Data and File Structure

Hashing

Best searching method:

Binary search takes (O $\log_2 n$) for a table of n entries. Here, we will see a class of search technique whose search times can be independent of the number of entries in the table.

Symbol Table:

"An important part of any compiler is the construction and maintenance of a dictionary containing names and their associated value. Such a dictionary is called a symbol table." In a typical compiler there may be several symbol tables corresponding to variable names, labels, literals etc.

[Dictionary: Dictionary is a collection of data elements uniquely identified by a field called key. It does support the operation of search, insert and delete. A dictionary supports both sequential and random access. Hash table ideal data structure for dictionaries. They favor efficient storage and retrieval of data lists which are linear in nature.]

Constraints in the design of symbol table:

1. Processing time
2. Memory space

Usually there exists some inverse relationship between the speed of a symbol-table algorithm and the memory space it requires.

Phases associated with construction of symbol tables:

1. Building

   This phase involves the insertion of symbols and their associated values into a table.

2. Referencing

   Referencing is the fetching or accessing of values from a table.

Symbol table accessing method:

1. Linear Search
2. Binary Search (O $\log_2 n$)

   BS is good compare to linear search.

New approach for searching:

The position of a particular entry in the table is determined by the value of the key for that entry. This association is realized through the use of a hashing function.

What is Hashing?

The searching techniques (linear search, binary search) discussed so far depends on the number n of elements in the collection S of data. So, Hashing is being introduced which is essentially independent of the number n.

Tree based data structure: AVL Tree, B tree, tries, red-black trees, splay trees

There exist applications which deal with linear or tabular forms of data, devoid of any superior-subordinate relationship.

Our discussion on hashing is oriented toward "file management". Assume there is a file F of n records with a set K of keys which uniquely determine the records in F. F is maintained in memory by a table T of m memory locations and that L is set of memory locations in T.

We assume keys in K and address in L are decimal integers.

e.g.

A company with 68 employees assigns a 4 digit employee number to each employee which is used as the primary key in the company's employee file. We can, in fact, use the employee number as the address of the record in memory. The search will require no comparisons at all. But unfortunately, this technique will require space for 10000 memory locations, whereas space for fewer than 30 such locations would actually be used. Clearly, this trade-off of space for time is not worth the expense.

The general idea of using the key to determine the address of a record is an excellent idea, but it must be modified so that a great deal of space is not wasted. This modification takes the form of a function H from the set K of keys into the set L of memory address.

Such function,

H: K☐L is called a ***hash function*** or hashing function.

If we want to store the elements, they are stored with respect to some function of the key value. This function is called Hash Function and the search technique we are using is called Hashing.

Collision

Unfortunately, such a function H may not yield distinct values: it is possible those two different keys k_1 and k_2 will yield the same hash addresses, this situation is called "collision", and some method must be used to resolve it.

[*Definition*]: Suppose we want to add a new record R with key k to our file F, but suppose the memory location address H (k) is already occupied. This situation is called "collision".

The act of two or more 'synonyms' vying (compete) for the same position in the hash table is known as "collision". Naturally this entails a modification in the structure of the hash table to accommodate the synonyms.

Hashing is mainly divided into two parts:

1. Hashing Methods
2. Collision resolution

Hashing Methods

1. Division Method
2. Mid-Square Method
3. Folding Method
4. Multiplicative Method
5. Digit Analysis

The two principal criteria used in selecting hash function H: K☐L is as follows:

1. The function H should be very easy and quick to compute.
2. The function H should, as far as possible uniformly distribute the hash addressed throughout the set L so that there are a minimum number of collisions.

Hashing Methods/Functions:

1. **Division Method [Modular Arithmetic]**

Choose a number m or table_size larger like the number n of keys. The number m is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions.

Hash function is defined by,

Hash (key) =key % table_size or

Hash (key) =key % table_size + 1

The second formula helps us to start the hash function from 1 rather than to start it from 0.

Hash (2763) = 2763 % 97 +1 = 48, (where m=97 that is prime number close to 99)

2. **Mid-Square Method**

The key is squared. We define the Hash Function in this case as:

Hash (Key) = P

Where, P is obtained by deleting digits from both end of $key^{2.}$ We emphasize that same position of key^2 must be used for all the keys.

The following calculations are performed:

Key	2763	6589	1825
key^2	7634169	43414921	3330625
Hash (key)	34	14	30

Observe that fourth and fifth digit from the right side are chosen for the hash address.

3. **Folding Method**

We partition the key k into a number of parts k_1, k_2,... kr, where each part, except possibly the last, has the same number of digits as the required address. We then add the parts together and ignore the last carry.

That is

Hash (k) = $k_1 + k_2 + ….. + k_r$

Where leading digit if carries, if any, are ignored. Sometimes, for extra “milling”, the even numbered parts, k_2, k_4 …. Are each reserved before the addition. Folding assures better spread of keys across the hash table.

4. **Multiplicative Method**
5. **Digit Analysis**

Collision Resolution Technique

Load Factor: $\lambda = n/m$

Where n is number of keys in K (which is the number of records in F) and m is number of hash addresses in L.

