Juan Pablo Montoya Estévez

A01251887

Tarea 5_1

TC 1031.11

Introducción

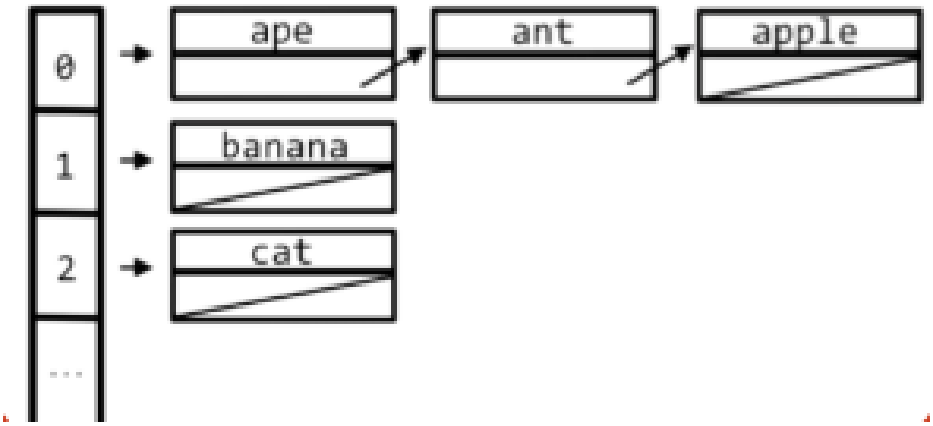
Para manejar estructuras de datos a mayor escala es cuando optamos por manejar las hash tables (o mapa hash, de dispersión) que permiten tener un manejo de la información más estandarizada y es una estructura de datos que permite almacenar llaves y valores, es muy útil para casos donde existen subsecciones y se deban identificar de forma ordenada y rápida, por ejemplo, diccionarios, inventario de almacén, bibliotecas, etc.

Para poder organizar los datos que se le están dando en el main se usan nodos o hash nodes con esta estructura:

```
template<typename K, typename V>
class HashNode
{
    public:
        K key;
        V value;
        HashNode(K, V);
};
```

De este modo se guarda la llave (K) y su valor (V). También utilizan esta función para calcular el índice en el que se almacenará el valor que se va leyendo (index).

Antes de entrar en rigor con cada función distinta que se puede implementar para controlar las colisiones de los nodos se debe analizar qué es lo que está pasando con esta estructura que permite para distintos contextos manejar sus datos de la forma más adecuada posible. Lo que ocurre en el algoritmo es que primero se reciben las llaves y los valores que tomarán los nodos y posteriormente se almacenarán en los buckets de la tabla. Previamente a este paso se define el tamaño del arreglo o cantidad de elementos que se van a guardar, así como se ve abajo se tienen 3 buckets y un tamaño de 5, después se van a recorrer los buckets de la tabla para dejarlos con un valor nulo en ese espacio, de esta forma si se llega a uno vacío se puede identificar eficientemente por el algoritmo y se podrá manejar más rápido el control de colisiones. En este caso se usó una técnica de encadenamiento, pero esta sólo es una forma de implementarlo, y como se mencionó anteriormente, existe una correlación entre los métodos para colisiones y las situaciones para las que se creó el programa.



Métodos (con casos)

Para ahondar más en estos distintos tipos de manejarse estas se tiene el método de cadena o encadenamiento del ejemplo anterior, donde primero se determina el índice al que iría de manera normal, con el cálculo de index que es el residuo de la llave que tenga el nodo entre BUCKET (el hash function es el que nos permite ver este factor de acomodo para el índice).

```
int getIndex(K key);
void remove(K key_);

int hashFunction(K x) {
    return (x % BUCKET);
}
```

```
void Chain<K, V>::insertItem(K key, V value)
{
    int index = hashFunction(key);

    table[index].push_back(new HashNode{key,value});

}
```

(Así se implementa en la función de cadena).

De manera que, si el índice de alguno de los nodos llega a colisionar, lo que sucederá en el acomodo es que, en vez de pasarse al siguiente espacio que esté disponible (como se explicará a continuación más a detalle) se queda en el mismo bucket, pero se le van vinculando al último nodo del bucket como una linked list (memoria dinámica) donde al final se le va conectando

con apuntadores al último nodo el que va llegando, y así sucesivamente hasta que dejen de chocar los índices de los nodos. Consecuentemente se entiende que varias funciones como lo es agregar y borrar nodos de la lista funciona igual que las linked list. Esto representa un grado de dificultad para manejar los datos, pues si se tienen que borrar varios elementos de la tabla, o agregar varios en el mismo bucket y varias veces, la complejidad crecerá exponencialmente conforme se incrementen el número de tamaño, de buckets, de índices repetidos, disminuyendo simultáneamente la capacidad de entregar un resultado más eficiente que otros métodos al usuario del programa.

