Índice

Métodos de ordenamiento

Básicos O(n²)

Inserción

En cada iteración se inserta un elemento del vector no ordenado en la posicion correcta dentro del vector ordenado.

```
xxxxxxxxxx
int vector[] = {11, 32, 1, 26, 17, 4};
int aux, elementos = 6;
for (int i = 0; i < elementos; i++) {
    aux = vector[i];
    j = i - 1;
    while ((vector[j] > aux) && (j >= 0)){
        vector[j + 1] = vector[j];
        j--;
    }
    vector[j + 1] = temp;
}
```

Selección

En cada iteración se selecciona el menor elemento del vector no ordenado y se intercambia con el primer elemento de este mismo vector.

```
xxxxxxxxxx
int vector[] = {11, 32, 1, 26, 17, 4};
int aux, elementos = 6;
for (int i = 0; i < elementos - 1; ++i) {
    int min = i;
    for (int j = i + 1; j < elementos; ++j) {
        if (vector[min] > vector[j])
            min = j;
    }
    aux = vector[i];
    vector[i] = vector[min];
    vector[min] = aux;
}
```

Burbujeo

El método recibió este nombre porque los elementos más pequeños "burbujean" hasta el comienzo del vector, y los más grandes se "hunden" hasta el final. En la primera iteración se comparan los dos primeros elementos, si el segundo es mayor al primero se mantiene así, de lo contrario se intercambian; luego se compara el segundo con el tercero y se repite sucesivamente.

```
xxxxxxxxxx
int vector[] = {11, 32, 1, 26, 17, 4};
int aux, elementos = 6;
for(int i = 0; i < elementos; i++) {
    for(int j = 0; j < elementos - i; j++) {
        if(vector[j] > vector[j + 1]) {
            aux = vector[j];
            vector[j]=vector[j + 1];
```

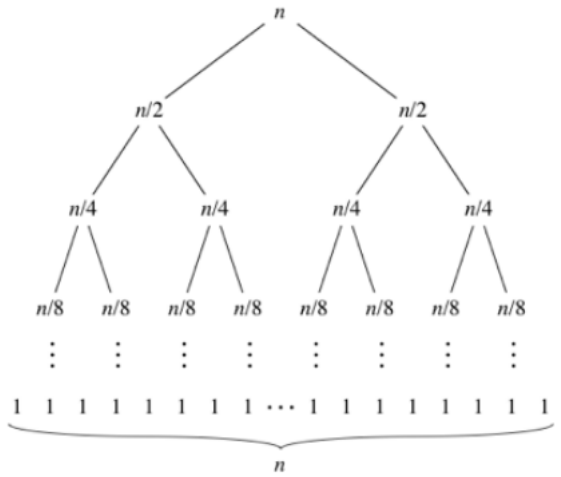
```
            vector[j + 1]=aux;
        }
    }
}
```

Divide y vencerás

Mergesort O(n · log 2 (n))

1. Dividir el vector en 2.
2. Ordenar recursivamente cada mitad.
3. Combinar las mitades ordenadas, para esto se utiliza el algoritmo de merge visto en Algo I.

Para calcular el costo temporal podemos pensar que en el primer merge se recorre el vector entero y hacer las comparaciones correspondientes, revisamos los n elementos del vector. En el segundo merge se repite lo mismo, y así sucesivamente. Podríamos decir que este proceso se hace k veces, por lo tanto el orden seróa O(n · k) pero ¿cuánto vale k? Nos podemos ayudar viendo el siguiente árbol, que representa el tamaño del problema