S (λ) =average number of probes for a successful search

U (λ) =average number of probes for a unsuccessful search

1. *Linear Probing*

   Suppose that a new record R with key k is to be added to the memory table T, but that the memory location with hash address H(k)=h is already filled. One natural way to resolve the collision is to assign R to the first available location following T(h). (We assume that the table T with m locations is circular, so that T[1] comes after T[m].) Accordingly, with such a collision procedure, we will search for the record R in the table T by linearly searching the location T[h], T[h+1]... until finding R or meeting empty location, which indicates unsuccessful search.

   The above collision resolution technique is called "Linear Probing".

   $S(\lambda)=\frac{1}{2}(1+(1/(1-\lambda)))$
   $U(\lambda)= \frac{1}{2}(1+(1/(1-\lambda)^2))$

   One main disadvantage of linear probing is that records tend to cluster, that is, appear next to one another, when the load factor is greater than 50 percent. Such a clustering substantially increases the average search time for a record.

   Two techniques to minimize clustering:

   a. Quadratic Probing

      Suppose a record R with key K has the hash address H(k)=h. Then, instead of searching the locations with addresses h, h+1, h+2 ...., we linearly search the locations with addresses h, h+1, h+4,h+9,h+16...h + square(i)

      If the number m of locations in the table T is a prime number, then the above sequence will access half of the locations in T.

   b. Double Hashing

      Suppose a record R with key K has the hash addresses H (k) =h and H' (k) = h'! = m. Then we linearly search the locations with addresses

      h, h+h', h+2h', h+3h',

      If m is a prime number, then the above sequence will access all the locations in the table T.

2. *Chaining*

   Chaining involves maintaining tables in memory. First of all, as before, there is a table T in memory which contains the records in F, except that T now has an additional field LINK which is used so that all records in T with the same hash address h may be linked together to form a linked list. Second, there is a hash address table LIST which contains pointers to the lined list in T.

   Suppose a new record R with key k is added to the file F. We place R in the first available location in the table T and then add R to the linked list with pointer LIST [H(k)]. If the linked lists of records are not sorted, than R is simply inserted at the

beginning of its linked list. Searching for a record or deleting a record is nothing more than searching for a node or deleting a node from a new linked list, as discussed in.

$S(\lambda)=1+½(\lambda)$
$U(\lambda)= e^{-\lambda} +\lambda$

Here the load factor $\lambda=n/m$ may be greater than 1, since the number m of hash addresses in L (not the number of locations in T) may be less than the number n of records in F.

The main disadvantage to chaining is that one needs 3m memory cells for the data. Specifically, there are m cells for the information field INFO, there are m cells for the link field LINK, and there are m cells for the pointer array LIST. Suppose each record requires only 1 word for its information field. Then it may be more useful to use open addressing with a table with 3m location, which has the load factor <= 1/3, than to use chaining to resolve collisions.

In other words every bucket is maintained as a singly linked list with synonyms represented as nodes. The buckets continue to be represented as a sequential data structure as before and to favour the hash function computation. Such a method of handling overflow is called chaining or open hashing or separate chaining.

Fig. Chained hash table

The link field of the last synonym in each chain is a null pointer. Those buckets which are yet to accommodate keys are also marked null.

Operations on chained hash tables:

a. Search
b. Delete

Performance Analysis:

Complexity of the chained hash table is dependent on the length of the chain of nodes corresponding to the buckets. The best case is O (1) and worst case (when all the n elements map to the same bucket and the length of the chain corresponding to that bucket is n, with the searched key turning out to be the last in the chain.) is O (n)

3. *Hashing with Buckets [Linear Open Addressing]*

Let us suppose a group of keys are to be inserted into a hash table HT of size L, making use of the modulo arithmetic function H(k)=k mod L. Since the range of hash table index is limited to lie between 0 and L-1, for a population of N(N>L) keys collisions are bound to occur. Hence a provision needs to be made in the hash table to accommodate the data elements that are synonyms.

We choose to adopt a sequential data structure to accommodate the hash table. Let HT[0:L-1] be the hash table. Here the L locations of the hash table are termed as buckets. Every bucket provides accommodation for the data elements. However to accommodate synonyms keys which map to the same bucket, it is essential that a provision be made in the buckets. We therefore partition buckets into what are called slots to accommodate synonyms. Thus if a bucket b has s slots, then s synonyms can be accommodate in the bucket b. In the array implementation hash table is represented as HT[0:L-1,0:s-1]. The choice of number of slots in a bucket needs to be decided based on the application.

The bucket to which the key is mapped by the hash function is known as the home bucket. To handle overflow linear probing or Linear open addressing or closed hashing can be used.

Operations on linear open addressed hash tables:

a. Search
b. Delete

Performance Analysis:

Smaller the loading factor better is the average case performance of the hash table in comparison to that of linear lists.

4. **Rehashing:**
5. **Quadratic Probing**
6. **Random probing**

The above 4, 5 and 6 are the other collision resolution techniques with open addressing.

Applications

Application of hash tables in the fields of compiler design, relational database query processing and file organization.

a. Representation of a keyword table in compiler:

   "Hash Table turn out to be one of the best propositions for the representation of symbol tables"

b. Hash tables in the evaluation of a join operation on relational databases
c. Hash tables in a direct file organization

Reference:

DATA STRUCTURES AND ALGORITHMS Concepts, Techniques and Applications by G A V PAI