Aquí está un ejemplo de cómo funcionará en un caso prueba:

```
int a[] = {1, 199, 24, 2, 128};
float v[] = {1.5, 1.6, 1.7, 1.8, 1.32};
```

```
0
1 --> Key1 Value : 1.5
2 --> Key2 Value : 1.8 --> Key128 Value : 1.32
3 --> Key199 Value : 1.6 --> Key24 Value : 1.7
4
5
6
```

Ahora para la técnica lineal o linear en inglés, se observa que simplemente se va a ir recorriendo
en orden los buckets de la tabla hash, aquí se tiene el algoritmo que nos permite observar que una
vez recibido el tamaño de la tabla, se calcula el índice que se representa con el “hv”, resultando
en el residuo de esta división “key/tsize”, con el que se van buscando los índices de los nodos ya
delimitados. Si el espacio de la tabla en el índice que le tocó al nodo actual que se acaba de
calcular su índice, si está desocupado (es nulo) entonces aquí guardamos el valor del nodo (en
conjunto con su llave).

En caso de que ocurra lo contrario a lo que se mencionaba, si no se encuentra desocupado el espacio, y son los mismos índices, lo que hace el programa es tomar los demás índices y recorrerlos hasta encontrar el primero que encuentre desocupado después de este índice, esto no quiere decir que forzosamente debe ser uno posterior, puede ser también que se dé la vuelta completa el algoritmo y esté libre un índice de menor magnitud que el original.

```
template <typename K, typename V>
int Linear<K,V>::getIndex(K key)
{
        int tsize = table.size();

        int hv = key % tsize;

        if (table[hv] ==NULL)
            return hv;
        else
        {

            for (int j = 0; j < tsize; j++)
            {
                int t = (hv + j ) % tsize;
                if (table[t] == NULL)
                {

                    return t;
                }
            }
        }
    return -1;
}
```

Aquí se tiene un caso prueba de este algoritmo, donde claramente se observa que a diferencia del anterior, ninguno está en el mismo bucket (todos con índice diferente).

```
int a[] = {1, 199, 24, 2, 128};
float v[] = {1.5, 1.6, 1.7, 1.8, 1.32};
```

```
KEY : 1 Value : 1.5
KEY : 2 Value : 1.8
KEY : 24        Value : 1.7
KEY : 128       Value : 1.32
KEY : 199       Value : 1.6
```

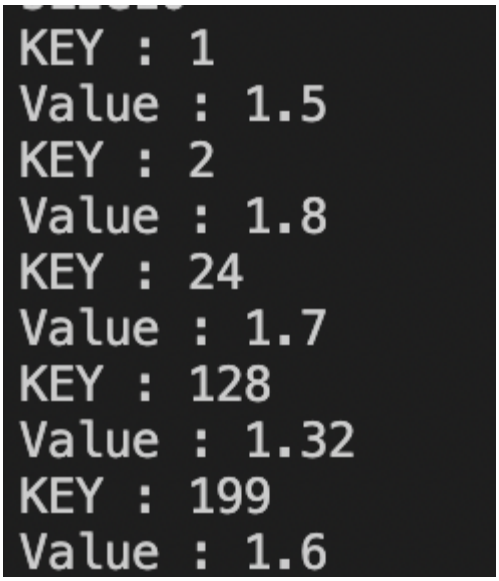
Finalmente para el método de cuadrado retomamos un poco de la forma lineal, porque es casi la misma lógica que se seguirá en este método, lo único que

cambiaría es que se con esta forma de calcular el índice de almacenamiento en la tabla existe menor probabilidad de que se repitan los índices, lo cual es bueno para el rendimiento de memoria y tiempo del programa, así como vemos en el código, se maneja del mismo modo que el lineal, pues busca si el espacio del índice en la tabla es nulo o no, en caso de que no lo sea sencillamente se inserta el nodo actual en este espacio, de lo contrario si no está libre se van recorriendo los índices que sí lo estén, pero aquí se calcula con el resultado de "(hv + j*j) % tsize", con j al cuadrado, de ahí sale el nombre.

```
10  void Quadratic<K,V>::insert(K key, V value )
11  {
12
13      int tsize = (table).size();
14      cout << "size" << tsize <<endl;
15
16      int hv = key % tsize;
17
18
19
20      if (table[hv] ==NULL)
21          (table[hv]) = new HashNode<K,V>(key, value);
22      else
23      {
24
25          for (int j = 0; j < tsize; j++)
26          {
27
28              int t = (hv + j*j ) % tsize;
29              if (table[t] == NULL)
30              {
31
32                  table[t] = new HashNode<K,V>(key, value);
33                  break;
34              }
35          }
36      }
37
38
39  }
```

Para entender mejor este concepto y cómo está funcionando se tiene este caso prueba:

```
int a[] = {1, 199, 24, 2, 128};
float v[] = {1.5, 1.6, 1.7, 1.8, 1.32};
```



Conclusión

Es de vital relevancia tomar la consideración de sopesar y evaluar cuáles son los métodos que van más afín al proyecto o la actividad que se busca realizar antes de llevarse a cabo, pues podría implicar una pérdida tanto de tiempo, esfuerzo y recursos el tener una estructura de datos que sea ineficiente para las necesidades del usuario, y al final, volver a instanciar e implementar una estructura de datos más adecuada. En estos casos prueba se vieron las dificultades que representaría para bases grandes la de chain, pues se ve rápidamente que tomaría más tiempo la inserción, la búsqueda y la eliminación de nodos.