



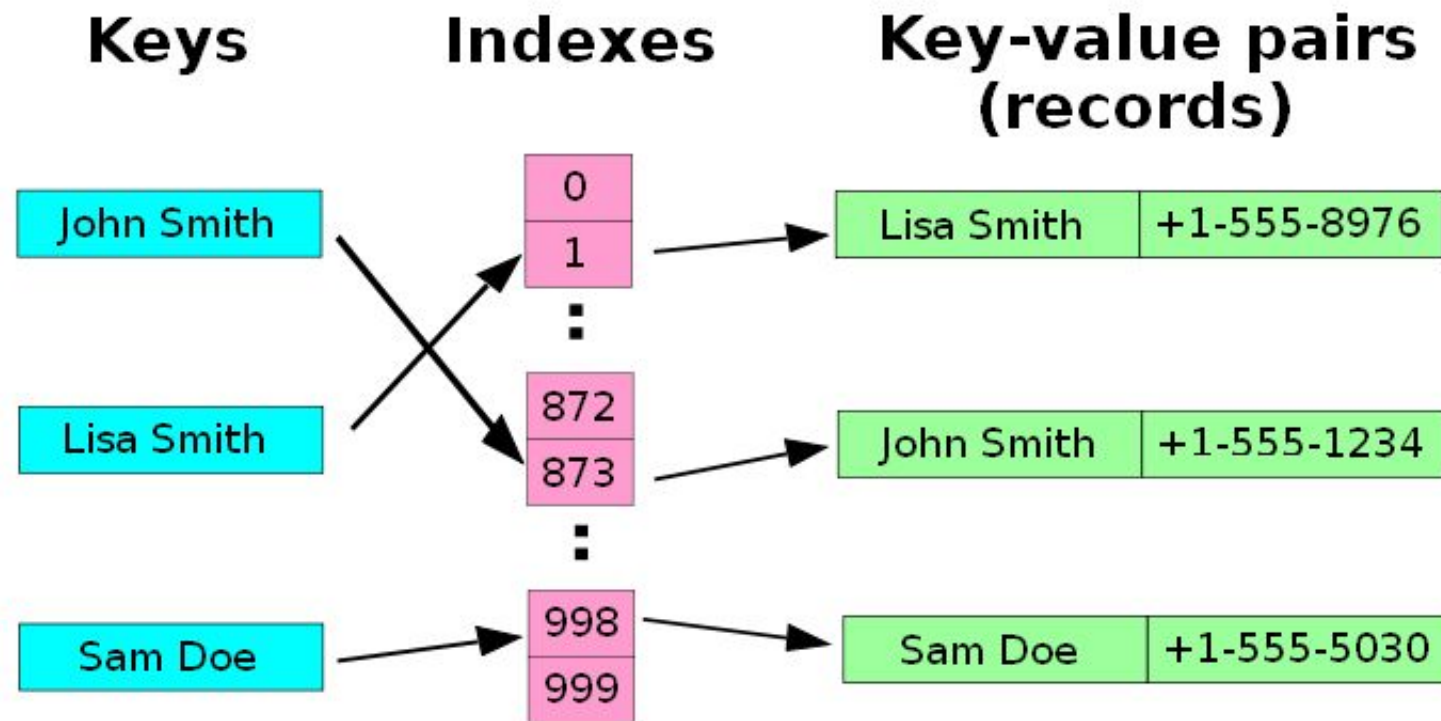
Chapter 14 Hashing



Concept of Hashing

- A **hash table** is a data structure that associates keys (names) with values (attributes).
- Look-Up Table
- Dictionary
- Cache
- Memcached: a.k.a. distributed hash table

Example



A small phone book as a hash table.

(Figure from Wikipedia)



Dictionaries

- Collection of pairs
 - (key, value)
 - Each pair has a unique key
- Operations.
 - Get(key)
 - Delete(key)
 - Insert(key, value)



Overall Idea

- Hash table :
 - Collection of pairs,
 - Lookup function (Hash function)
- Hash tables are often used to implement associative arrays,
 - Worst-case time for **Get**, **Insert**, and **Delete** is **$O(\text{size})$** .
 - Expected time is **$O(1)$** .



