Data Structures

Lecture 11: Hashing

Dongbo Min
Department of Computer Science and Engineering
Ewha Womans University, Korea
E-mail: dbmin@ewha.ac.kr

Contents

- Hashing definition
- ADT of dictionary structure
- Hashing structure
- Hashing function
- Collision

Hashing

- Search operation is based on the comparison with key value
  - Finds the item to be searched by comparing the key with saved items
  - Time complexity: $O(n)$ for unsorted data, $O(log_2 n)$ for sorted data

- Hashing
  - Computes the item address on the hash table by an arithmetic operation on the key value, and then accesses the item
  - Time complexity: $O(1)$ ideally
  - Hashing is similar to organizing things

- Hash table
  - Structure that can be directly accessed by the key value

Abstract Data Type of Dictionary Structure

- Dictionary structure
  - Called as map or table
  - Consists of two fields associated with search.
    - Search key: e.g., an English word or a person's name
    - Value associated with key: e.g. the word definition, or phone number

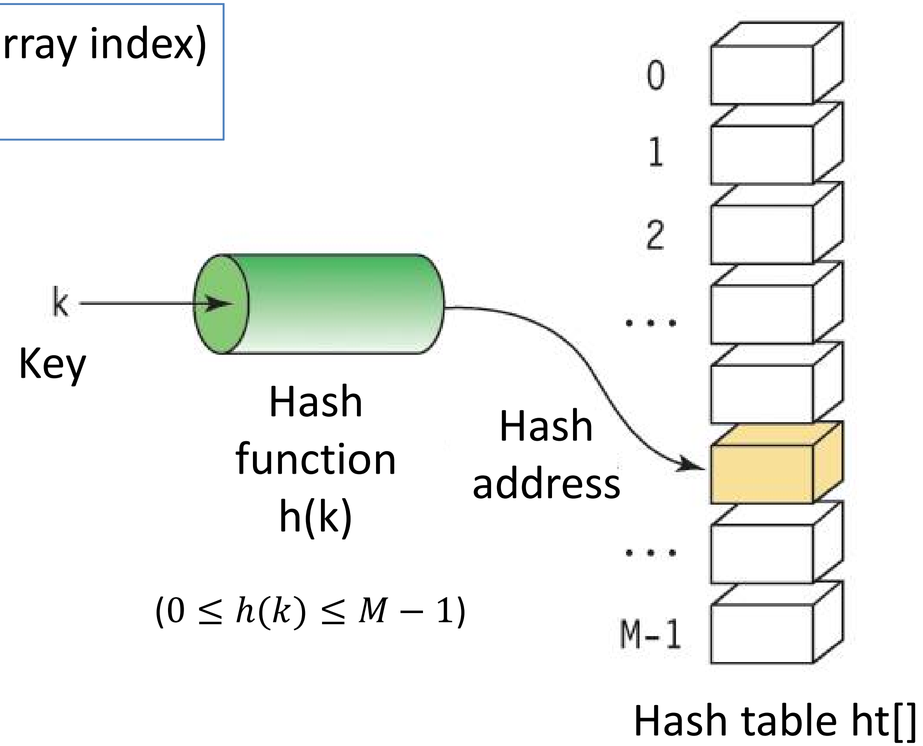
```
Object: a set of pairs of (key, value)

• Operation:
 - add(key, value) ::= add (key, value) to the dictionary.
 - delete(key) ::= delete (key, value) corresponding to key.
                   Return the associated value, or NULL if the search fails.
 - search(key) :: = find the value corresponding to key and return it.
                   If the search fails, return NULL.
```

Hashing Structure

- Hash function
  - Generate hash address using search key
  - Hash address: index of the hash table implemented as an array.