Viendo eso podemos deducir que k = log 2 n (porque el vector se divide recursivamente en 2 hasta llegar a 1).

```
xxxxxxxxxx
void mergesort(double lista[], int elementos) {
    if (elementos != 1) {
        int mitad = elementos / 2;
        double *izquierda = new double[mitad],
               *derecha = new double[elementos - mitad];
        for (int i = 0;  i < mitad;  i++)
            izquierda[i] = lista[i];
        for (int i = mitad;  i < elementos;  i++)
            derecha[i - mitad] = lista[i];
        mergesort(izquierda, mitad);
        mergesort(derecha, elementos - mitad);
        merge(lista, izquierda, derecha, mitad, elementos - mitad);
    }
}


void merge(double lista[], double izquierda[], double derecha[], int elementosIzq, int elementosDer) {


    int indiceIzq = 0,
            indiceDer = 0,
            indiceLista = 0;


    while ((indiceIzq < elementosIzq) && (indiceDer < elementosDer)) {
        if ( izquierda[indiceIzq] <= derecha[indiceDer]) {
            lista[indiceLista] = izquierda[indiceIzq];
            indiceIzq++;
        }
```

```
        else {
            lista[indiceLista] = derecha[indiceDer];
            indiceDer++;
        }
        indiceLista++;
    }

    while (indiceIzq < elementosIzq) {
        lista[indiceLista] = izquierda[indiceIzq];
        indiceIzq++;
        indiceLista++;
    }

    while (indiceDer < elementosDer) {
        lista[indiceLista] = derecha[indiceDer];
        indiceDer++;
        indiceLista++;
    }
}
```

Quicksort $O(n^2)$

1. Elegir un pivote. Hay muchas formas de hacer esto: puede ser un elemento random, el primero, el último, podemos recorrer todo el vector para elegir el mejor valor posible, etc.
2. Dividir el vector en 2 recursivamente. Hay que tener en cuenta que al hacer la division deben quedar en una mitad todos los elementos menores al pivote, y en la otra todos los mayores. Al hacer la división podemos agregar el pivote en alguno de los vectores (al final del izquierdo o al principio del derecho) o dejarlo afuera.
3. Combinar las mitades ordenadas

   Si el pivote quedo afuera: vector_izquierdo - pivote - vector_derecho

   Si el pivote está en algún vector: vector_izquierdo - vector_derecho

El costo de este algoritmo en el **mejor caso** es el mismo que el del mergesort **$O(n \cdot log\ 2\ (n))$**, pero en el **peor caso** (si elegimos por ejemplo el menor o mayor elemento del vector) será de **$O(n^2)$**

```
xxxxxxxxxx
void quicksort(double vector[], int primero, int ultimo) {
    int pivote;
    if (primero < ultimo) {
        pivote = dividirVector(vector, primero, ultimo);
        quicksort(vector, primero, pivote - 1);
        quicksort(vector, pivote + 1, ultimo);
    }
}

int dividirVector(double vector[], int primero, int ultimo) {
    int pivote = vector[ultimo],
        i = primero;

    for (int j = primero; j <= ultimo; j++) {
        if (vector[j] < pivote) {
            i++;
            intercambiar(vector[i], vector[j]);
        }
    }

    intercambiar(vector[i + 1], vector[ultimo]);
    return i;
}
```

```
void intercambiar(double &a, double &b) {
    int aux = a;
    a = b;
    b = aux;
}
```

Heapsort O(n · log n)

Nota: para entender este ordenamiento con mayor facilidad recomiendo ver los apuntes de árboles, la sección de heap

1. Construir el heap: son n operaciones de inserción sobre un árbol parcialmente ordenado => el costo es de **O(n · log n)**
2. Extraer el menor/mayor elemento del heap: son n operaciones de borrado sobre un árbol parcialmente ordenado => el costo es de **O(n · log n)**

```
xxxxxxxxxx
void heapsort(double vector[], int elementos) {
    for (int i = (elementos / 2) - 1; i >= 0; i--)
        armarHeap(vector, elementos, i);

    for (int i = elementos - 1; i >= 0; i--) {
        intercambiar(vector[0], vector[i]);
        armarHeap(vector, i, 0);
    }
}

void armarHeap(double vector[], int elementos, int posRaiz) {
    int mayor = posRaiz;
    int hijoIzq = 2 * posRaiz + 1;
    int hijoDer = 2 * posRaiz + 2;

    if (hijoIzq < elementos && vector[mayor] < vector[hijoIzq])
        mayor = hijoIzq;

    if (hijoDer < elementos && vector[mayor] < vector[hijoDer])
        mayor = hijoDer;

    if (mayor != posRaiz) {
        intercambiar(vector[posRaiz], vector[mayor]);
        armarHeap(vector, elementos, mayor);
    }
}

void intercambiar(double &a, double &b) {
    int aux = a;
    a = b;
    b = aux;
}
```

Shellsort

Mejora del ordenamiento por inserción (básico). En este caso se comparan elementos separados por varias posiciones y en cada iteración da saltos cada vez menores se va ordenando el vector.