Origins of the Term

- The term "**hash**" comes by way of analogy with its standard meaning in the physical world, to "**chop and mix.**"
D. Knuth notes that **Hans Peter Luhn** of IBM appears to have been the first to use the concept, in a memo dated January 1953; the term **hash** came into use some ten years later.



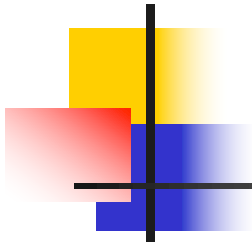
Search vs. Hashing

- Search tree methods: key comparisons
 - Time complexity: $O(\text{size})$ or $O(\log n)$
- Hashing methods: hash functions
 - Expected time: $O(1)$
- Types
 - Static hashing
 - Dynamic hashing



Static Hashing

- Key-value pairs are stored in a fixed size table called a *hash table*.
 - A hash table is partitioned into many *buckets*.
 - Each bucket has many *slots*.
 - Each slot holds one record.
 - A hash function $f(x)$ transforms the identifier (key) into an address in the hash table



Hash table

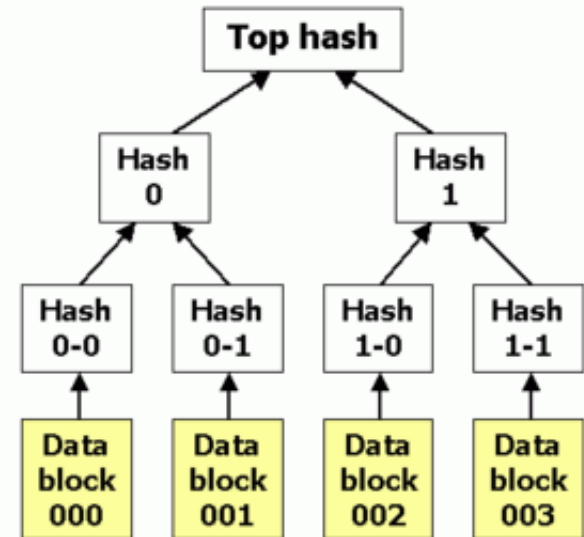
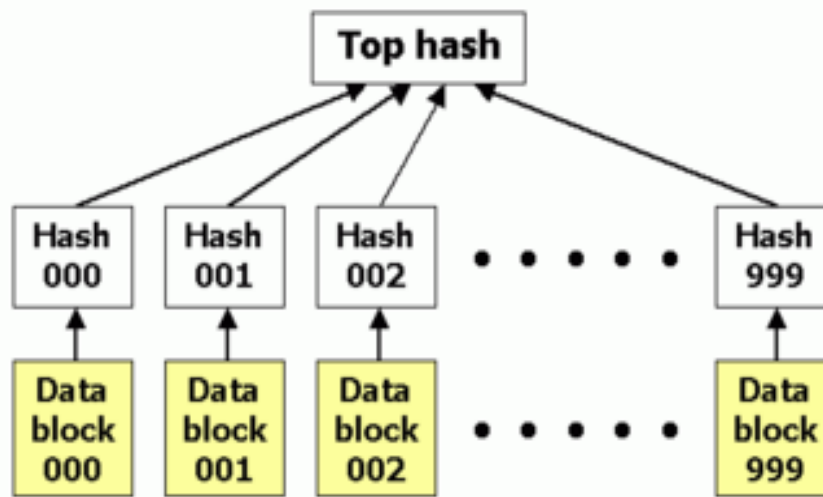
		s slots			
		0	1		s-1
b buckets	0			...	
	1				
	
	b-1			...	



