Hash-Based Index

Profesor Heider Sanchez

P1. Función hash primo:

Para disminuir la probabilidad de overflow se puede manejar un numero primo (M) de buckets que cumpla lo siguiente:

Función hash: $h(k)=k \bmod M$

En donde: $M \geq \lceil (n/fb)*(1+d) \rceil$, n es el numero de registros, fb es el factor de bloque y d es el factor de corrección que significa el espacio libre en el archivo.

Por ejemplo: si $n=20, fb=4, d=0.1$; entonces $\lceil (20/4)*(1+0.1) \rceil = 6$. Por lo que el siguiente número primo seria $M=7$.

Muestre el estado final del archivo tanto para M=6 y M=7. Verifique si se reduce la cantidad de colisiones.

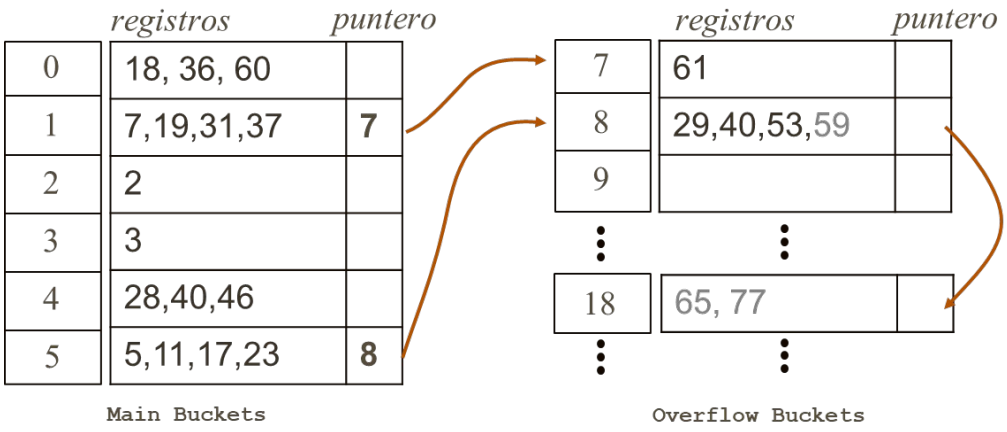
Keys
2,3,5
7,11,17
18,19,23
28,29,31
36,37,40
41,46,53
60,61

0	**18,0,60**
1	**7,1,19,32,37,6**
2	**2**
3	**3**
4	**28,40,46**
5	**5,11,17,23,29, 41,53**

0	**7,28**
1	**29,36**
2	**2,23,37**
3	**3,17,31**
4	**11,18,46,53, 60**
5	**5,19,40,61**
6	**41**

P2. Desbordamiento encadenado.

Existe diferentes alternativas para el manejo de colisiones en disco duro. Una alternativa de manejo es el desbordamiento encadenado, el cual consiste en mantener un puntero al siguiente bucket colisionado, tal como se muestra en la siguiente imagen:



Analice como sería la estructura del índice para implementar el desbordamiento encadenado en disco duro. Luego diseñe los siguientes algoritmos:

1- Algoritmo de inserción de un registro.

```
M = 6;
fb = 4;
int número;
map hash;
void insert(número, fb, M, map hash)
{
	key = número%M;
	if hash[key].length=fb
	{
		hash[key].puntero = BucketLibre;
		insert(número, fb, M, BucketLibre)
	}
	if hash[key].length<fb
	{
		hash[key] add número;
	}


}
```

2- Algoritmo de búsqueda de un registro dada la clave de búsqueda.

```
key = numero%M
bucket buscar(numero, map hash, M, fb, key)
{
	for i in hash
	{
		if key == i.first()
		{
			for x in i[key].registro
			{
				if x == numero
				{return key}
				else if x is never == numero
				{
					key = i[key].pointer
					buscar(numero, map hash, M, fb, key)
				}
			}
		}
	}
```

```
    }
```

P3. Hash Multiples:

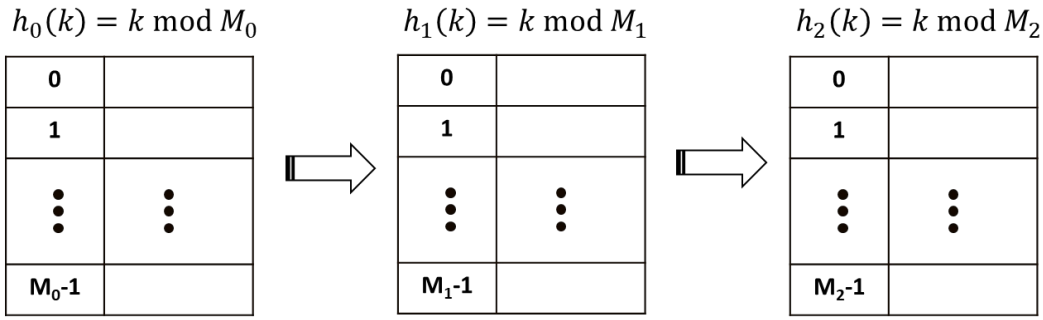
Otra alternativa para desbordamiento es el manejo de múltiples funciones hash, uno para cada nivel de desbordamiento. Por lo que se requiere una secuencia de números de buckets predefinidos.

$M = \{ M_1, M_2, \ldots, M_m \}$

La función hash se evalúa en cada nivel de buckets:

$$h_i(k) = k \bmod M_i$$

La localización de un registro es secuencial por los niveles de desbordamiento:



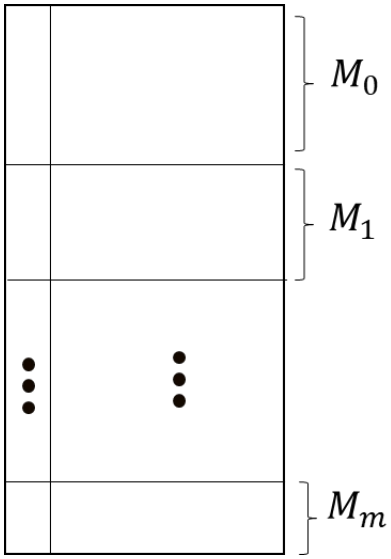
A continuación, el algoritmo de inserción:

```
function insert(obj)
{
    inserted = false;
    i=0;
    bucket = readMainBucket(hash\_i(obj.key));
    while(!inserted)
    {
        if(full(bucket)){
            i=i+1;
            bucket = readOverflowBucket(hash\
_i(key));
        } else {
           appendRecord(bucket, obj);
           inserted = true;
        }
    }
}
```

Se le pide resolver lo siguiente:

1- Proponga el algoritmo de búsqueda.
2- ¿Qué cambios haría usted al algoritmo si se le pide gestionar en un solo archivo tanto los buckets principales como todos los niveles de desbordamiento? ¿Cómo sería la función hash en cada nivel?

$h_i(k)=\ldots$



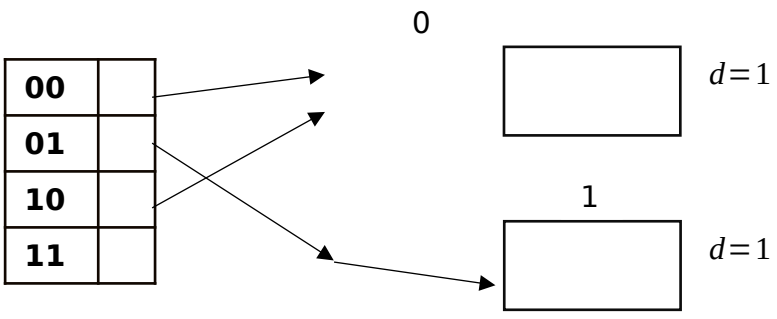
P3. Hash Extensible:

Dado el siguiente conjunto de claves de registros a insertar en un archivo gestionado por un hash extensible, ¿Cómo quedaría organizado los datos al final de todas las inserciones?

Keys
2,3,6
7,11,16
18,20,23
28,29,30

Considere como profundidad global de dos bits $(d=2)$ y un factor de bloque de 4 registros $(fb=4)$. Ilustre paso a paso el proceso de spliting. Sea ordenado y claro en su solución.

Paso 1:



Finalmente, diseñe el algoritmo de inserción en memoria secundaria.