Laboratory 8: Hashing and Hash tables

CSC205A Data structures and Algorithms Laboratory B. Tech. 2015

Vaishali R Kulkarni

Department of Computer Science and Engineering
Faculty of Engineering and Technology
M. S. Ramaiah University of Applied Sciences
Email: vaishali.cs.et@msruas.ac.in
Tel: +91-80-4906-5555 (2212) WWW: www.msruas.ac.in

Introduction and Purpose of Experiment

- Hashing is an efficient search technique used in many applications.
- A **hash table** is a collection of items which are stored in such a way as to make it easy to find them later.
- In this experiments will learn how to implement and search an element in hash table and analyse the efficiency of hashing technique.

Aim and objectives

Aim:

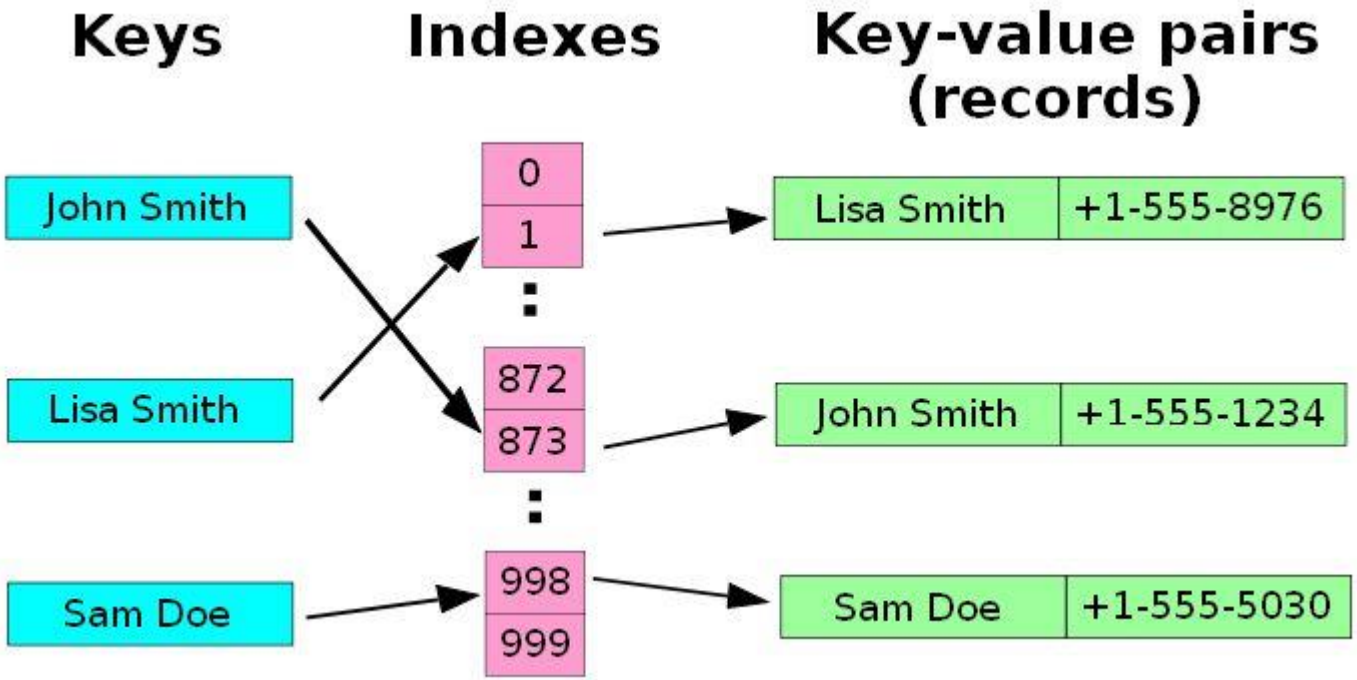
- To design and develop hash table in C and demonstrate search operation

Objectives:

At the end of this lab, the student will be able to

- Implement hash table
- Search an element using hash table
- Design different collision resolution strategies
- Analyze the performance of hash table

Example



A small phone book as a hash table.