Data Structure for Hash Table

```
#define MAX_CHAR 10
#define TABLE_SIZE 17
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

Other Extensions

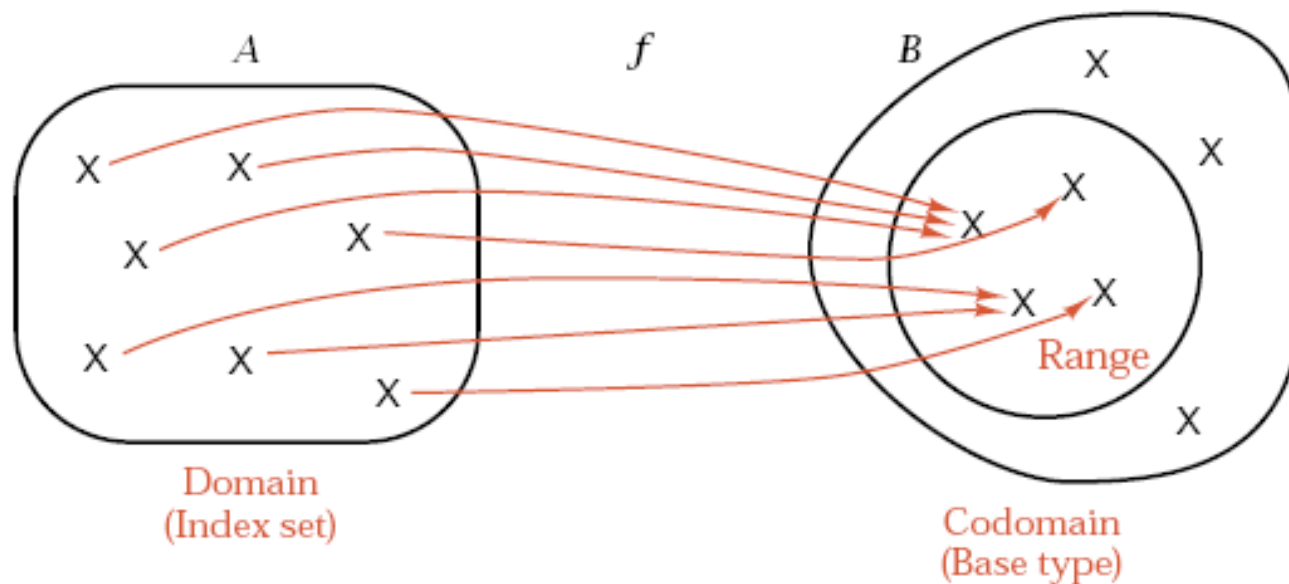


Hash List and Hash Tree

(Figure is from Wikipedia)

Formal Definition

- Hash Function
 - In addition, one-to-one / onto





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]]`.



What Can Go Wrong?

- More than one keys can be hashed to the same bucket
- Keys that have the same home bucket are **synonyms**
- How to deal with this?
 - First, choose a good hash function to minimize collisions
 - Second, handle overflow efficiently



Some Issues

- **Choice of hash function**

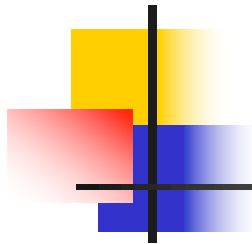
- *Really tricky!*

- To avoid **collision** where two different pairs are in the same bucket

- Size (number of buckets) of hash table is desired to be small

- **Overflow handling method**

- **Overflow**: there is no space in the bucket for the new pair.



Example

synonyms:
char, ceil,
clock, ctime



overflow

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



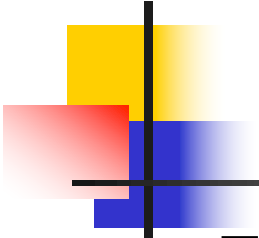
Choice of Hash Function

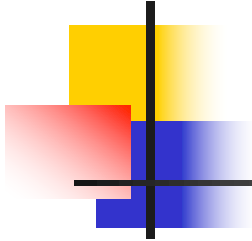
- Requirements
 - easy to compute
 - minimal number of collisions
- If a hashing function groups key values together, this is called **clustering** of the keys
- A good hashing function distributes key values uniformly throughout the range
- Ideally, simple uniform hashing



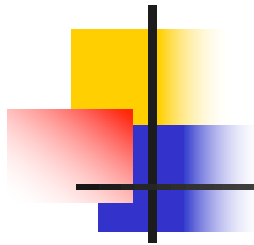
Some hash functions

- Middle of square
 - $H(x) :=$ return middle digits of x^2
- Division
 - $H(x) :=$ return $x \% m$
 - Choosing good m is tricky
 - Bad example: m is power of 2
 - Generally, choose a prime that is not so close to power of 2 as m

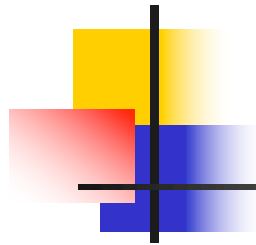
- 
-
- E.g. we wish to allocate a hash table to hold $n=2000$ character strings.
 - We do not mind examining an average of 3 elements in an unsuccessful search.
 - What hash table size should we choose? ($m=701$)
 - It is a prime near $2000/3$ but not near any power of 2.
 - Treating each key x as an integer, with the hash function as $h(x) = x \bmod 701$



- Multiplicative:
 1. Choose A where $0 < A < 1$
 2. Multiply key k by A
 3. Extract the fractional part of $k \cdot A$
 4. Multiply the fractional part by the number of slots m
 5. Take the floor of the result
- Pro: Value of m is not critical
- Con: slower than division
 - advocated by D. Knuth in TAOCP vol. III.



- Knuth suggests $A \approx (\sqrt{5}-1)/2 = 0.6180339887\dots$
- Typically choose $m=2^p$ (a power of 2) because multiplying by is easy to calculate.
- Suppose the word size of the machine is w bits and k fits into a single word.
- The integer part of $k \cdot A$ fits into a single word. If we shift left $k \cdot A$ a single word (equal to $\times 2^w$), the result is a $2w$ -bit value.
- The p highest-order bit of the lower w -bit half of the product forms the desired hash value.



- E.g. we have $k=123456$, $p=14$, $m=2^{14}=16384$, $w=32$.
- $A \approx (\sqrt{5}-1)/2$, $s=A \times 2^w = A \times 2^{32} = 2654435768$.
- $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$.
- The lower w -bit half of the product = 17612864.
- The $p=14$ highest-order bits of the lower w -bit half yields the value $h(k)=67$.



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 $123+203+241+112+20=699$

- 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 remaining digits as hash address.
- Distribution of keys may not be known in advance



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 $2^{32}/b$.
- 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, ...
- 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;
}
```