Hash: accesses the address (array index) of the item from the key

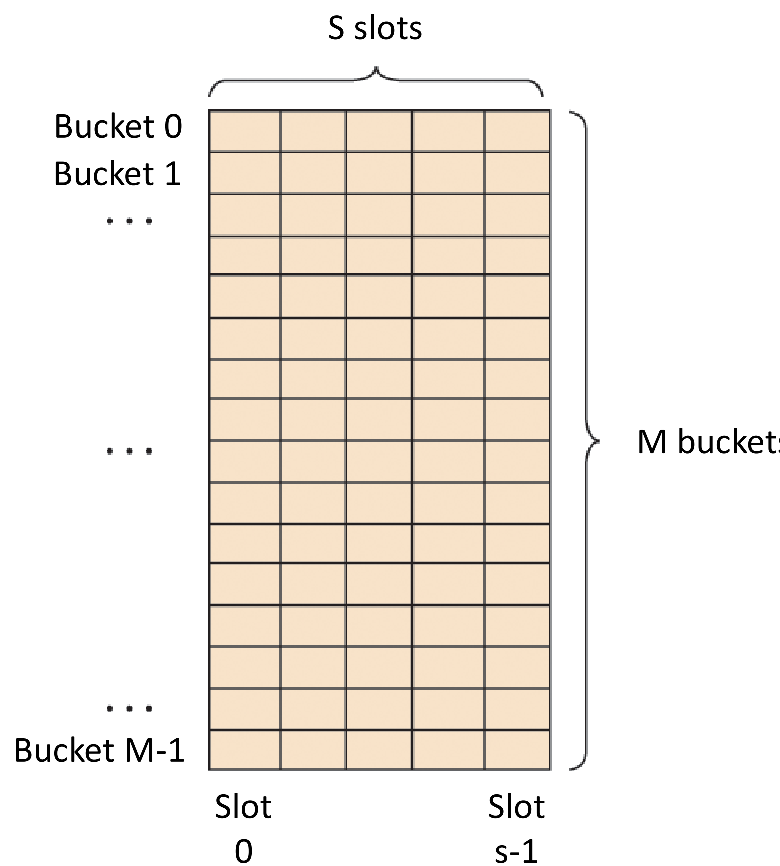


Hash Table

- Hash table
  - Table with M buckets
  - s slots for each bucket

- Collision
  - For different search key k1 and k2, the hash function h(k1)=h(k2)
  - Then, items are saved in the different slot in the same bucket.

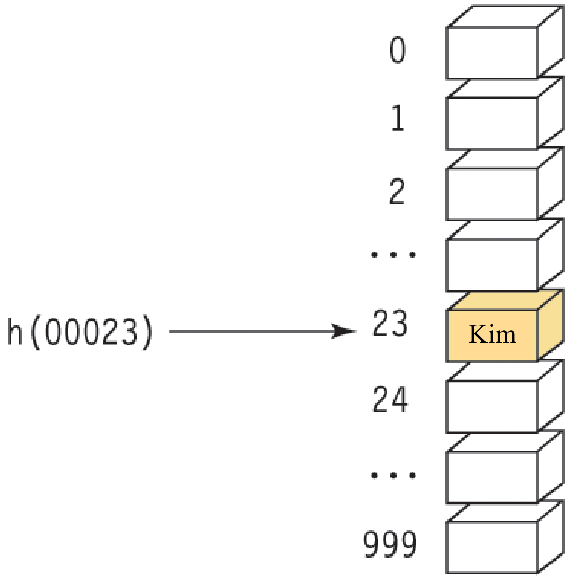
- Overflow
  - The collision occurs more than the number of slots allocated to the bucket
  - Then, the item cannot be saved any longer



Hash table ht[]

Ideal Hashing

- Example: save and search student data with hashing
  - Student ID: 5 digits (2 digits for department and 3 digits for student number)
  - For students from the same department, use only the last 3 digits.
  - When student ID: 00023, the student's information is stored in 'ht[23]'
  - If the hash table has 1000 spaces, the search time is O(1), which is ideal.
    - No collision (or overflow) occurs.



