Lecture 17
Introduction to Hashing

Why Hashing?
Internet has grown to millions of users generating terabytes of content every day. According to internet data tracking services, the amount of content on the internet doubles every six months. With this kind of growth, it is impossible to find anything in the internet, unless we develop new data structures and algorithms for storing and accessing data. So what is wrong with traditional data structures like Arrays and Linked Lists? Suppose we have a very large data set stored in an array. The amount of time required to look up an element in the array is either O(log n) or O(n) based on whether the array is sorted or not. If the array is sorted then a technique such as *binary search* can be used to search the array. Otherwise, the array must be searched *linearly*. Either case may not be desirable if we need to process a very large data set. Therefore we discuss a new technique called **hashing** that allows us to **update and retrieve** any entry in constant time **O(1).** The constant time or O(1) performance means, the amount of time to perform the operation does not depend on data size n.

The Map Data Structure
In a mathematical sense, a map is a relation between two sets. We can define Map M as a set of pairs, where each pair is of the form (key, value), where for given a key, we can find a value using some kind of a "function" that maps keys to values. The key for a given object can be calculated using a function called a **hash function**. In its simplest form, we can think of an array as a Map where key is the index and value is the value at that index. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i]. The idea of a hash table is more generalized and can be described as follows.

The concept of a **hash table** is a generalized idea of an array where key does not have to be an integer. We can have a name as a key, or for that matter any object as the key. The trick is to find a **hash function** to compute an index so that an object can be stored at a specific location in a table such that it can easily be found.

Example:
Suppose we have a set of strings {"abc", "def", "ghi"} that we'd like to store in a table. Our objective here is to find or update them quickly from a table, actually in O(1). We are not concerned about ordering them or maintaining any order at all. Let us think of a simple schema to do this. Suppose we assign "a" = 1, "b"=2, … etc to all alphabetical characters. We can then simply compute a number for each of the strings by using the sum of the characters as follows.

"abc" = 1 + 2 + 3=6, "def" = 4 + 5 + 6=15 , "ghi" = 7 + 8 + 9=24

If we assume that we have a table of size 5 to store these strings, we can compute the location of the string by taking the **sum mod 5**. So we will then store

"abc" in 6 mod 5 = 1, "def" in 15 mod 5 = 0, and "ghi" in 24 mod 5 = 4 in locations 1, 0 and 4 as follows.

0	1	2	3	4
def	abc			ghi

Now the idea is that if we are given a string, we can immediately compute the location using a simple hash function, which is **sum of the characters mod Table size**. Using this hash value, we can search for the string. This seems to be great way to store a Dictionary. Therefore the idea of hashing seems to be a great way to store pairs of (key, value) in a table.

Problem with Hashing
The method discussed above seems too good to be true as we begin to think more about the hash function. First of all, the hash function we used, that is the sum of the letters, is a **bad one**. In case we have permutations of the same letters, "abc", "bac" etc in the set, we will end up with the same value for the sum and hence the key. In this case, the strings would hash into the same location, creating what we call a **"collision".** This is obviously not a good thing. Secondly, we need to find a good table size, preferably a prime number so that even if the sums are different, then collisions can be avoided, when we take mod of the sum to find the location. So we ask two questions.

Question 1: How do we pick a good hash function?
Question 2: How do we deal with collisions?

The problem of storing and retrieving data in O(1) time comes down to answering the above questions. Picking a "good" hash function is key to successfully implementing a hash table. What we mean by "good" is that the function must be *easy to compute* and *avoid collisions* as much as possible. If the function is hard to compute, then we lose the advantage gained for lookups in O(1). Even if we pick a very good hash function, we still will have to deal with "some" collisions.

Finding a "good" hash Function
It is difficult to find a "perfect" hash function, that is a function that has no collisions. But we can do "better" by using hash functions as follows. Suppose we need to store a dictionary in a hash table. A dictionary is a set of Strings and we can define a hash function as follows. Assume that S is a string of length n and
$S = S_1 S_2 \ldots S_n$

$$\mathbf{H(S) = H(\text{“}S_1 S_2 \ldots S_n\text{”}) = S_1 + p\,S_2 + p^2\,S_3 + \ldots + p^{n-1}\,S_n}$$

where p is a prime number. Obviously, each string will lead to a unique number, but when we take the number Mod TableSize, it is still possible that we may have collisions but may be fewer collisions than when using a naïve hash function like the sum of the characters.