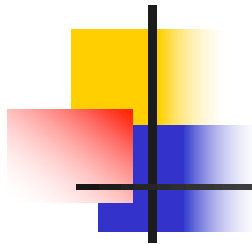
```
int hash(char *key)
{
    return (transform(key)
           % TABLE_SIZE);
}
```

```
int transform(char *key)
{
    int number=0;
    while (*key) {
        number += *key++;
        if (*key) number += ((int) *key++) << 8;
    }
    return number;
}
```



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



Example

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

$b=26, s=2, n=10, \alpha=10/52=0.19, f(x)=\text{the first char of } x$



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. 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

- Compute hash function $H(k)$
- If occupied, probe $H(k) + 1, H(k) + 2, \dots$
- divisor = b (number of buckets) = 17.
- Home bucket = $\text{key} \% 17$.

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



Performance Of Linear Probing

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

- Pro: Easy to implement
- Con: Primary clustering
 - Long runs of occupied slots build up. An empty slot preceded by i full slots gets filled next with probability $(i+1)/m$
- Worst case time is $\Theta(n)$. This happens when all pairs are in the same cluster



Problem of Linear Probing

- Identifiers tend to cluster together
- Adjacent cluster tend to coalesce
- Increase the search time



Quadratic Probing

- Compute $H(k)$
- If occupied, probe $H(k) + 1^2$, $H(k) + 2^2$, $H(k) + 3^2$, ...
- Secondary clustering
 - If the initial hash value is the same for two different keys, their probe sequences are the same
 - Could be better than primary clustering though



Random Probing

- Random Probing works incorporating with random numbers.
 - $H(x) := (H'(x) + S[i]) \% b$
 - $S[i]$ is a table with size $b-1$
 - $S[i]$ is a random permutation of integers $1, 2, \dots, b-1$.



Rehashing

- **Rehashing:** Try H_1, H_2, \dots, H_m in sequence if collision occurs. (H_i is a hash function.)



Rehashing (cont'd)

- **Double hashing** is one of the best methods for dealing with collisions.
 - If the slot is full, then a second hash function is calculated and combined with the first hash function.
 - $H(k, i) = (H_1(k) + i H_2(k)) \% m$
 - Probe sequence is:
 - $H_1(k) \% m$
 - $(H_1(k) + 1H_2(k)) \% m$
 - $(H_1(k) + 2H_2(k)) \% m$
 - ...