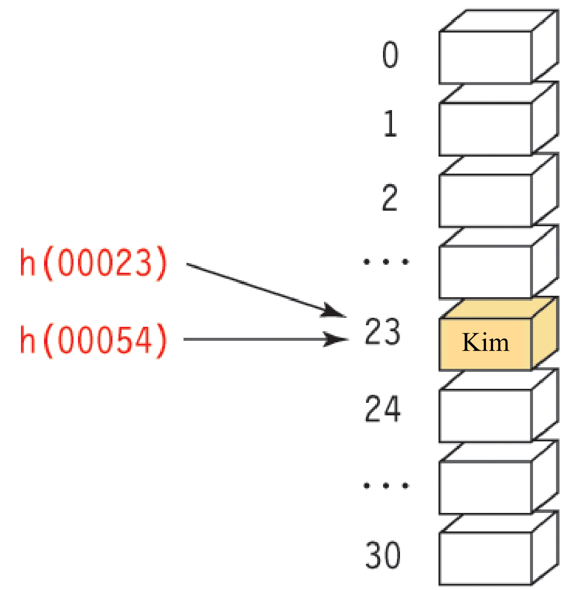
Hash table ht[]

Hashing in Practice

- In practice, the size of the hash table is limited
  -> you can not allocate storage space for all possible keys.

- Usually, Hash table size << # of possible keys
  - e.g., data size: 50,000,000   key: 13 digits

- Example: $h(k) = k \bmod M$
  - Note) collosions and overflows may occur



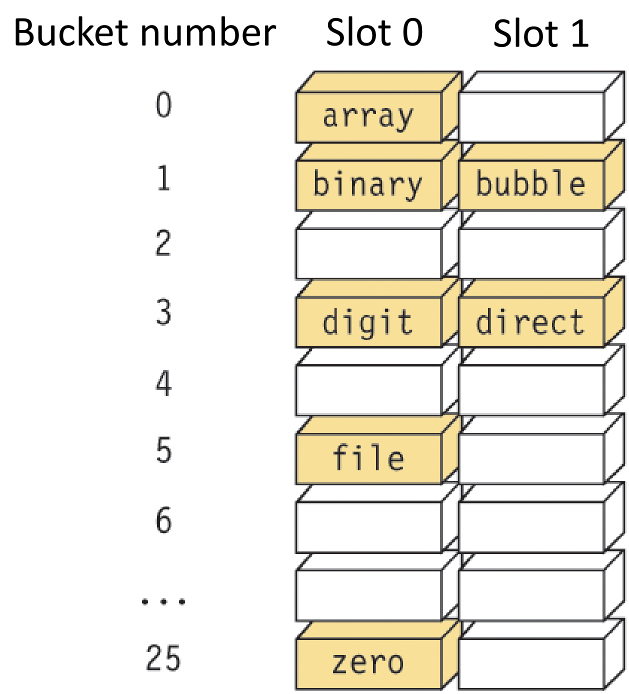
Hash table ht[]

Hashing in Practice

- Example)
  Search key: alphabet
  Hash function: returns the order of the first letter of the key

  h (“array”) = 1
  h (“binary”) = 2

- Input data: array, binary, bubble, file, digit, direct, zero, bucket



Hash table ht[]

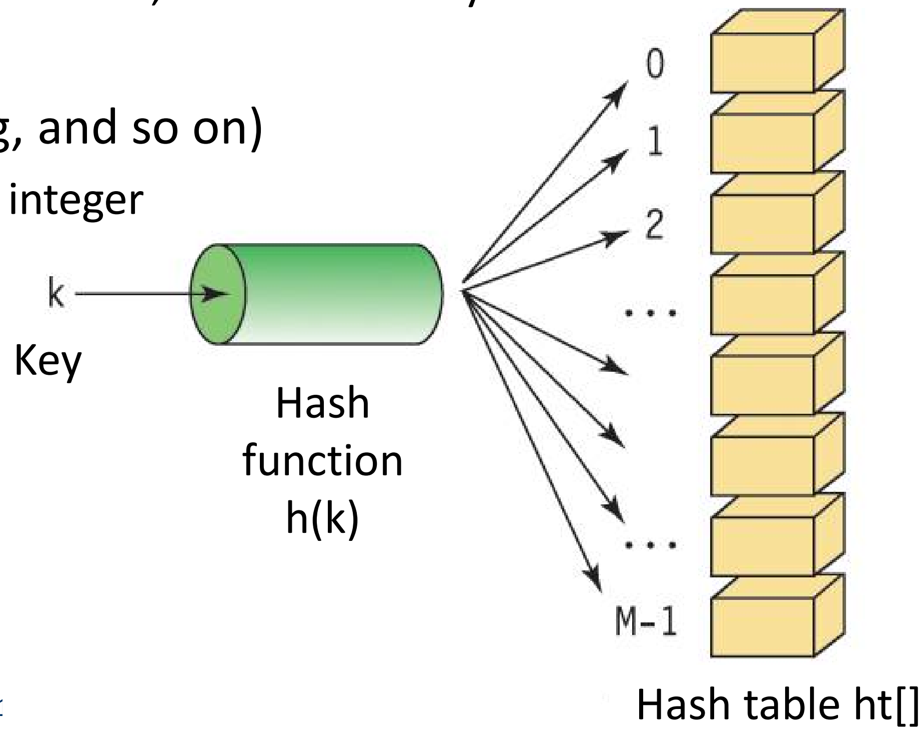
“bucket” cannot be saved due to overflow