(

Data Structure for Hash Table

```
#define MAX_CHAR  10
#define TABLE_SIZE  13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

Ideal Hashing

- Uses an array table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
- Every dictionary pair (key, element) is stored in its home bucket table[f[key]].

Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key (mod 11).

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The bucket for (26,g) is already occupied.

Some Issues

- **Choice of hash function.**
  - ***Really tricky!***
  - To avoid collision (two different pairs are in the same the same bucket.)
  - Size (number of buckets) of hash table.
- **Overflow handling method.**
  - Overflow: there is no space in the bucket for the new pair.

Example (fig 8.1)

synonyms:
char, ceil,
clock, ctime

↑

overflow

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

synonyms

synonyms

Choice of Hash Function

- Requirements
  - easy to compute
  - minimal number of collisions
- If a hashing function groups key values together, this is called clustering of the keys.
- A good hashing function distributes the key values uniformly throughout the range.

Some hash functions

- Middle of square
  - H(x):= return middle digits of x^2
- Division
  - H(x):= return x % k
- Multiplicative:
  - H(x):= return the first few digits of the fractional part of x*k, where k is a fraction.

Some hash functions II

- Folding:
  - Partition the identifier x into several parts, and add the parts together to obtain the hash address
  - e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699
  - Shift folding vs. folding at the boundaries
- Digit analysis:
  - If all the keys have been known in advance, then we could delete the digits of keys having the most skewed distributions, and use the rest digits as hash address.

Hashing By Division

- Domain is all integers.
- For a hash table of size *b*, the number of integers that get hashed into bucket *i* is approximately $\mathbf{2^{32}/b}$.
- The division method results in a uniform hash function that maps approximately the same number of keys into each bucket.

Hashing By Division II

- In practice, keys tend to be correlated.
  - If divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
    - 20%14 = 6, 30%14 = 2, 8%14 = 8
    - 15%14 = 1, 3%14 = 3, 23%14 = 9
  - divisor is an odd number, odd (even) integers may hash into any home.
    - 20%15 = 5, 30%15 = 0, 8%15 = 8
    - 15%15 = 0, 3%15 = 3, 23%15 = 8

Hashing By Division III

- Similar biased distribution of home buckets is seen in practice, when the divisor is a multiple of prime numbers such as 3, 5, 7, …
- The effect of each prime divisor p of b decreases as p gets larger.
- Ideally, choose large prime number b.
- Alternatively, choose b so that it has no prime factors smaller than 20.

Hash Algorithm via Division

```
void init_table(element ht[])
{
  int i;
  for (i=0; i<TABLE_SIZE; i++)
    ht[i].key[0]=NULL;
}

int transform(char *key)
{
  int number=0;
  while (*key) number += *key++;
  return number;
}
```