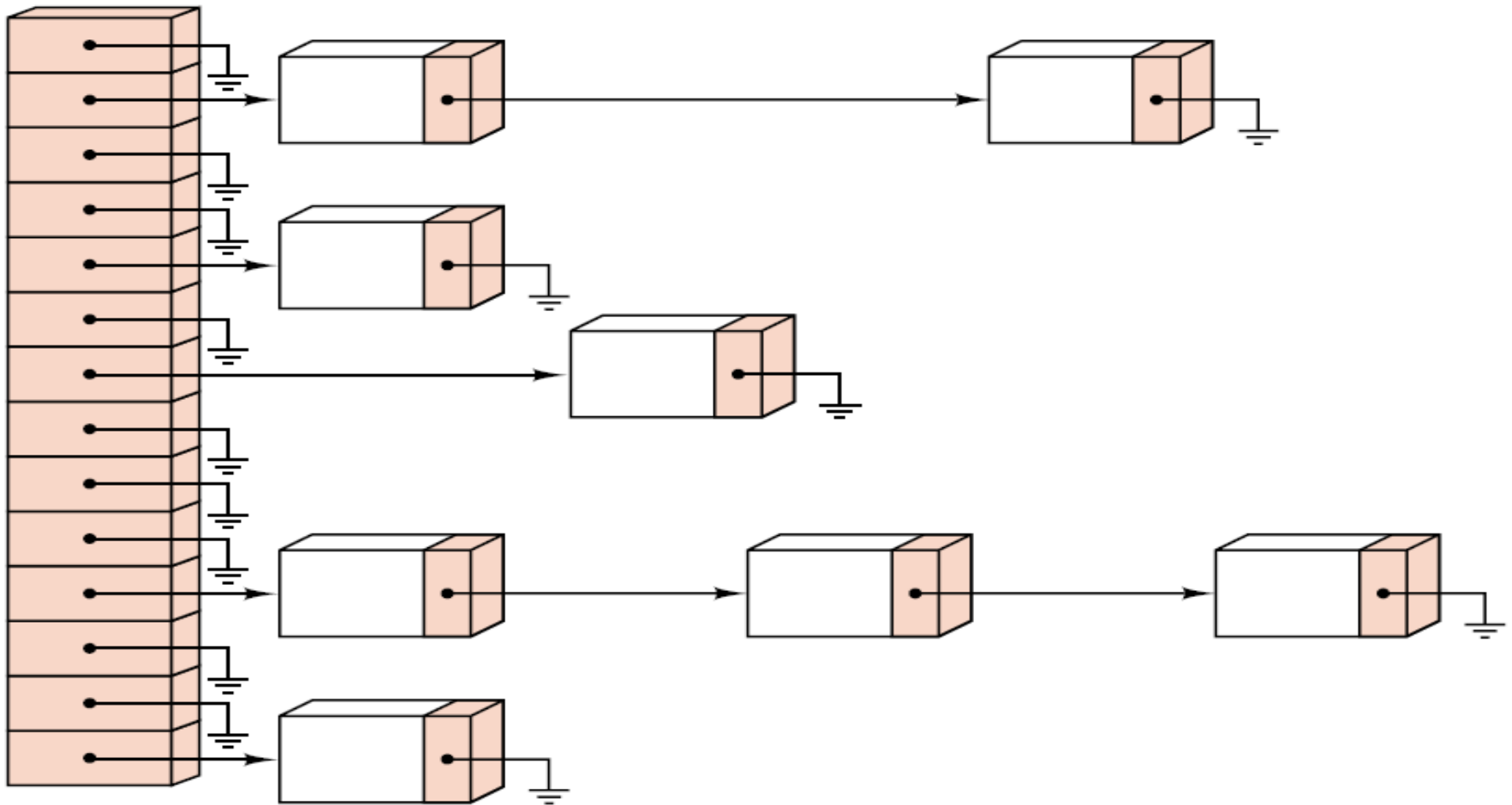


Data Structure for Chaining

```
#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];
```