Although the above function minimizes the collisions, we still have to deal with the fact that function must be **easy to compute**. Rather than directly computing the above functions, we can reduce the number of computations by rearranging the terms as follows.

$H(S) = S_1 + p (S_2 + p(S_3 + \ldots. P (S_{n-1} + p S_n))))$

This rearrangement of terms allows us to compute a good hash value quickly.

Implementation of a Simple Hash Table

A hash table is stored in an array that can be used to store data of any type. In this case, we will define a generic table that can store nodes of any type. That is, an array of void*'s can be defined as follows.

```
void*  A[n];
```

The array needs to be initialized using

```
for (i  = 0; i < n ; i++)
     A[i] = NULL;
```

Suppose we like to store strings in this table and be able to find them quickly. In order to find out where to store the strings, we need to find a value using a hash function. One possible hash function is

Given a string $S = S_1S_2\ldots. S_n$
Define a hash function as
$H(S) = H(\text{“}S_1S_2\ldots. S_n\text{”}) = S_1 + p S_2 + p^2 S_3 + \ldots. + p^{n-1} S_n$ ----------(1)

where each character is multiplied by a power of p, a prime number.

The above equation can be factored to make the computation more effective (see exercise 2). Using the factored form, we can define a function hashcode that computes the hash value for a string s as follows.

```
int hashcode(char* s){
   int sum = s[strlen(s)-1], p = 101;
   int i;
   for (i=1;i<strlen(s);i++)
       sum =  s[strlen(s)-i-1] + p*sum;
    return sum;
}
```

This allows any string to be placed in the table as follows. We assume a table of size 101.

```
A[hashcode(s)%101] = s;    // we assume that memory for s is already being allocated.
```

One problem with above method is that if any collisions occur, that is two strings with the same hashcode, then we will lose one of the strings. Therefore we need to find a way to handle collisions in the table.

Collisions

One problem with hashing is that it is possible that two strings can hash into the same location. This is called a **collision**. We can deal with collisions using many strategies, such as linear probing (looking for the next available location i+1, i+2, etc. from the hashed value i), quadratic probing (same as linear probing, except we look for available positions i+1 , i + 4, i + 9, etc from the hashed value i and separate chaining, the process of creating a linked list of values if they hashed into the same location. We will discuss hashing and collisions in detail in the next lesson.

EXERCISES

1. Indicate whether you use an Array, Linked List or Hash Table to store data in each of the following cases. Justify your answer.
   a. A list of employee records need to be stored in a manner that is easy to find max or min in the list
   b. A data set contains many records with duplicate keys. Only thing needed is to keep the list in sorted order.
   c. A library needs to maintain books by their ISBN number. Only thing important is finding them as soon as possible.
   d. A data set needs to be maintained in order to find the median of the set quickly

2. Given the hash function as

H(S) = H("$S_1S_2\ldots S_n$") = $S_1 + p\,S_2 + p^2\,S_3 + \ldots + p^{n-1}\,S_n$

Find the total number of multiplications and additions to compute hash code of the string "gunawardena" using standard formula (as given above) and as a factored form of the same formula as given by

H(S) = H("$S_1S_2\ldots S_n$") = $S_1 + p\,[\,S_2 + p[\,pS_3 + \ldots p\,[S_{n-1} + p\,S_n]]]$

3. Given a hash table defined as **void* A[n],** Complete the function **insert(void*** Aptr, char* word);**
   that inserts word to the hashtable using the hash function defined in (1). You can assume the hashcode function is given. Also assume that there are no collisions (we will deal with collisions in the next lesson)

4. How would you expand the definition of hash table (given in problem 3) to include to create a linked lists of nodes of all nodes that collided to the same hash code ?

5. When a hash table fills up (say more than 70% of the capacity) a technique is to double the size of the table and rehash all elements from the old table. Using the hash table defined in problem 3, complete the following function

   int resize(void*** Aptr, int newsize)

6. Consider that hash table is defined as an array of void*'s. That is void* hashTable[n]. Write a function, findMax(void* A[], int n, int (fp)(void*,void*)) that will take the array A, its size n, and a function pointer fp and finds the maximum element in the array.

