Hash Tables

Prof. Rhadamés Carmona

Última revisión: 22/04/2019

Agenda

- Definición
- Operaciones
- Cálculo de hash
- Manejo de las colisiones
- Aplicaciones
- Implementaciones
- Ideas finales
- Tarea

Definición

- Una tabla hash es una estructura de datos que mapea claves a valores, utilizando una función de diseprsión o hash que calcula el índice del “slot” o “bucket” donde debería almacenarse o encontrarse el par clave-valor.

Update?
Erase?
Duplicates?

```
vector<vector<Pair<Key, Value>>> table;

void Insert(Key &key, Value &value)
{
    // precond: Search(key)==NULL
    int pos = HashF(key);
    table[pos].append(make_pair<key,value>);
}
```

```
Value *Search(Key &key)
{
    int pos = HashF(key);
    for(int i = 0; I < table[pos].size(); i++)
    {
        if(table[pos][i].first== key)
            return &table[pos][i].second;
    }
    return NULL;
}
```

Definición

- Es un ejemplo claro de time-space trade-off. Si la memoria fuera infinita, o la cantidad de claves fuera “pequeña” y conocida, la función hash podría asignar una posición distinta a cada clave posible de manera que no haya colisión alguna. El slot consistiría en un único elemento, y las operaciones serían O(1).
- Si la función hash retorna siempre cero, entonces el “slot” es un arreglo con todos los elementos de la tabla, y las operaciones serían O(n=m), como en un vulgar y simple arreglo desordenado.

Operaciones

- Insert
- Erase o delete
- Search
- Update
- Utilizando clases, se puede sobrecargar el operador [ ] de manera tal que cuando hagamos table[key], retorna la referencia al Value (existente o recién creado). Así se implementaría fácilmente insert/update.
- Ejemplo, table[“hello”] = x.

Cálculo de hash

- Una búsqueda por dispersión o hashing consiste en:
  - Evaluar función de dispersión
  - Resolver la colisión
- La función de dispersión debe transformar la clave (típicamente un entero o string) en un índice en [0,m-1], donde m es el número de slots. Debe ser fácil de calcular, e idealmente la probabilidad de generar cualquier posición debe ser la misma.

Cálculo de hash

- Debemos crear una hash para cada tipo de key.
- Por ejemplo, para punto flotantes en [a,b) podemos hacer esta conversión:

  $$index(x,m) \equiv int(m*(x-a)/(b-a));$$

- Para enteros de w bits, podemos dividir entre 2^w, y multiplicar por m.
- Ambos arrojan valores uniformes en 0..m-1.

Cálculo de hash

- Sin embargo, los subrangos podrían no ser equi-probables. Por ejemplo, al insertar elementos por su edad, sabemos que hay menos personas con edades dentre 70 y 120 años.
- **Para enteros**, una mejor solución es tomar m como número **primo**, y usar el "**mod**".

  Index(x,m) $\equiv$ x % m;

- Esta dispersión de los datos en m posiciones es equitativa.
- Qué pasaría si tomamos m como un número par? Ejemplo 24.

Cálculo de hash

- Podemos ahora ir de vuelta con esta idea al punto flotante.
- Si la clave está en [a,b], podemos escalar a $[0,2^w)$ y truncar para representarlo como entero de w bits, y entonces usar el “mod”.
- Para poder usar “mod” de esta manera, m tiene que ser primo. Si el programador quiere reservar una tabla de 100 slots, podemos simplemente buscar el primo más cercano (101).

Cálculo de hash

- Una alternativa para enteros (sin que *m* sea primo) es combinar la multiplicación con el mod (α en [0,1)):

  Index(x,m) $\equiv$ int(α*x) % m;

- Cierta combinaciones de m y α pueden generar resultados indeseables.
- Se suele utilizar el "radio dorado" para α

  α=(sqrt(5.0)-1.0)/2.0 $\cong$ 0.618033

Cálculo de hash

- Qué hacer si la clave es un string?
- Podemos ver todo el string como un número, al que le podemos obtener el “mod” m.
- Por ejemplo, “now” puede ser escrito como

  256^{2}*’n’ + 256^{1}*’o’ + 256^{0}*’w’
- Usando la factorización polinómica de Hörner

  (’n’*256+’o’)*256+’w’

