Algoritmos y Estructuras de Datos - Pauta Control 2

Profesor: Patricio Poblete

2 de Junio del 2009

Pregunta 2

(a) La idea de linear probing es resolver las colisiones buscando el casillero libre mas cercano hacia la derecha (de manera circular). Asumimos dada la funcion de hashing h, el arreglo tabla y una variable M que indica el tamaño de la tabla. Tambien asumimos por simplicidad que siempre hay un casillero libre en la tabla de hash (esto se puede verificar facilmente).

```
public void insertar(int x)
{
    int i=h(x); //calculamos la funcion de Hash
    while(tabla[i] != 0) //buscamos el primero espacio vacio
        i=(i+1) % M;
    tabla[i]=x;
}
```

(b) Ahora la idea es, en caso de colision, dejar el elemento entrante en la posicion que le asigna la funcion de hash, y desplazar todo el bloque de colision una casilla la derecha (siempre de manera circular). Asumimos lo mismo que en la parte (a).

```
public void insertar(int x)
{
    int i=h(x);
    int aux=x; //recuerda el elemento que sera sobreescrito
    while(tabla[i] != 0) //vamos desplazando los elementos a la derecha
    {
        int aux2=tabla[i];
        tabla[i]=aux;
        aux=aux2;
        i=(i+1) % M;
    }
    tabla[i]=aux;
}
```

(c) El efecto de una insercion sobre el costo total de busqueda es exactamente el mismo en los dos casos. Si llamamos d al tamaño del bloque de colision (si d es 0 lo interpretamos como que no hay colision), en el caso (a) el costo total de busqueda se ve incrementado solo por el costo asociado al nuevo elemento (ya que el de los demas no varia), el cual sera $d+1$. En el caso (b), el costo se ve incrementado debido al nuevo elemento y a los que pertenecen al bloque de colision. El nuevo elemento aporta al costo una unidad, al igual que cada elemento en el bloque de colision ,por lo tanto el incremento en el costo total de busqueda tambien es $d+1$. Lo anterior nos dice que si partimos con la tabla vacia e insertamos n elementos utilizando en un caso (a) para resolver colisiones y (b) en el otro, tendremos el mismo costo total de busqueda, lo cual implica que tenemos el mismo costo esperado de busqueda exitosa (asumiendo distribucion uniforme sobre las n llaves).

Otra forma de ver que el costo esperado de busqueda exitoso es el mismo, es notar que en los 2 casos formamos un nuevo bloque de colision de tamaño $d+1$, en la misma posicion de la tabla, lo cual nos dice que despues de n inserciones, la forma en que estan distribuidas las llaves en la tabla es exactamente la misma en

en el string 'ab' pues hay un prefijo al inicio idéntico, de forma que el valor de la función de fracaso para el sub-string de largo 4 del patrón vale 2, el tamaño del sufijo.

2.1.2. Función de fracaso de KMP

j	1	2	3	4	5	6	7
f(j)	0	0	1	2	0	1	1

2.1.3. Algoritmo aplicado

a	b	a	b	a	b	b	a	b	a	b	b	a	a	
1	2	3	4	X										
			f(4)=2	3	4	5	6	X						
							f(6)=1	2	3	4	5	6	7	calce!

Para aplicar el algoritmo de búsqueda hay que notar en qué indice falló el calce de carácteres. Luego de esto se alinea el patrón según el índice que indica la función de fracaso y se sigue comparando a partir de éste en la cadena de carácteres. El proceso se repite sucesivamente hasta llegar al final del texto o hasta encontrar un calce.

2.2. Parte (b): BMH

En BMH basta alinear el carácter de la cadena que causa el error con la primera ocurrencia de la letra en el patrón de búsqueda. Si no existe dicho carácter entonces se mueve completamente el patrón.

2.2.1. Algoritmo aplicado

I	N	G	E	N	I	E	R	I	A	_	Y		C	I	E	N	C	I	A	
C	I	E	N	C	I	A														
			_	C	I	E	N	C	I	A										
											C	I	E	N	C	I	A			
													C	I	E	N	C	I	A	calce!

2.3. Parte (c): BMS

Este algoritmo es muy parecido a BMH, salvo que se compara el carácter que falló del patrón con la letra siguiente de la cadena de carácteres.

2.3.1. Algoritmo aplicado

I	N	G	E	N	I	E	R	I	A		Y		C	I	E	N	C	I	A	
C	I	E	N	C	I	A														
								C	I	E	N	C	I	A						
													C	I	E	N	C	I	A	calce!