```
int hash(char *key)
{
  return (transform(key)
          % TABLE_SIZE);
}
```

Criterion of Hash Table

- The key density (or identifier density) of a hash table is the ratio n/T
  - n is the number of keys in the table
  - T is the number of distinct possible keys
- The loading density or loading factor of a hash table is $\alpha = n/(sb)$
  - s is the number of slots
  - b is the number of buckets

Overflow Handling

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- We may handle overflows by:
  - Search the hash table in some systematic fashion for a bucket that is not full.
    - Linear probing (linear open addressing).
    - Quadratic probing.
    - Random probing.
  - Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.
    - Array linear list.
    - Chain.

Linear probing (linear open addressing)

- **Open addressing** ensures that all elements are stored directly into the hash table, thus it attempts to resolve collisions using various methods.

- **Linear Probing** resolves collisions by placing the data into the next open slot in the table.

Linear Probing – Get And Insert

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing – Delete

0			4					8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Delete(0)

0			4					8				12				16
34		45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0			4					8				12				16
34	45					6	23	7			28	12	29	11	30	33

Linear Probing – Delete(34)

0			4					8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0			4					8				12				16
	0	45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0			4					8				12				16
0		45				6	23	7			28	12	29	11	30	33

0			4					8				12				16
0	45					6	23	7			28	12	29	11	30	33

Linear Probing – Delete(29)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
34	0	45				6	23	7			28	12		11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	0	45				6	23	7			28	12	11		30	33

0				4				8				12				16
34	0	45				6	23	7			28	12	11	30		33

0				4				8				12				16
34	0					6	23	7			28	12	11	30	45	33

Linear Probing

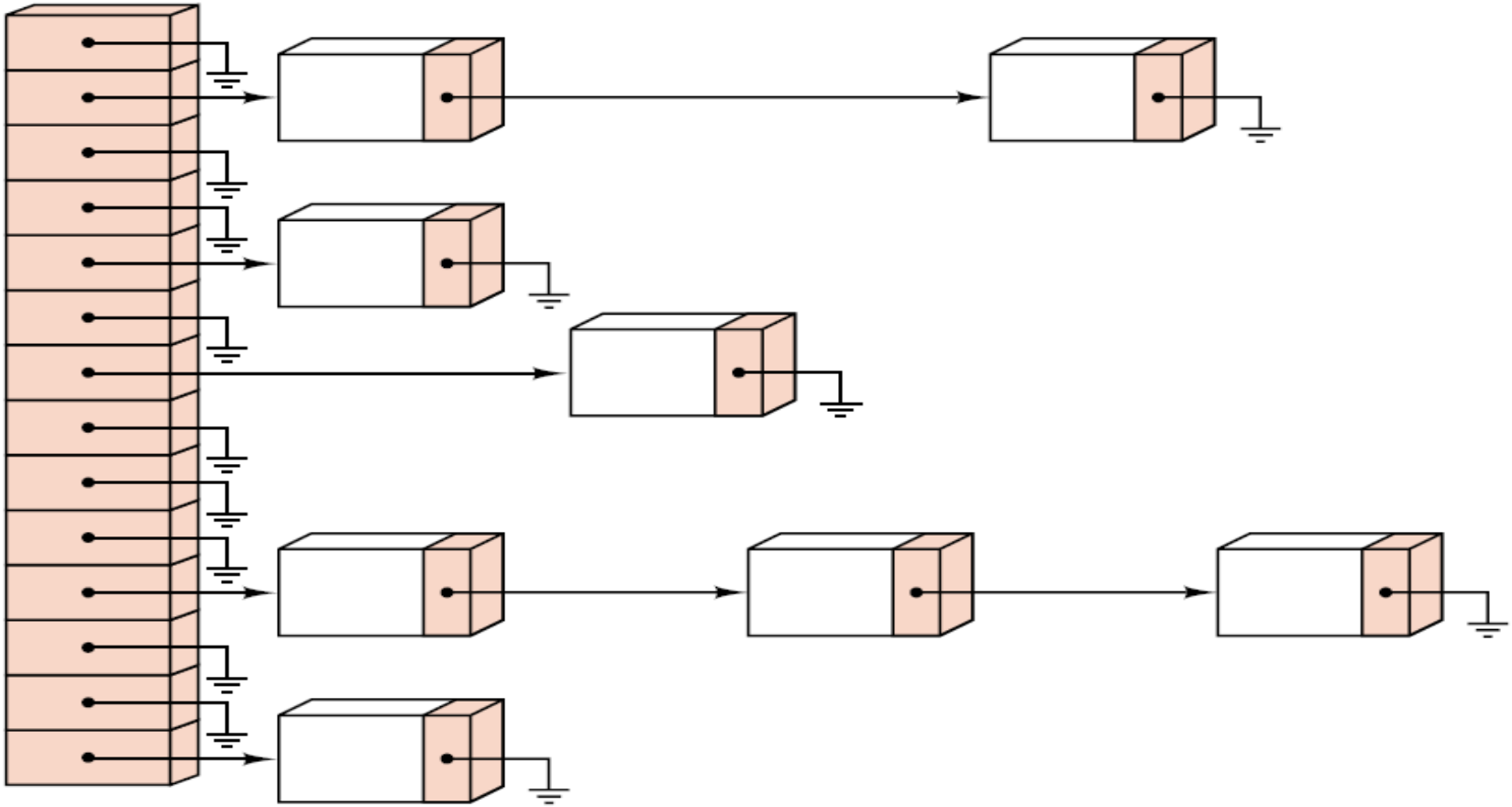
```
void linear_insert(element item, element ht[]){
  int i, hash_value;
  i = hash_value = hash(item.key);
  while(strlen(ht[i].key)) {
      if (!strcmp(ht[i].key, item.key)) {
                fprintf(stderr, “Duplicate entry\n”);  exit(1);
     }
     i = (i+1)%TABLE_SIZE;
     if (i == hash_value) {
           fprintf(stderr, “The table is full\n”); exit(1);
     }  }
  ht[i] = item;
}
```

Data Structure for Chaining

