Chapter 4 - Hashing

Symbol Table

- Symbol table is used widely in many applications. – dictionary is a kind of symbol table
- In general, the following operations are performed on a symbol table
  1. determine if a particular name is in the table
  2. retrieve the attribute of that name
  3. modify the attributes of that name
  4. insert a new name and its attributes
  5. delete a name and its attributes
- A technique called hashing can provide a good performance for search, insert, and delete.
- Instead of using comparisons to perform search, hashing relies on a formula called the hash function.
- Hashing can be divided into static hashing and dynamic hashing

Static Hashing

- Identifiers are stored in a fixed-size table called hash table.
- The address of location of an identifier, x, is obtained by computing some arithmetic function h(x).
- The memory available to maintain the symbol table (hash table) is assumed to be sequential.
- The hash table consists of b buckets and each bucket contains s records.
- h(x) maps the set of possible identifiers onto the integers 0 through b-1.
- The **identifier density** of a hash table is the ratio n/T, where n is the number of identifiers in the table and T is the total number of possible identifiers. The **loading density** or **loading factor** of a hash table is $\alpha = n/(sb)$.
- Two identifiers, I1, and I2, are said to be **synonyms** with respect to h if h(I1) = h(I2).
- An **overflow** occurs when a new identifier i is mapped or hashed by h into a full bucket.
- A **collision** occurs when two non-identical identifiers are hashed into the same bucket.
- If the bucket size is 1, collisions and overflows occur at the same time.

Example 1: Consider the hash table ht with b = 26 buckets and s = 2. We have n = 10 distinct identifiers, each representing a C library function. This table has a loading factor, a, of 10/52 = 0.19. The hash function must map each of the possible identifiers onto one of the number, 0-25. We can construct a fairly simple hash function by associating the letter, a-z, with the number, 0-25, respectively, and then defining the hash function, f (x), as the first character of x. Using this scheme, the library functions acos, define, float, exp, char, atan, ceil, floor, clock, and ctime hash into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively.

synonyms:
char, ceil,
clock, ctime

overflow

	Slot 0	Slot 1	
0	acos	atan	synonyms
1			
2	char	ceil	
3	define		
4	exp		
5	float	floor	synonyms
6			
…			
25			

b=26, s=2, n=10, α=10/52=0.19, f(x)=the first char of x

x: acos, define, float, exp, char, atan, ceil, floor, clock, ctime

f(x):0, 3, 5, 4, 2, 0, 2, 5, 2, 2

Hash Function

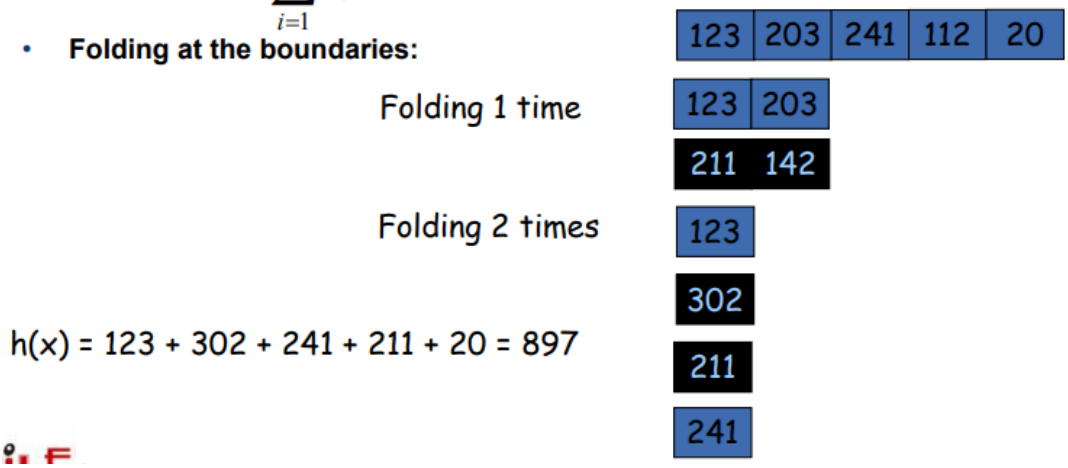
- A hash function, h, transforms an identifier, x, into a bucket address in the hash table.
- Ideally, the hashing function should be both easy to compute and results in very few collisions.
- A uniform hash function supports that a random identifier x has an equal chance of hashing into any of the b buckets
- There are several types of uniform hash functions, and we shall describe four of them. We assume that the identifiers have been suitably transformed into a numerical equivalent.
- **Mid-Square**
  - Mid-Square function, hm, is computed by squaring the identifier and then using an appropriate number of bits from the middle of the square to obtain the bucket address.
  - Since the middle bits of the square usually depend on all the characters in the identifier, different identifiers are expected to result in different hash addresses with high probability.
- **Division**
  - Another simple hash function is using the modulo (%) operator.
  - An identifier x is divided by some number M and the remainder is used as the hash address for x.
  - The bucket addresses are in the range of 0 through M-1.
  - If M is a power of 2, then $h_D(x)$ depends only on the least significant bits of x.
  - If a division function h_D is used as the hash function, the table size should not be a power of two.
  - If M is divisible by two, the odd keys are mapped to odd buckets and even keys are mapped to even buckets. Thus, the hash table is biased.
  - To avoid the above problem, M needs to be a prime number. Then, the only factors of M are M and 1.