Hash Function

- Condition for hash function
  - Fewer collisions
  - The hash function values should be distributed within the address space of the hash table as *evenly* as possible.
  - Fast computation

- Type of hash function
  - Division, folding, median-square, bit extraction, numerical analysis

- Key can be various types (integer, string, and so on)
  - In the beginning, assume that key is an integer



Hash Function

- Division function
  - $h(k) = k \bmod M$
  - The size M of the hash table is a *prime* number

    When if M is even number?

```
int hash_function(int key)
{
        int hash_index = key % M;
        if (hash_index < 0)
                hash_index += M;
        return hash_index;
}
```

Hash Function

- Folding function
  - Shift folding, boundary folding, and XOR folding

Address of hash table: decimal 3 digits

Search key	123	203	241	112	20

Shift folding	123	+	203	+	241	+	112	+	20	=	699
Boundary folding	123	+	302	+	241	+	211	+	20	=	897

XOR folding

Ex) Search key: 32 bit, Hash table: 16 bit

```
hash_index = (short)(key ^ (key>>16))
```

Hash Function

- Median-square function
  - Squares the search key, then takes a few bits to generate a hash address

- Bit extraction function
  - Hash table size: $M = 2^k$
  - Considering the search key as a binary number, use k bits at an arbitrary position as a hash address

- Numerical analysis method
  - Consider the distribution property of digits of key
  - Combine keys that are evenly distributed according to the size of the hash table

Ex) Student ID: 200812345

'2008' is used in all student ID.
So, do not use

Hash Function

- When the key is a string

- Simple solution
  - Each character is numbered as ‘a’ = 1, ‘b’=2, …, ‘z’=26
  - Using ASCII code or Unicode

    ASCII code ‘b’ of ‘book’ -> but, ‘cup’ and ‘car’ are indistinguishable

    Summing all ASCII codes of ‘book’ -> but, ‘are’ and ‘era’ are indistinguishable

  - Summing (ASCII code * value based on position)
    String s with n characters ($u_0 u_1 \ldots u_{n-1}$)

$$u_0 g^{n-1} + u_1 g^{n-2} + \cdots + u_{n-2} g + u_{n-1}$$

g: positive integer

$$\longrightarrow \left(..\left(\left(u_0 g + u_1\right) g + u_2\right) + \cdots + u_{n-2}\right) g + u_{n-1}$$

```
int hash_function(char *key)
{
        int hash_index = 0;
        while (*key)
                hash_index = g*hash_index + *key++;
        return hash_index;
}
```

Collision

- Collision
  - Items with *different* search keys have the *same* hash address
  - Cannot save items in hash table when collision occurs (for a single slot)

- Solutions to collision
  - Probing: stores conflicted items in a different location of the hash table (linear probing and quadratic probing)
  - Chaining: uses the linked list in each bucket

Linear Probing

- Procedure
  - If a collision occurs at ht[k], investigate whether ht [k + 1] is empty
  - If it is not empty, go to ht[k + 2]
  - Continue until finding an empty space
  - If you reach the end of the table, go to the first part of the hash table
  - If you come back to ht[k], the table is full.