ANSWERS

1. Indicate whether you use an Array, Linked List or Hash Table to store data in each of the following cases. Justify your answer.
   a. A list of employee records need to be stored in a manner that is easy to find max or min in the list

[Hash tables are not good for finding ordered data. Therefore an array is the best data structure to use based on the need]

   b. A data set contains many records with duplicate keys. Only thing needed is to keep the list in sorted order.

[Since there are many duplicate keys, it would be challenging to find a good hash function. Therefore an array or linked list is the best data structure to use based on the need]

   c. A library needs to maintain books by their ISBN number. Only thing important is finding them as soon as possible.

[The key here is that ISBN numbers are distinct, so therefore less likely to cause any collisions and the important thing is finding things fast. So hash table is ideal here]

   d. A data set needs to be maintained in order to find the median of the set quickly

[hash tables are not good data structures for finding ordered data. So we would use a sorted array to store the data and find the median immediately using the middle element in the array]

2. Given the hash function as

$H(S) = H(\text{“}S_1S_2\ldots S_n\text{”}) = S_1 + p\,S_2 + p^2\,S_3 + \ldots + p^{n-1}\,S_n$

Find the total number of multiplications and additions to compute hash code of the string “gunawardena” using standard formula (as given above) and as a factored form of the same formula as given by

$H(S) = H(\text{“}S_1S_2\ldots S_n\text{”}) = S_1 + p\,[\,S_2 + p[\,pS_3 + \ldots p\,[S_{n-1} + p\,S_n]]]]$

3. Given a hash table defined as **void* A[n],** Complete the function **insert(void*** Aptr, char* word);** that inserts word to the hashtable using the hash function defined in (1). You can assume the hashcode function is given. Also assume that there are no collisions (we will deal with collisions in the next lesson)

```
int insert(void*** Aptr, char* word) {
      if (*Aptr == NULL) /* assign memory*/
         *Aptr = malloc(LENGTH*sizeof(node*)); /*assume LENGTH defined*/
```

```
    (*Aptr)[ hashcode(word)%TABLESIZE] = word;
}
```

4. How would you expand the definition of hash table (given in problem 3) to include to create a linked lists of nodes of all nodes that collided to the same hash code?

The idea here would be to define a node type as follows.

```
typedef struct node {
    char* word;
    struct node* next;
} node;
```

Then each word that hashed into the same code can be inserted to the front of that list. Here is a modified version of the insert function.

```
int insert(void*** Aptr, char* word) {
     if (*Aptr == NULL) /* assign memory*/
        { *Aptr = malloc(LENGTH*sizeof(node*)); /*assume LENGTH defined*/
          int i=0;
          for (i=0;i<LENGTH;i++)  (*Aptr)[i] = NULL;
        }
     node* ptr = malloc(sizeof(node));
     ptr->word = malloc(strlen(word)+1);
    strcpy(ptr->word, word);
     ptr -> next = (*Aptr)[ hashcode(word)%TABLESIZE];
    (*Aptr)[ hashcode(word)%TABLESIZE]=ptr;
}
```

5. When a hash table fills up (say more than 70% of the capacity) a technique is to double the size of the table and rehash all elements from the old table. Using the hash table defined in problem 3, complete the following function

```
int resize(void*** Aptr, int newsize) {
     void** temp = malloc(sizeof(void*)*newsize);
    int i;
    for (i=0;i<newsize/2;i++)
        temp[i] = (*Aptr)[i];
    *Aptr = temp;
  return 0;
}
```

6. Consider that hash table is defined as an array of void*'s. That is void* hashTable[n]. Write a function, findMax(void* A[], int n, int (*fp)(void*,void*)) that will take the array A, its size n, and a function pointer fp and finds the maximum element in the array.

```
int findMax(void* A[], int n, int (*fp)(void*,void*)) {
      int i = 0;
      void* max=null;
      for (i=0; i<n; i++){
        if (A[i] != null && fp(A[i],max)>0)
              max = A[i];
      }
}
```