- **Folding**
  - The identifier x is partitioned into several parts, all but the last being of the same length.
  - All partitions are added together to obtain the hash address for x.
    - Shift folding: different partitions are added together to get h(x).
    - Folding at the boundaries: identifier is folded at the partition boundaries, and digits falling into the same position are added together to obtain h(x). This is similar to reversing every other partition and then adding.

- **x=12320324111220 are partitioned into three decimal digits long. P1 = 123, P2 = 203, P3 = 241, P4 = 112, P5 = 20.**
- **Shift folding:**

$$h(x) = \sum_{i=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

- **Folding at the boundaries:**



- Digit Analysis
  - This method is useful when a static file where all the identifiers in the table are known in advance.
  - Each identifier x is interpreted as a number using some radix r.
  - The digits of each identifier are examined.
  - Digits having most skewed distributions are deleted. Enough digits are deleted so that the number of remaining digits is small enough to give an address in the range of the hash table.

Overflow Handling

Because the size of the identifier space, T, is usually several orders of magnitude larger than the number of buckets, b, and s is small, overflows necessarily occur. Hence, a mechanism to handle overflow is needed. There are two ways to handle overflow: – Linear Open addressing/Open Addressing – Chaining

Linear Open Addressing / Open Addressing / Closed Hashing

It is a method of collision resolution in hash tables. With this method a hash collision is resolved by probing, or searching through alternate locations in the array (the *probe sequence*) until an unused array slot is found. Well known open addressing methods include:

Linear probing

In linear probing, following hash function is used to resolve the collision:

h(k , i) = [h'(k) + i] mod n, where n is the size of hash table, ***h'(k) = k mod n*** , and i is probe number, varies from 0 to n-1.

Therefore, for a given key, first the location generated by ***h'(k) mod n*** is probed; because first time i = 0. If the location is free, the value is stored in it, else the second probe generates the address of the location given by ***[h'(k) + 1] mod n***. Similarly, if the location is occupied, the subsequent probes generate the address as ***[h'(k) + 2] mod n***, ***[h'(k) + 3] mod n***, and so on, until a free location is found.

In case of searching a value, the value stored at the location generated by hash function is compared with the value to be searched. If the two values match, then the search operation is successful. Otherwise, if the values do not match, then sequential search is performed until the value is found or an empty slot is reached. When an empty slot is reached then the search is unsuccessful.

Example:

Consider a hash table, (call ht) of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, and 92 into the table.

Let ***h'(k) = k mod n, n = 10***
Initially the hash table can be given as:

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 1:

Key = 72
h(72, 0) = (72 mod 10 + 0) mod 10
= (2) mod 10
= 2
Since ht[2] is vacant, insert key 72 at this location.

-1	-1	72	-1	-1	-1	-1	-1	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 2:

Key = 27
h(27, 0) = (27 mod 10 + 0) mod 10
= (7) mod 10
= 7

Since ht[7] is vacant, insert key 27 at this location.

-1	-1	72	-1	-1	-1	-1	27	-1	-1
0	1	2	3	4	5	6	7	8	9

Step 3:

Key = 36
h(36, 0) = (36 mod 10 + 0) mod 10
= (6) mod 10
= 6
Since ht[6] is vacant, insert key 36 at this location.

-1	-1	72	-1	-1	-1	36	27	-1	-1
0	1	2	3	4	5	6	7	8	9

Step 4:

Key = 24
h(24, 0) = (24 mod 10 + 0) mod 10
= (4) mod 10
= 4
Since ht[4] is vacant, insert key 24 at this location.