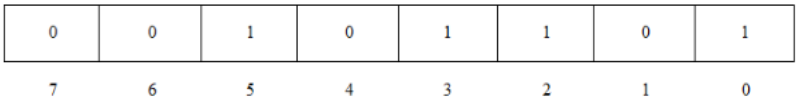
```
xxxxxxxxxx
void shellsort(double vector[], int elementos) {
    for (int salto = elementos / 2; salto > 0; salto /= 2) {
        for (int i = salto; i < elementos; i++) {
            for (int j = i; j >= salto; j -= salto)
                if (vector[j] < vector[j - salto])
                    intercambiar(vector[j], vector[j - salto]);
        }
    }
```

}

Array de bits

Un array de bits es un arreglo consecutivo de memoria donde se almacenan bits. Se utilizan para indicar si un elemento está o no en un conjunto pero ¿de qué manera? Bueno cada elemento se corresponde con el subíndice del array y lo que se guarda en cada posición es un 0 o un 1 (por convención 1 es que está y 0 que no está).

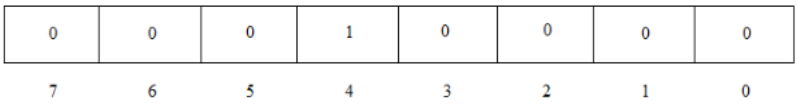
Entonces a partir de la siguiente imagen:



Podemos decir que 0, 2, 3 y 5 están en el conjunto mientras que 1, 4, 6 y 7 no.

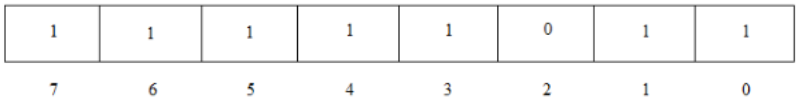
Es importante entender que no se puede manipular cada bit por separado, y que necesitamos una máscara (implementada también a través de bytes) que nos permiten modificar el valor de un bit en particular.

Por ejemplo si quiero **agregar** el número 4 al conjunto necesito una máscara como la que se ve a continuación:



Y luego realizar una operación *or* con el array original.

Para **eliminar** el número 2 tendría que crear una máscara así:



Y luego realizar una operación *and* con el array original.

Hashing

Algunas definiciones:

- Una ***tabla de hash*** es una estructura de datos que asocia claves con valores que funcionan a modo de direcciones en la misma. Si la cantidad de claves es *n*, el tamaño de la tabla *t* deberá ser aproximadamente 0,8 y esta proporción se llama factor de carga λ => λ = n/t , λ <= 0,8
- Una ***funcion de hash*** toma la clave del dato y devuelve un valor, que será el índice de entrada de la tabla. Una buena funcion tiene buena disperción y evitará lo más posible las colisiones entre claves
- Si k es una clave y p una posicion de la tabla decimos que *h: K -> P / p = h(k)*