- Problem: clustering and coalescing

Linear Probing

- Example: $h(k) = k \bmod 7$

Input: 8, 1, 9, 6, 13

Step 1 (8): h(8) = 8 mod 7 = 1 (store)

Step 2 (1): h(1) = 1 mod 7 = 1 (collision)
(h(1) +1) mod 7 = 2 (store)

Step 3 (9): h(9) = 9 mod 7 = 2 (collision)
(h(9) +1) mod 7 = 3 (store)

Step 4 (6): h(6) = 6 mod 7 = 6 (store)

Step 5 (13): h(13) = 13 mod 7 = 6 (collision)
(h(13) +1) mod 7 = 0 (store)

	Step 1	Step 2	Step 3	Step 4	Step 5
[0]					13
[1]	8	8	8	8	8
[2]		1	1	1	1
[3]			9	9	9
[4]					
[5]					
[6]				6	6

Example: “do”, “for”, “if”, “case”, “else”, “return”, “function”

Hash function: transform the string key into an integer by summing ASCII codes and then apply ‘key mod 13’
Linear probing is used. Hash table size = 13

Search key	Conversion	Sum	Hash address
do	100+111	211	3
for	102+111+114	327	2
if	105+102	207	12
case	99+97+115+101	412	9
else	101+108+115+101	425	9
return	114+101+116+117+115+110	672	9
function	102+117+110+99+116+105+111+110	870	12

Bucket	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
[0]							function
[1]							
[2]		for	for	for	for	for	for
[3]	do	do	do	do	do	do	do
[4]							
[5]							
[6]							
[7]							
[8]							
[9]				case	case	case	case
[10]					else	else	else
[11]						return	return
[12]			if	if	if	if	if

```c
#define KEY_SIZE10// Maximum size of search key
#define TABLE_SIZE13// Hash table size (prime number)

typedef struct element {
           char key[KEY_SIZE];
} element;
element hash_table[TABLE_SIZE];

// Initialize the hash table
void init_table(element ht[]) {
           //each bucket is initialized as null
           for (int i = 0; i < TABLE_SIZE; i++)
                      ht[i].key[0] = NULL;
}
// Transform the string key into an integer by summing ASCII codes
int transform(char *key) {
           int number = 0;
           while (*key)
                      number += *key++;
           return number;
}
// Division function ( key mod TABLE_SIZE )
int hash_function(char *key) {
           return transform(key) % TABLE_SIZE;
}
```

```c
#define empty(e) (strlen(e.key)==0)
#define equal(e1, e2) (!strcmp(e1.key, e2.key))
// Add the key into the hash table
// Collision is handled using the linear probing
void hash_lp_add(element item, element ht[]) {
        int i, hash_value;
        hash_value = i = hash_function(item.key);
        while (!empty(ht[i])) {
                if (equal(item, ht[i])) {
                        fprintf(stderr, "Duplicate search key\n");
                        return;
                }
                i = (i + 1) % TABLE_SIZE;
                if (i == hash_value) {
                        fprintf(stderr, "Table is full (overflow).\n");
                        return;
                }
        }
        ht[i] = item;
}
void hash_lp_search(element item, element ht[])
{
        int i, hash_value;
        hash_value = i = hash_function(item.key);
        while (!empty(ht[i])) {
                if (equal(item, ht[i])) {
                        fprintf(stderr, "Search success: position = %d\n", i);
                        return;
                }
                i = (i + 1) % TABLE_SIZE;
                if (i == hash_value) {
                        fprintf(stderr, "Search key is not in hash table.\n");
                        return;
                }
        }
}
```

```c
void hash_lp_print(element ht[])
{
		for (int i = 0; i < TABLE_SIZE; i++)
				printf("[%d]%s\n", i, ht[i].key);
}

void main()
{
		element tmp;
		int op;
		while (1) {
				printf("Enter the operation to do (0: insert, 1: search, 2: termination): ");
				scanf_s("%d", &op);
				if (op == 2)		break;

				printf("Enter the search key: ");
				scanf_s("%s", tmp.key, sizeof(tmp.key));
				if (op == 0)
						hash_lp_add(tmp, hash_table);
				else if (op == 1)
						hash_lp_search(tmp, hash_table);
				hash_lp_print(hash_table);
				printf("\n");
		}
}
```