-1	-1	72	-1	24	-1	36	27	-1	-1
0	1	2	3	4	5	6	7	8	9

Step 5:

Key = 63
h(63, 0) = (63 mod 10 + 0) mod 10
= (3) mod 10
= 3
Since ht[3] is vacant, insert key 63 at this location.

-1	-1	72	63	24	-1	36	27	-1	-1
0	1	2	3	4	5	6	7	8	9

Step 6:

Key = 81
h(81, 0) = (81 mod 10 + 0) mod 10
= (1) mod 10
= 1
Since ht[1] is vacant, insert key 81 at this location.

-1	81	72	63	24	-1	36	27	-1	-1
0	1	2	3	4	5	6	7	8	9

Step 7:

Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10

= (2) mod 10

= 2

Now, ht[2] is occupied, so we cannot store 92 in ht[2]. Therefore, try again for the next location. Thus probe, i = 1, this time,

Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10

= (3) mod 10

= 3

ht[3] is occupied, so we cannot store 92 in ht[3]. Therefore, try again for the next location. Thus probe, i = 2, this time,

Key = 92

h(92, 2) = (92 mod 10 + 2) mod 10

= (4) mod 10

= 4

ht[4] is also occupied, so we cannot store 92 in ht[4]. Therefore, try again for the next location. Thus probe, i = 3,

Key = 92

h(92, 3) = (92 mod 10 + 3) mod 10

= (5) mod 10

= 5

Since ht[5] is vacant, insert 92 at this location.

-1	81	72	63	24	92	36	27	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Quadratic probing

In this technique, following hash function is used to resolve the collision:

h(k , i) = [h'(k) + c_1 i + c_2 i^2] mod n , , where n is the size of hash table, ***h'(k) = k mod n*** , and i is probe number, varies from 0 to n-1, and c_1 and c_2 are constants such that c_1 and $c_2 \neq 0$.

Instead of doing a linear search, quadratic probing does a quadratic search.

Example:

Consider a hash table, (call ht) of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$.

Let ***h'(k) = k mod n, n = 10***
Initially the hash table can be given as:

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

we have ***h(k , i) = [h'(k) + c_1 i + c_2 i^2] mod n***

Step 1:

Key = 72
h(72, 0) = (72 mod 10 + 1 X 0 + 3 X 0) mod 10
= (2) mod 10
= 2
Since ht[2] is vacant, insert key 72 at this location. The hash table now becomes:

-1	-1	72	-1	-1	-1	-1	-1	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 2:

Key = 27
h(27, 0) = (27 mod 10 + 1 X 0 + 3 X 0) mod 10
= (7) mod 10
= 7
Since ht[7] is vacant, insert key 27 at this location. The hash table now becomes:

-1	-1	72	-1	-1	-1	-1	27	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 3:

Key = 36
h(36, 0) = (36 mod 10 + 1 X 0 + 3 X 0) mod 10
= (6) mod 10
= 6
Since ht[6] is vacant, insert key 36 at this location. The hash table now becomes:

-1	-1	72	-1	-1	-1	36	27	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 4:

Key = 24

h(24, 0) = (24 mod 10 + 1 X 0 + 3 X 0) mod 10

= (4) mod 10

= 4

Since ht[4] is vacant, insert key 24 at this location. The hash table now becomes:

-1	-1	72	-1	24	-1	36	27	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 5:

Key = 63

h(63, 0) = (63 mod 10 + 1 X 0 + 3 X 0) mod 10

= (3) mod 10

= 3

Since ht[3] is vacant, insert key 63 at this location. The hash table now becomes:

-1	-1	72	63	24	-1	36	27	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 6:

Key = 81

h(81, 0) = (81 mod 10 + 1 X 0 + 3 X 0) mod 10

= (1) mod 10

= 1

Since ht[1] is vacant, insert key 81 at this location. The hash table now becomes:

-1	81	72	63	24	-1	36	27	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Step 7:

Key = 101

h(101, 0) = (101 mod 10 + 1 X 0 + 3 X 0) mod 10

= (1) mod 10

= 1

Since ht[1] is already occupied, the key 101 cannot be stored in ht[1]. Therefore, try again for next location. Thus probe i = 1, this time.

h(101, 0) = (101 mod 10 + 1 X 1 + 3 X 1) mod 10

= (5) mod 10

= 5

Since ht[5] is vacant, insert key 101 at this location. The hash table now becomes:

-1	81	72	63	24	101	36	27	-1	-1
0	1	2	3	4	5	6	7	8	9

Double Hashing

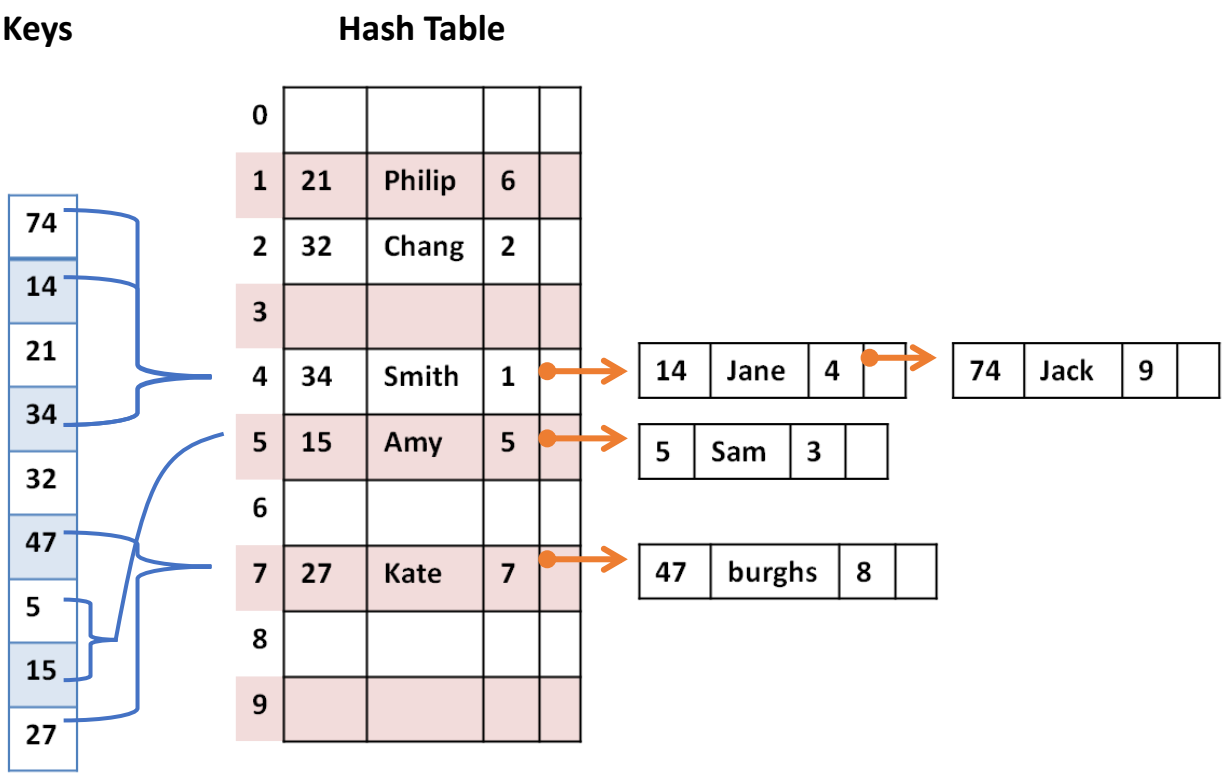
Double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing. Double hashing uses two hash functions rather than a single function. The hash function in case of double hashing can be given as:

h(k , i) = [h₁ (k) + i h₂ (k)] mod n, where n is the size of hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as ***h₁(k) = k mod n, h₂(k) = k mod n′ ,*** i is probe number that varies from 0 to n-1, and ***n′*** is chosen to be less than n. We can choose ***n′*** = n-1 or n-2.

When we have to insert a key k into the hash table, we first probe the location given by applying ***h₁(k) mod n*** because during the first probe, i = 0. If the location is vacant, the key is inserted into it, else subsequent probes generate locations that are at an offset of ***h₂(k) mod n′*** from the previous location. Since the offset may vary with every probe depending on the value generated by the second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.

Chaining

Open hashing, or separate chaining, is to keep a list of all elements that hash to the same value. For example consider student records with Regd.No being key, Name and Rank being values are to be inserted into the hash table using separate chaining. Let the size of hash table be 10. Hash function used is h(key) = key mod 10. Student records with regd.nos 74, 14, and 34 will hash into same cell with index 4. Separate chaining maintains linked list for each hash location. After the insertion of 9 elements,74, 14, 21,---, 15, 27 the resulting hash table is as shown in the following figure.