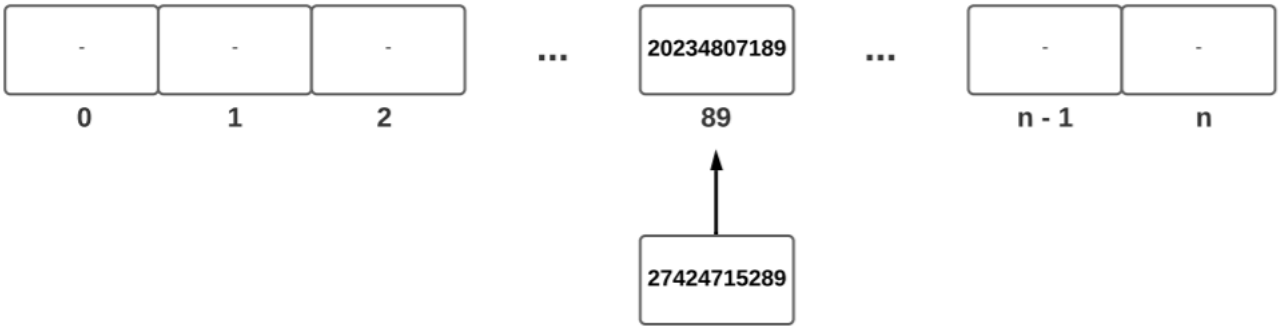
Hay que tener en cuenta que si las claves no son números enteros hay que convertirlas. Si fueran letras o palabras podriamos por ejemplo tomar los valores ASCII y sumarlos. Un ejemplo podría ser:

"ab" = 97 x 128 + 98 x 128 = 12416 + 12544 = 24960

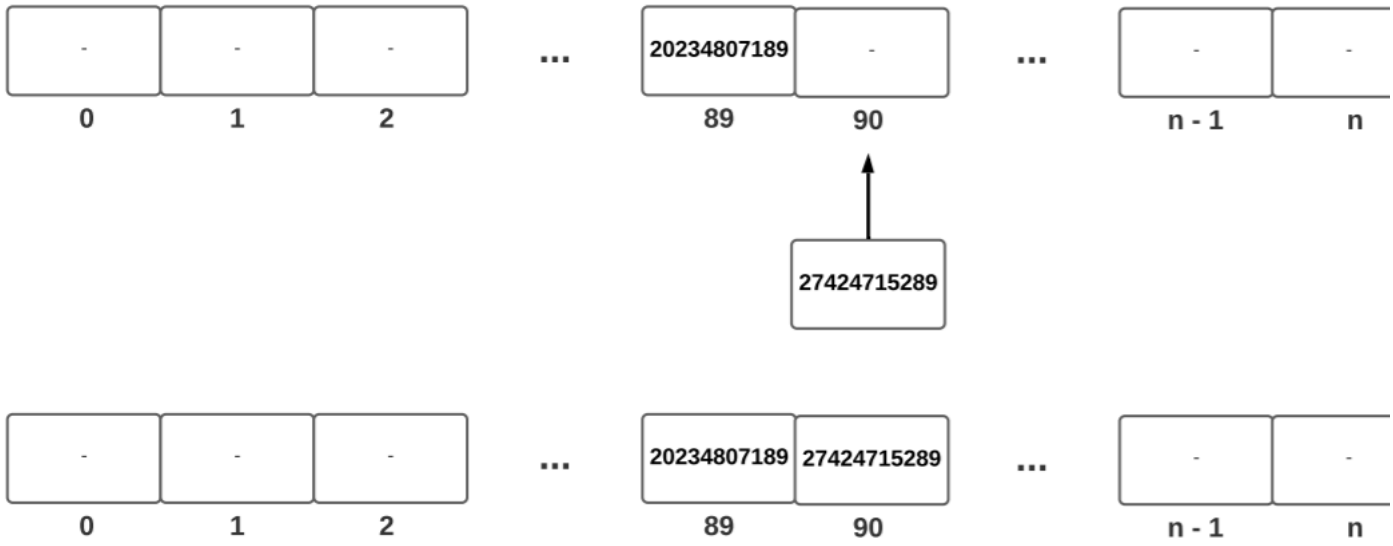
Ahora bien ¿que aplicación le podemos dar a esto? ¿para qué nos sirve? Supongamos que queremos guardar los numeros de CUIL de un grupo de personas. Lo más eficiente para buscar o eliminar un dato sería que cada número de CUIL se guarde en la posicion del vector que se corresponda, de esta manera si queremos eliminar el número 20234807189 debería acceder a la posición 20234807189 del vector y asignarle un valor nulo. El problema que conlleva esto es el costo espacial, porque si solo necesitamos guardar *algunos* vamos a tener un vector de 44.500.000.000 posiciones lleno de nulos.