Execution results

```
Enter the operation to do (0: insert, 1: search, 2: termination): 0
Enter the search key: and
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]    and
[9]
[10]
[11]
[12]

Enter the operation to do (0: insert, 1: search, 2: termination): 0
Enter the search key: dna
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]    and
[9]    dna
[10]
[11]
[12]
Enter the operation to do (0: insert, 1: search, 2: termination):
```

Quadratic Probing

- Quadratic Probing
  - $(h(k) + inc * inc) \bmod M$
  - The locations examined are
    $h(k), h(k) + 1, h(k) + 4, h(k) + 9, \ldots$
  - Clustering and coalescing, which is a problem in linear probing, can be greatly mitigated.

Double Hashing

- Double hashing
  - Also known as rehashing
  - When an overflow occurs, it uses a *different* hash function than the original hash function.

$$step = C - (k \bmod C)$$

$$h(k), h(k) + step, h(k) + 2 \times step, \ldots$$

  - Note) Linear or quadratic probing use $step = i$ or i^2 for $i = 1,2,..$

Example) In the hash table of size 7,
The first hash function: $k \bmod 7$
When an overflow occurs, hash function is applied with $step = 5 - (k \bmod 5)$

Step 1 (8) : h(8) = 8 mod 7 = 1 (Store)

Step 2 (1) : h(1) = 1 mod 7 = 1 (Collision)   step = 5-(1 mod 5) = 4
(h(1) + step) mod 7 = (1 + 4) mod 7 = 5 (Store)

Step 3 (9) : h(9) = 9 mod 7 = 2 (Store)

Step 4 (6) : h(6) = 6 mod 7 = 6 (Store)

Step 5 (13) : h(13) = 13 mod 7 = 6 (Collision)   step = 5-(13 mod 5) = 2
(h(13) + step) mod 7 = (6 + 2) mod 7= 1 (Collision)
(h(13) + 2*step) mod 7 = (6 + 4) mod 7= 3 (Store)

Input (8, 1, 9, 6, 13)

	Step 1	Step 2	Step 3	Step 4	Step 5
[0]					
[1]	8	8	8	8	8
[2]			9	9	9
[3]					13
[4]					
[5]		1	1	1	1
[6]				6	6

```c
int hash_function2(char *key)
{
		int C = 5;
		return ( C -( transform(key) % C) );
}

void hash_dh_add(element item, element ht[])
{
		int i, hash_value, inc;
		hash_value = i = hash_function(item.key);
		inc = hash_function2(item.key);
		while (!empty(ht[i])) {
				if (equal(item, ht[i])) {
						fprintf(stderr, "Duplicate search key\n");
						return;
				}
				i = (i + inc) % TABLE_SIZE;
				if (i == hash_value) {
						fprintf(stderr, "Table is full (overflow).\n");
						return;
				}
		}
		ht[i] = item;
}
```

Chaining

- Address collision and overflow issues with *linked lists*
  - Each bucket is not assigned a fixed slot, but a linked list that is easy to insert and delete
  - Sequential search in the linked list for each bucket

Problems of linear/quadratic probing and double hashing

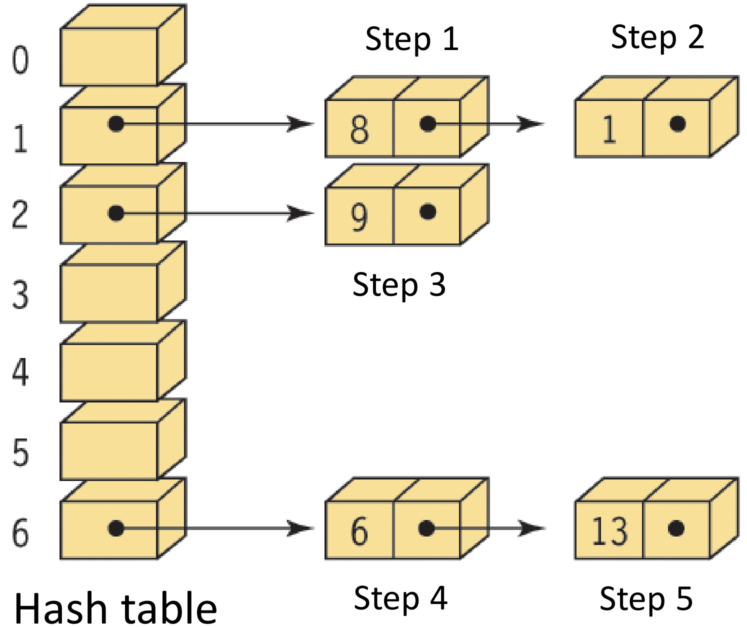
Search becomes slow when many items are compared due to collisions.

```
void hash_lp_search(element item, element ht[])
{
		int i, hash_value;
		hash_value = i = hash_function(item.key);
		while (!empty(ht[i])) {
				if (equal(item, ht[i])) {
						fprintf(stderr, "Search success: position = %d\n", i);
						return;
				}
				i = (i + 1) % TABLE_SIZE;
				if (i == hash_value) {
						fprintf(stderr, "Search key is not in hash table.\n");
						return;
				}
		}
}
```

