eficiencia

Como sabemos, el arreglo es una estructura estática que permite un tiempo de acceso constante a cualquier posición, por lo tanto si la función hash es simple de calcular y produce muy pocas colisiones, el tiempo medio de recuperación de la información será constante (de orden *O(1)*), es decir que el tiempo para insertar, buscar y/o eliminar un elemento no dependerá del tamaño de la tabla ni del número de elementos almacenados en la misma. Sin embargo, esta eficiencia depende siempre de que la función hash distribuya uniformemente las claves. Si la función está mal diseñada o no es la adecuada para el conjunto de elementos a almacenar, se producirán muchas colisiones y la eficiencia será más pobre.

4.4.6. Ventajas y desventajas de las tablas hash

La principal ventaja de las tablas hash es que el acceso a los datos suele ser muy rápido, si se cumplen las siguientes condiciones:

- Una razón de ocupación no muy elevada (a partir del 75 % de ocupación se producen demasiadas colisiones y la tabla se vuelve ineficiente).
- Una función hash que distribuya uniformemente las claves.

Sin embargo, también cuenta con las siguientes desventajas:

- *Problema con conjuntos que crecen en tamaño.* Si la cantidad de elementos a almacenar crece considerablemente, será necesario ampliar el espacio de la tabla. Esto requiere crear otro arreglo (más grande) y recorrer el arreglo antiguo completamente, aplicando la nueva función hash para cada elemento antes de pasarlo a la tabla nueva. Este proceso se llama de reasignación.
- *Es poco eficiente para operaciones que necesitan recorrer todos los elementos*, ya que requiere visitar todas las celdas de la tabla, estén ocupadas o no. Por ejemplo, buscar el elemento más pequeño o más grande de la tabla, poner todos los elementos en una lista, o buscar todos los elementos en un rango.
- *Se desaprovecha memoria.* Si se reserva espacio para todos los posibles elementos, se consume más memoria de la necesaria.

4.4.7. Bibliografía sobre tablas hash

- A.V. Aho, J.E. Hopcroft y J.D. Ullman, Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988. (Capítulo 4, Sección 7)