Cálculo de hash

- … y así evitar calcular potencias

  (’n’*256+’o’)*256+’w’

(a*b) mod m = (a mod m * b) mod m

(a+b) mod m = (a mod m + b) mod m

```
int hashF(const char *key, int m)
{
        int h = 0, base = 256;
        for (char *p=key; *p !=NULL;  p++)
                h = ( h * base + int(*p) ) % m;
        return h;
}
```

Cálculo de hash

- Aún hay sesgos, porque los strines no son totalmente random. La dispersión **NO** es uniforme.
- Para evitar “anomalías” con tablas de tamaño par o tablas potencias de 2, reemplazamos 256 por 255. Hay soluciones para strings de 7 bits, usando base = 127.

```
int hashF(const char *key, int m)
{
        int h = 0, base = 255;
        for (char *p=key; *p !=NULL;  p++)
                h = ( h * base + int(*p-1) ) % m;
        return h;
}
```

Cálculo de hash

- Sin embargo, hay ciertas anomalías para tamaños de tabla múltiplos de 255 (o de 127 en el otro caso).
- Una “hash universal” le pondría un peso aleatorio a cada término del string. En este caso, la probabilidad de colisión es el deseado 1/m. En hashU se hace un intento al generar una secuencia pseudo random de pesos.

```
int hash U(const char *key, int m)
{
        int h = 0, a = 31415, b=27183;
        for (char *p=key; *p !=NULL;  a = (a*b)%(m-1))
                h = ( h * a + int(*p) ) % m;
        return (h < 0) ? h+m : h;        // por overflow, puede resultar negativo
}
```

Manejo de las colisiones “separate chaining”

- Ya hemos visto qué slot le corresponde a una clave.
- Qué hacer cuando en ese slot ya hay 1 o más elementos? (colisión)
- Podemos definir ese slot como una lista de pares (key, value). Este método es llamado “separate chaining”, porque los items que colisionan se encadenan en una lista separada por slot. Pueden almacenarse en orden o desordenados.

Manejo de las colisiones “separate chaining”

- Buscar un elemento en un slot toma en promedio O(n/m), donde n es el número de elementos de la tabla, y m el número de slots. La inserción es O(1) u O*(1) dependiendo si se usan listas o arreglos en cada slot.
- Si se hacen muchas más búsquedas que inserciones, usar un “vector” por slot con una inserción ordenada O(n/m), con búsquedas O(log(n/m)). Podría usarse un AVL o Rojo-negro para inserción O(log(n/m)).

Manejo de las colisiones "separate chaining"

- Note que n/m puede ser pequeño (por ejemplo n=1000, m=400). En estos casos, el tiempo promedio de búsqueda es constante.
- Se puede tener un estimado de n para seleccionar un valor de m adecuado en términos de memoria y relación n/m.
- Acerca de las claves duplicadas, se puede permitir insertar o no claves duplicadas, dependiendo del caso. Si no se permite duplicados, hay un costo adicional de búsqueda antes de insertar.

Manejo de las colisiones
“Linear probing” (sondeo lineal)

- En este caso se almacenan los pares key-value en una tabla de tamaño m>n, donde las áreas vacías se usan para almacenar colisiones.
- Cuando hay colisión, se almacena la clave en la próxima posición libre de la tabla.
- Pero entonces ocuparía la posición de otra clave.
- Por lo tanto, al buscar una clave, se recorre la tabla circularmente desde la posición “esperada” hasta encontrarla o hasta una posición vacía.

Manejo de las colisiones "Linear probing" (sondeo lineal)

- Para insertar, consideremos que keys y vals son "vector" de M apuntadores (inicialmente NULL cada uno)

```
void insert(Key *key, Value *val) {
        if (N >= M/2) resize(2*M); // duplicar M
        int i;
        for (i = hash(key); keys[i] != NULL; i = (i + 1) % M)
                if (*keys[i]==*key) {
                        vals[i] = val;  return;
                }
        keys[i] = key; vals[i] = val; N++;
}
```

Manejo de las colisiones "Linear probing" (sondeo lineal)

- Para buscar, hacemos el recorrido desde la posición "esperada" hasta encontrarlo, o hasta encontrar una posición vacía.

```
Value * search(Key *key) {
        for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
                if (*keys[i]==*key)
                        return vals[i];  // found!
        return NULL;                     // not found!
}
```