(Example) In the hash table of size 7
Hash function $h(k) = k \bmod 7$

Input (8, 1, 9, 6, 13)

Step 1 (8) : h(8) = 8 mod 7 = 1 (Store)
Step 2 (1) : h(1) = 1 mod 7 = 1 (Collision -> store the item in newly generated node)
Step 3 (9) : h(9) = 9 mod 7 = 2 (Store)
Step 4 (6) : h(6) = 6 mod 7 = 6 (Store)
Step 5 (13) : h(13) = 13 mod 7 = 6 (Collision -> Store the item in newly generated node)



```c
#define KEY_SIZE 10
#define TABLE_SIZE 13

typedef struct element {
		char key[KEY_SIZE];
} element;
typedef struct ListNode {
		element item;
		ListNode *link;
} ListNode;
ListNode *hash_table[TABLE_SIZE];

// Transform the string key into an integer by summing ASCII codes
int transform(char *key) {
		int number = 0;
		while (*key)
				number += *key++;
		return number;
}
// Division function ( key mod TABLE_SIZE )
int hash_function(char *key) {
		return transform(key) % TABLE_SIZE;
}
```

```c
#define equal(e1, e2) (!strcmp(e1.key, e2.key))
void hash_chain_add(element item, ListNode *ht[])
{
	int hash_value = hash_function(item.key);
	ListNode *ptr;
	ListNode *node_before = NULL;
	ListNode *node = ht[hash_value];

	for (; node; node_before = node, node = node->link)
	{
		if (equal(node->item, item)) {
			fprintf(stderr, "Duplicate search key\n");
			return;
		}
	}
	ptr = (ListNode *)malloc(sizeof(ListNode));
	ptr->item = item;
	ptr->link = NULL;
	if (node_before)
		node_before->link = ptr;
	else
		ht[hash_value] = ptr;
	}

void hash_chain_search(element item, ListNode *ht[])
{
	ListNode *node;
	int hash_value = hash_function(item.key);
	for (node = ht[hash_value]; node; node = node->link) {
		if (equal(node->item, item)) {
			printf("Search success\n");
			return;
		}
	}
	printf("Search failed\n");
}
```

```c
void hash_chain_print(ListNode *ht[])
{
	ListNode *node;
	for (int i = 0; i < TABLE_SIZE; i++) {
		printf("[%d]", i);
		for (node = ht[i]; node; node = node->link)
			printf(" -> %s", node->item.key);
		printf(" -> null\n");
	}
}
void init_table(ListNode *ht[])
{
	for (int i = 0; i < TABLE_SIZE; i++)
		ht[i] = NULL;//each node is initialized as null
}

void main()
{
	element tmp;
	int op;
	init_table(hash_table);
	while (1) {
		printf("Enter the operation to do (0: insert, 1: search, 2: termination): ");
		scanf_s("%d", &op);
		if (op == 2)		break;

		printf("Enter the search key: ");
		scanf_s("%s", tmp.key, sizeof(tmp.key));
		if (op == 0)
			hash_chain_add(tmp, hash_table);
		else if (op == 1)
			hash_chain_search(tmp, hash_table);
		hash_chain_print(hash_table);
		printf("\n");
	}
}
```

```
Enter the operation to do (0: insert, 1: search, 2: termination): 0
Enter the search key: and
[0] -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> null
[7] -> null
[8] -> and -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null

Enter the operation to do (0: insert, 1: search, 2: termination): 0
Enter the search key: test
[0] -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> test -> null
[7] -> null
[8] -> and -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null
```

```
Enter the operation to do (0: insert, 1: search, 2: termination): 0
Enter the search key: dna
[0] -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> test -> null
[7] -> null
[8] -> and -> dna -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null
Enter the operation to do (0: insert, 1: search, 2: termination):
```