```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))
typedef struct {
   char key[MAX_CHAR];
   /* other fields */
} element;
typedef struct list *list_pointer;
typedef struct list {
   element item;
   list_pointer link;
};
list_pointer hash_table[TABLE_SIZE];
```

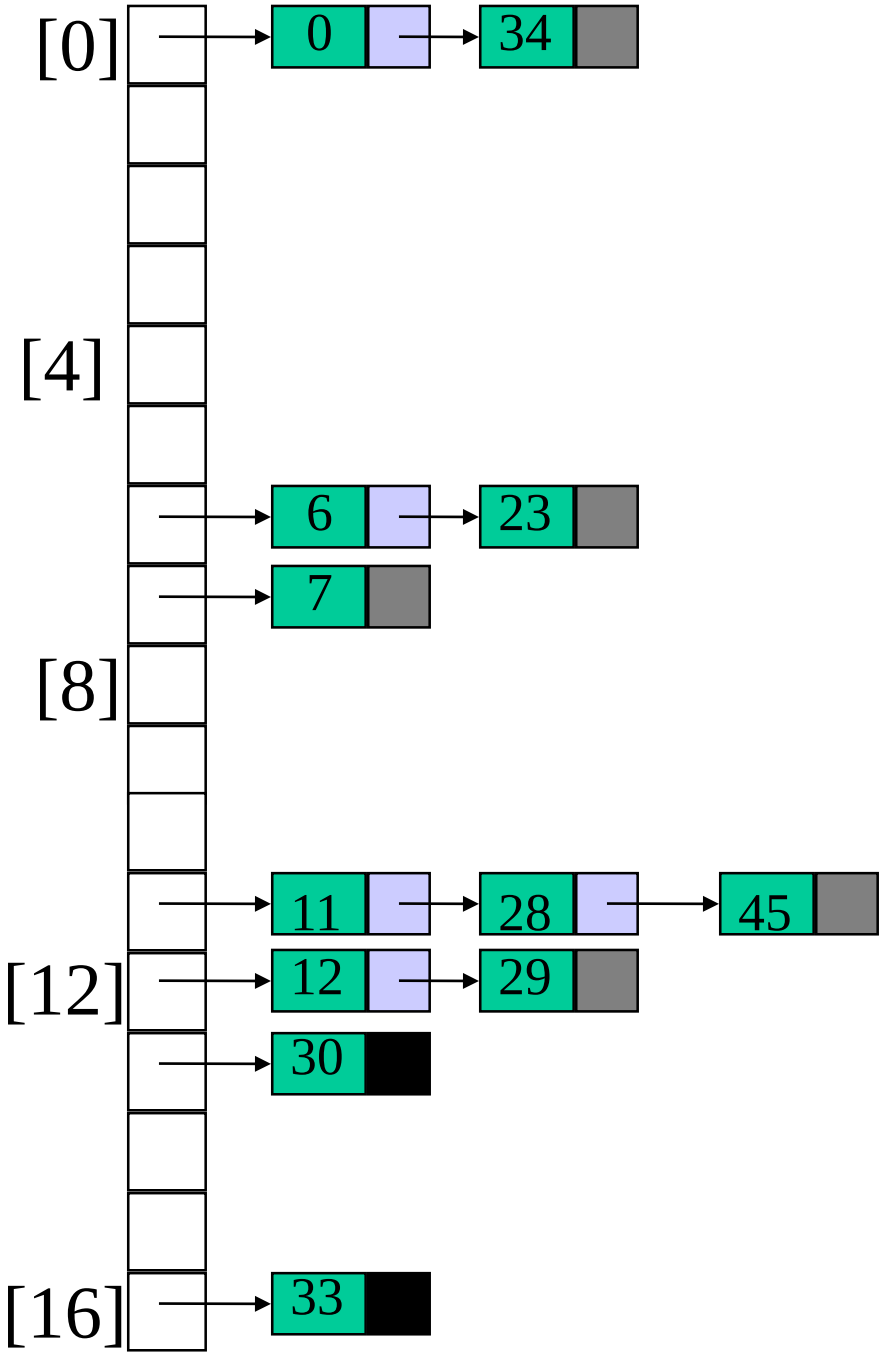
The idea of **Chaining** is to combine the linked list and hash table to solve the overflow problem.

Chaining



Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Bucket = key % 17.



Comparison : Load Factor

- If **open addressing** is used, then each table slot holds at most one element, therefore, the loading factor can **never** be greater than 1.

- If **external chaining** is used, then each table slot can hold many elements, therefore, the loading factor **may be** greater than 1.

Experimental Procedure

- Analyse the problem statement
- Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
- Implement the algorithm in C language
- Compile the C program
- Design test cases and test the implemented program
- Document the Results
- Analyse and discuss the outcomes of your experiment

Exercise

- Design an algorithm and write a program to create a hash table and use suitable hash function and demonstrate search operation for a given data. Tabulate the output for various inputs and verify against expected values. Analyse the efficiency of hash function used. Describe your learning along with the limitations of both, if any. Suggest how these can be overcome.

Key factors and discussion

- Application of different hash functions
  - Demonstrate what is a good and bad hash functions
  - Apply different hash functions
  - Analyse the performance
- Collison resolution
  - Understand the concept of collisions and how it is resolved using chaining strategy

Results and Presentations

- Calculations/Computations/Algorithms

  The calculations/computations/algorithms involved in each program has to be presented

- Presentation of Results

  The results for all the valid and invalid cases have to be presented

- Analysis and Discussions

  how the data is manipulated or transformed, what are the key operations involved. Errors encounters and how they are resolved.

- Conclusions

  Summary

Comments

- Limitations of Experiments

  Outline the loopholes in the program, data structures or solution approach.

- Limitations of Results

  Present the test cases; justify if the program is tested correctly considering all the outcomes. Mention what is not tested, if any.

- Learning happened

  What is the overall learning happened

- Conclusions

  Summary

References

- Gilberg, R. F., and Forouzan, B. A. (2007): A Pseudocode Approach With C, 2nd edn. Cengage Learning
- Slide 5 Figure is from Wikipedia
-