Entonces ¿qué podemos hacer? Aplicar una función de hashing que nos convierta ese 20234807189 en un número más chico, y guardar ahí el CUIL. Supongamos que aplicamos la función de extracción (que se explicará más adelante) y tomamos los últimos 2 números. Con esto nos quedaría el número 20234807189 guardado en la posición 89 en vez de en la 20234807189.

¿Qué pasa si ahora quiero agregar el CUIL 27424715289? Si vuelvo a tomar los últimos 2 números me queda de nuevo el 89, pero no lo puedo guardar ahí porque ya guardé el anterior! Esto se llama colisión y más adelante vamos a ver bien las distintas formas de tratarlas.



Ahora lo primero que podríamos pensar es "bueno si la 89 está ocupada, lo ponemos en la 90 y listo"



Esta resolución es de direccionamiento abierto lineal, y si bien funciona hay opciones mejores.

Otra opción es buscar otra función para obtener la clave y en vez de usar la extracción usar la radix-transformation o el folding, pero la realidad es que no podríamos garantizar que no van a producirse colisiones. Quizás no pase con estos dos valores, pero probablemente pase con otros.

Funciones

División

Es una de las funciones de dispersión más simple: se divide la clave *k* por la cantidad de posiciones de la tabla *t* y se toma el *resto* entonces

$p = h(k) = k \% t$

Se recomienda que el valor de *t* sea un número primo.

Ejemplo:

1. Hay que almacenar 5000 claves
2. Calculo el tamaño de tabla *t* => *n = 5000* con λ = 0,8 tenemos que *t = 5000 / 0,8 = 6250*
3. El numero primo más cercano a *t* es 6257 => p = k % 6257
4. Ingreso la clave 113521 en la posicion *p = 113521 % 6257 =* ***895***
5. Ingreso la clave 28413 en la posicion *p = 28413 % 6257 =* ***3385***
6. Ingreso la clave 44694 en la posicion *p = 113521 % 44694 =* ***895*** => se produce una colisión. Veremos como solucionar esto más adelante

Multiplicación

Se multiplica a la clave *k* por un valor *A* tal que *0 < A < 1*, se toma la parte fraccionaria del resultado y se la multiplica por *t* (el tamaño de la tabla). El resultado se redondea y obtenemos la posición final

$p = h(k) = t \cdot ((k \cdot A) \bmod 1)$

Logra mejor dispersión que la división

Folding

Se toma la cantidad de dígitos que tiene el tamaño de la tabla, y se divide la clave de manera tal que cada parte tenga esa cantidad de dígitos (o menos), y luego se suma cada una de esas partes.

Ejemplo:

1. El tamaño de la tabla es 789 => nos quedamos con que tiene 3 dígitos
2. El número de CUIT 23-31562313-7 podemos dividirlo en 4 partes tomando de a 3 dígitos nos quedaría: 233 - 156 - 231 – 37
3. Sumamos los valores: 233 + 156 + 231 + 37 = **657**

4. Si el tamaño de la tabla fuera menor, aplicamos la función división

Mid-square

Se eleva a la clave *k* al cuadrado y se toman los dígitos centrales

Ejemplo:

1. La clave es 1536 => $1536^2 = 2359296$
2. Nos quedamos con los dígitos centrales: **592**
3. Si el tamaño de la tabla es menor, se aplica la función división

Extraction

Se extrae una parte de la clave. Hay que ser cuidadosos con los dígitos a extraer porque si son por ejemplo números de CUIT/CUIL y se toman los primeros 4 los valores siempre empiezan con 20, 23, 24 y 27 pueden producirse muchas colisiones.

Radix Transformation

Se cambia la base de la clave

Ejemplo:

1. La clave es 425 y se va a hacer el cambio a base 16
2. La nueva clave es $k' = 4 \cdot 16\ 2 + 2 \cdot 16 + 5 =$ ***1061***
3. Si el tamaño de la tabla es menor, se aplica la función división

Colisiones

Las colisiones se producen cuando más de una clave devuelven el mismo valor luego de aplicar alguna función de hash, a continuación vamos a ver varios métodos para manejar estas colisiones.