Performance Analysis of Hashing

- Ideal hashing with no collision
  - Time complexity of search operation: $O(1)$
- In practice, collision occurs hashing
  - Time complexity of search operation > $O(1)$
  - It depends on the loading density of hash table
- Loading density (or loading factor)
  - The ratio of the number n of stored items to the size M of the hash table

$$\alpha = \frac{\#\ of\ stored\ items}{hash\ table\ size} = \frac{n}{M}$$

Linear probing: $0 \leq \alpha \leq 1$

Chaining: $0 \leq \alpha$

Assume that single slot is assigned for each bucket.

Note) no maximum of α exists in the chaining.

Performance Analysis of Hashing

The number of comparison operations in linear probing

α	Failed search	Successful search
0.1	1.1	1.1
0.3	1.5	1.2
0.5	2.5	1.5
0.7	6.1	2.2
0.9	50.5	5.5

Failed search $$\frac{1}{2}\left\{1+\frac{1}{(1-\alpha)^2}\right\}$$

Successful search $$\frac{1}{2}\left\{1+\frac{1}{(1-\alpha)}\right\}$$

The number of comparison operations in chaining

α	Failed search	Successful search
0.1	0.1	1.1
0.3	0.3	1.2
0.5	0.5	1.3
0.7	0.7	1.4
0.9	0.9	1.5
1.3	1.3	1.7
1.5	1.5	1.8
2.0	2.0	2.0

Failed search $$\alpha$$

Each bucket contains a linked list with α items on average

Successful search $$1+\frac{\alpha}{2}$$

Performance Analysis of Hashing

$\alpha = n/M$	0.5		0.7		0.9		0.95	
Hashing function	Chaining	Linear probing	Chaining	Linear probing	Chaining	Linear probing	Chaining	Linear probing
Median-Square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
Division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
Shift folding	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
Boundary folding	1.39	22.97	1.57	48.70	1.55	69.63	1.55	97.56
Numerical Analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
Theoretic	1.25	1.50	1.37	2.50	1.45	5.50	1.45	10.50

From V. Lum, P. Yuen, M. Dodd, CACM, 1971, Vol.14, No.4

Observation

- In the linear probing, keep $\alpha \leq 0.5$.
- In the quadratic probing and double hashing, keep $\alpha \leq 0.7$ (though not showed in table).
- For small α, the linear probing is better than the quadratic probing and double hashing.
- Chaining performance is linearly proportional to α, different from probing and double hashing.

Comparison with Other Search Methods

Search methods		Search	Insert	Deletion
Sequential search		$O(n)$	$O(1)$	$O(n)$
Binary search (using array)		$O(log_2 n)$	$O(n + log_2 n)$	$O(n + log_2 n)$
Binary search tree	Balanced tree	$O(log_2 n)$	$O(log_2 n)$	$O(log_2 n)$
	Oblique tree	$O(n)$	$O(n)$	$O(n)$
Hashing	Best case	$O(1)$	$O(1)$	$O(1)$
	Worst case	$O(n)$	$O(n)$	$O(n)$

Binary search vs. Hashing

- Hashing is usually faster than binary search.
- Insert is easier in hashing than binary search.

Binary search tree vs. Hashing

- In binary search, it is easier to 1) find larger or smaller data than current data and 2) traverse the data in order than hashing
- Data in hash table is not organized in order! (e.g. $k \bmod M$, probing, and chaining)
- Hash table size is hard to predict in advance.
- Worst time complexity of hashing is $O(n)$.