In order to search for an element same hash function is to be applied on the key to determine the index of the list to traverse. The list should be traversed to locate the item.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct hash *ht = NULL;
int hashTableSize = 0;
struct node
{
      int key, rank;
      char name[15];
      struct node *next;
};

struct hash
{
      struct node *head;
      int count;
};

void insertElement(int key, char *name, int rank)
{
      struct node *newnode;
      int hashIndex = key % hashTableSize;
      newnode = (struct node *)malloc(sizeof(struct node));
      newnode->key = key;
      newnode->rank = rank;
      strcpy(newnode->name, name);
      newnode->next = NULL;
      if(ht[hashIndex].head == NULL)
      {
            ht[hashIndex].head = newnode;
            ht[hashIndex].count = 1;
            return;
      }
      newnode->next = ht[hashIndex].head;
      ht[hashIndex].head = newnode;
       ht[hashIndex].count++;
}
void display()
{
      struct node *temp;
      int i;
      for (i = 0; i < hashTableSize; i++)
      {
            if (ht[i].count == 0)
                  continue;
            temp = ht[i].head;
            printf("\nData at index %d in Hash Table:\n", i);
            printf("StudentID     Name           Rank   \n");
           printf("-------------------------------\n");
            while (temp != NULL)
                  {
                        printf("%-12d", temp->key);
                        printf("%-15s", temp->name);
                       printf("%d\n", temp->rank);
```

```
                        temp = temp->next;
                }
        }
}
void deleteElement(int key) {
        int hashIndex = key % hashTableSize, flag = 0;
        struct node *temp, *myNode;
        myNode = ht[hashIndex].head;
        if (myNode == NULL) {
            printf("Given data is not present in hash Table!!\n");
                return;
        }
        temp = myNode;
        while (myNode != NULL) {
               if (myNode->key == key) {
                        flag = 1;
                        if (myNode == ht[hashIndex].head)
                                ht[hashIndex].head = myNode->next;
                        else
                                temp->next = myNode->next;

                        ht[hashIndex].count--;
                        free(myNode);
                        break;
                }
                temp = myNode;
                myNode = myNode->next;
        }
        if (flag)
            printf("Data deleted successfully from Hash Table\n");
        else
            printf("Given data is not present in hash
Table!!!!\n");
        return;
  }
void search(int key) {
        int hashIndex = key % hashTableSize;
        struct node *temp;
        temp = ht[hashIndex].head;
        if (temp == NULL) {
               printf("Search element unavailable in hash
table\n");
                return;
        }
        while (temp != NULL) {
                if (temp->key == key) {
                        printf("Regd.No  : %d\n", temp->key);
                        printf("Name     : %s\n", temp->name);
                        printf("Rank      : %d\n", temp->rank);
                        return;
                        }
                temp = temp->next;
        }
               printf("Search element unavailable in hash
table\n");
```

```
}
int main()
{
      int n, ch, key, rank;
      char name[100];
      printf("Enter the number of elements:");
      scanf("%d", &n);
    hashTableSize = n;
    ht = (struct hash *)calloc(n, sizeof (struct hash));
    while (1)
      {
            printf("\n1. Insertion\t2. Deletion\n");
            printf("3. Searching\t4. Display\n5. Exit\n");
            printf("Enter your choice:");
           scanf("%d", &ch);
            switch (ch)
            {
            case 1:
                        printf("Enter the key value:");
                       scanf("%d", &key);
                        printf("Name:");
                       scanf("%s",name);
                       printf("rank:");
                       scanf("%d", &rank);
                       insertElement(key, name, rank);
                      break;
                case 2:
                        printf("Enter the key to perform
deletion:");
                        scanf("%d", &key);
                        deleteElement(key);
                        break;
                case 3:
                        printf("Enter the key to search:");
                      scanf("%d", &key);
                        search(key);
                        break;
                case 4:
                        display();
                        break;
                default:
                        exit(0);
            }
      }
}
```

Applications of Hashing

In computing, **hash table** is used to implement

- Many types of in-memory tables. (symbol tables for example)
- Disk-based data structures and database indices.
- Caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media.
- Several dynamic languages, such as Perl, Python, JavaScript, and Ruby, use hash tables to implement objects.

Hash tables can also be used for storing massive amount of information. For ex:- To store driver's license information, telephone book databases, insurance details etc.