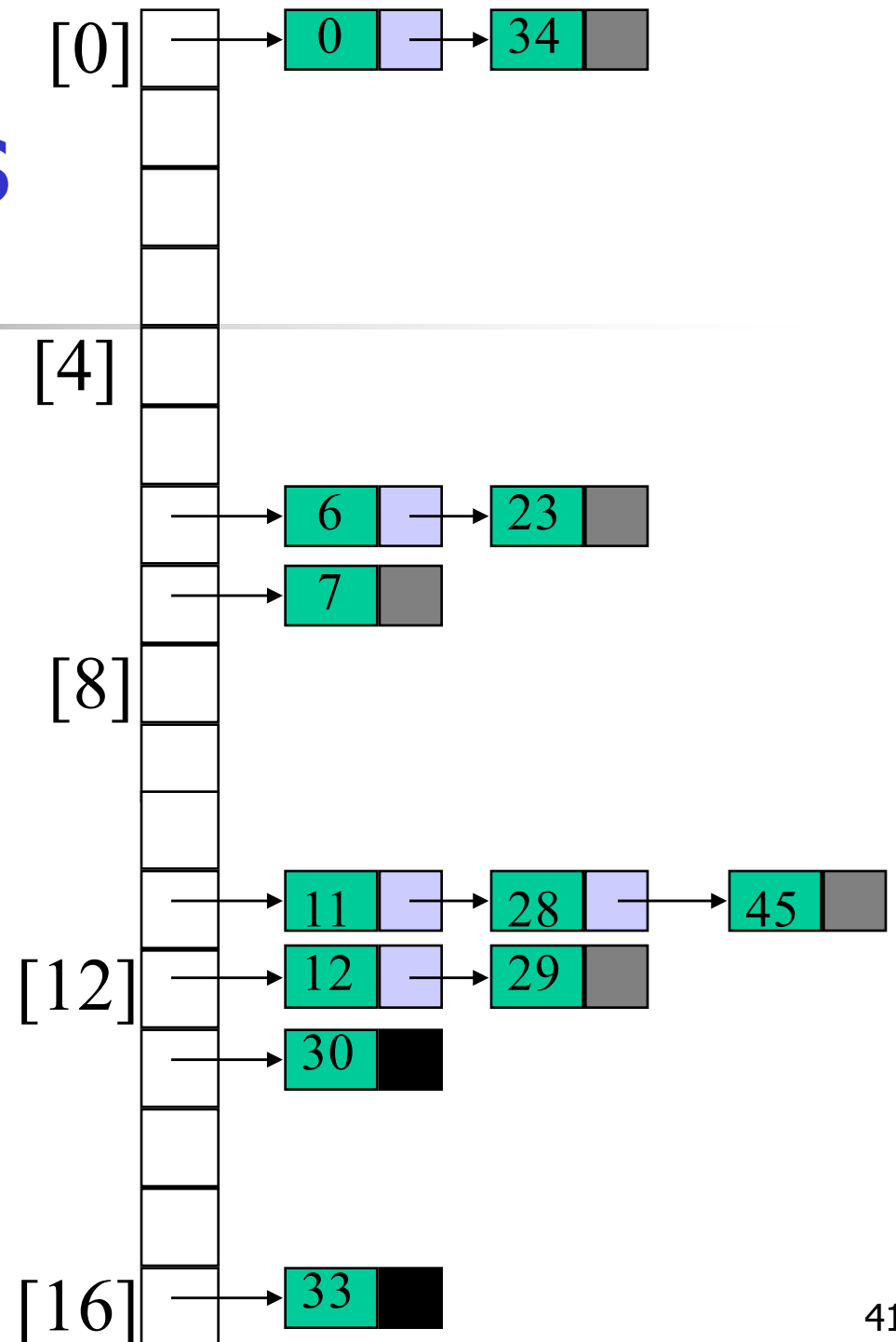
The idea of **Chaining** is to combine the linked list and hash table to solve the overflow problem.

Figure of Chaining



Sorted Chains

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





Conclusion

- **Linear probing** has the best cache performance but is most sensitive to clustering
- **Double hashing** has poorer cache performance but exhibits virtually no clustering
- **Quadratic probing** falls in between the two methods above

Dynamic hashing: Extendible hashing



- Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R., "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems* **4** (3): 315–344, September, 1979.
- For a good overview, read:
http://en.wikipedia.org/wiki/Extendible_hashing