Manejo de las colisiones "Linear probing" (sondeo lineal)

- El problema ahora es cómo eliminar una clave.
- No basta hacer la búsqueda y colocar la celda en NULL o vacía, porque afecta la búsqueda de otras claves.
- Se reinsertan todos los elementos con el nuevo tamaño M.

```
void erase(Key *key) {
        if (!contains(key)) return;
        int i = hash(key);
        while (!(*key == *keys[i]) )
                i = (i + 1) % M;
        keys[i] = null; vals[i] = null; i = (i + 1) % M;
        while (keys[i] != null) {
                //if (hash(keys[i]) != i) { probar…
                Key keyToRedo = keys[i];
                Value valToRedo = vals[i];
                keys[i] = null; vals[i] = null; N--;
                insert(keyToRedo, valToRedo);
                //}
                i = (i + 1) % M;
        }
        N--;
        if (N > 0 && N == M/8) resize(M/2);
}
```

Manejo de las colisiones "Linear probing" (sondeo lineal)

- Resize es llamado únicamente por el constructor, insert & erase (private methods)

```
private void resize(int capacity) {
   Key *k = new Key * [capacity];
   Value *v = new Value * [capacity];
   int oldM = M;
   M = capacity;
   for (int i=0; i< M; i++) {
            v[i] = NULL;
            k[i] = NULL;
   }
   swap(v,vals); swap(k,keys); N=0;
   for (int i = 0; i < oldM; i++)
            if (k[i] != null)
                     insert(k[i], v[i]);
   delete [] k; delete [] v;
}
```

Manejo de las colisiones
Otros métodos

- Aún el sondeo lineal puede generar clustering, i.e. se agrupan los elementos en ciertas áreas de la tabla haciendo que la inserción, búsqueda y eliminación consuman más tiempo es estos clusters.
- La solución es buscar la próxima posición libre en otras regiones de la tabla: (a) sondeo cuadrático, (b) doble hash.

Manejo de las colisiones
Otros métodos

- Sondeo cuadrático: en vez de buscar la posición libre siguiente, se utiliza una fórmula polinómica:

  pos = (hash(key) + c1*i + c2*i^2)%M, donde "i" empieza en 1, y se incrementa en cada iteración; c1 y c2 son constantes.

- Doble hash: se agrega una función de hash (hash2) simple para saltar a la próxima posición:

  pos = (hash(key) + i * hash2(key)) % M

  e.g. hash2(x)=7-(x%7), hash2(x)=x%97+1

  Nota: evitar que hash2 retorne 0, para evitar inf. loop

Aplicaciones

- Almacenamiento de tabla de símbolos de un compilador (nombre de símbolo, dirección).
- Remover elementos duplicados.
- Implementar una matriz esparcida (index, val)
- Diccionario (palabra = key, definición = value).
- DNS (IP, url).
- Phone book (nombre, teléfono) o al revés.

Implementaciones

- Implementaciones: java.util.Hashtable, std::unordered_map, std::unordered_set, std::unordered_multimap, std::unordered_multiset (c++11)
- Multi significa que acepta claves repetidas.

Ideas finales

- Hash tables se utilizan cuando se require acceso a elementos en tiempo constante.
- Se utilizan muy a menudo para relacionar pares key-value.
- Cuando se requiere que los elementos estén ordenados, se recomienda árboles rojo-negro, o en su defecto, AVL. Estos árboles tienen una garantía de performance de peor caso en O(log).

Ideas finales

- Hash tables no es la panacea. Idealmente las operaciones se realizan en tiempo constante. Pero para data sesgada con una función hash deficiente, puede degenerarse (todas las inserciones y búsquedas caen en el mismo slot). Además, el tiempo también depende de la longitud de la clave. No es eficiente para ordenar elementos, como sí lo es el heap, árboles AVL, Rojo-negro, etc.

Tarea

- Implementar “separate chaining”, “linear probing”, “quadratic probing” y “double hash”.
- Hacer pruebas realizando grupos de operaciones de inserción, búsqueda y eliminación, con distintos valores de M.
- Concluir quién tiene mejor rendimiento en distintas situaciones y valores de M.
- Comparar std::unordered_map con std::map en las operaciones de insertar, búsqueda y eliminación para distintos valores de M.