Open addresing

Los métodos de direccionamiento abierto u open addresing guardan toda la informacion en la misma tabla y si se produce una colisión se busca una nueva posición tomando alguna nueva función. Entonces si *h(k)* está ocupada prueba con *h(k) + p(1)*, luego con *h(k) + p(2)* , *h(k) + p(3)*, etc. donde *p* es la nueva función. Al finalizar siempre se aplica la función división.

Linear probing

Esta es la decisión más simple, donde *p(i) = i* entonces si *h(k) = 142* está ocupada intenta con 143, 144, 145, etc. Si al llegar al final no se encontró ninguna posicion libre, vuelve a empezar desde el principio.

El problema con esto es que agrupa claves en ciertas zonas de la tabla.

Quiadratic probing

Una mejora del linear probing es tomar $p(i) = i^2$. Volviendo al ejemplo de antes si *h(k) = 142* está ocupada intenta con $142 + 1^2 =$ **143**, $142 + 2^2 =$ **146**, $142 + 3^2 =$ **151**, etc. De esta forma las claves quedan mas dispersas.

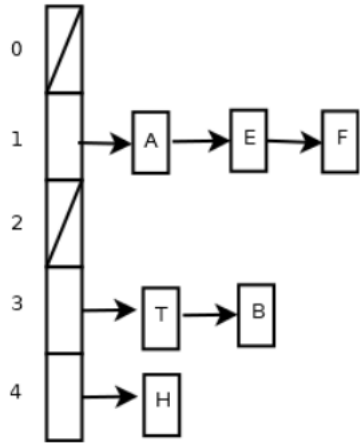
Double hashing

Es la mejor solución para el open addresing, se utilizan dos funciones de hash distintas en donde la secuencia para encontrar una posición vacía es $p = h(k) + i \cdot h2(k)$

Chaining

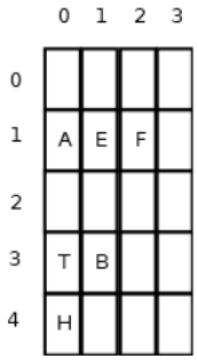
El encadenamiento o chaining contempla que las claves estén en distintas tablas, por lo que cada posicion de la tabla en realidad es un puntero a una lista enlazada con las claves. Por ejemplo:

- A, E y F devuelven el valor 1
- T y B devuelven el valor 3
- H devuelve el valor 4



Bucket addresing

El direccionamiento de cubos o bucket addresing guarda los datos en una misma tabla, pero en cada posicion tiene varias posiciones en vez de una sola. Esto puede hacerse con una matriz, si se mantienen las claves del ejemplo anterior quedaría:



Borrar un elemento

Para eliminar un elemento primero hay que realizar una búsqueda, que va a depender de la forma en la que se resolvieron las colisiones. Si quiero borrar la clave *k* pero en *h(k)* hay otro elemento es que hubo una colision, y *k* ahora está en otro lugar. Para saber en donde está tenemos que tener en cuenta con qué método se resolvió, supongamos que fue con open addresing lineal => hay que revisar $h(k) + i$ con $i = 0, 1, 2, \dots, n$ hasta encontrarlo. Una vez que lo encontremos, supongamos que esta en $h(k) + 2$ no podemos borrarlo y listo, porque si hubiera una clave nueva en $h(k) + 3$ nunca la encontraríamos porque cuando vemos que $h(k) + 2$ está libre dejamos de buscar.

Una posible solución es agregar un vector de tipo booleano que indique falso si nunca fue ocupado o verdadero si lo fue.

Funciones de hash perfectas

Son las que no producen colisiones y logran un tiempo de búsqueda óptimo: O(1). Si además la tabla tiene un tamaño *t* y se almacenan *t* datos se dice que es una función de hash perfecta *mínima* (no hay desperdicio de tiempo de búsqueda ni de memoria).

Claramente si las claves son desconocidas no podemos garantizar que no van a producirse colisiones, pero si son claves conocidas de antenamo se puede plantear una función de dispersión sin colisiones.