Data Structures

Chapter 8 Hashing
(Concentrating on Static Hashing)

Chapter 8 Hashing: Outline

- The Symbol Table Abstract Data Type
- Static Hashing
  - Hash Tables
  - Hashing Functions
    - Mid-square
    - Division
    - Folding
    - Digit Analysis
  - Overflow Handling
    - Linear Open Addressing, Quadratic probing, Rehashing
    - Chaining

The Symbol Table (1/3)

- In computer science, we generally use the term *symbol table* rather than dictionary
- We define the symbol table as a *set of name-attribute pairs*.
  - Example: In a symbol table for a compiler
    - the name is an identifier
    - the attributes might include an initial value
    - a list of lines that use the identifier.

The Symbol Table (2/3)

- **Operations** on symbol table:
  - Determine if a particular name is in the table
  - Retrieve/modify the attributes of that name
  - Insert/delete a name and its attributes

The Symbol Table (2/3)

- Implementations
  - Binary search tree: the complexity is O(*n*)
  - Some other binary trees (chapter 10): O(log *n*).
  - *Hashing*
    - A technique for search, insert, and delete operations that has very good expected performance.

Search Techniques

- Search tree methods
  - Identifier comparisons
- Hashing methods
  - Relies on a formula called the hash function.
- Types of hashing
  - Static hashing
  - Dynamic hashing

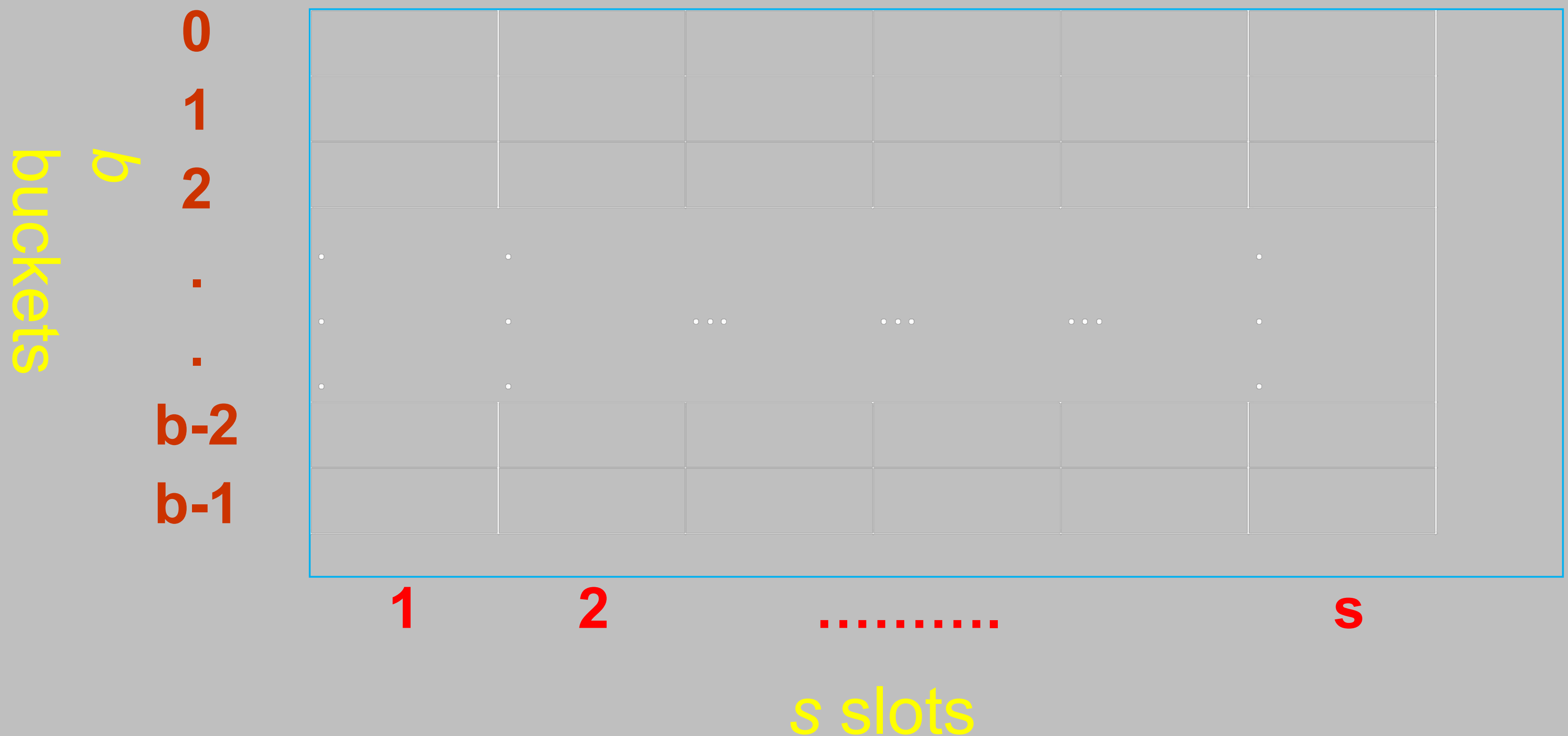
Hash Tables (1/6)

- In static hashing, we store the identifiers in a fixed size table called a *hash table*
- Arithmetic function, *f*
  - To determine the address of an identifier, *x*, in the table
  - *f*(*x*) gives the hash, or home address, of *x* in the table

Hash Tables (1/6)

- Hash table, *ht*
  - Stored in sequential memory locations that are partitioned into ***b*** buckets, ***ht*[0], …, *ht*[*b*-1]**.
  - Each bucket has *s* slots

Hash Tables (2/6)



Hash Tables (3/6)

- The ***identifier density*** of a hash table is the ratio ***n/T***
  - ***n*** is the number of identifiers in the table
  - ***T*** is possible identifiers
- 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

Hash Tables (4/6)

- Two identifiers, i_1 and i_2 are **synonyms** with respect to f **if $f(i_1) = f(i_2)$**

Hash Tables (4/6)

- An overflow occurs when we hash a new identifier into a full bucket
- A collision occurs when we hash two non-identical identifiers into the same bucket.
- When the bucket size is 1, collisions and overflows occur simultaneously.

Hash Tables (5/6)

- ***b* = 26 buckets** and ***s* = 2 slots**. Distinct identifiers *n* = 10
- hash function, *f*(*x*), as the first character of *x and associate* the letters, *a*-*z*, with the numbers, 0-25,

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

Synonyms

Synonyms

Synonyms

overflow: clock, ctime

Hash Tables (6/6)

- **The time required** to enter, delete, or search for identifiers does not depend on the number of identifiers *n* in use; it is ***O*(1)**.

Hash Tables (6/6)

- Hash function requirements/challenge:
  - Easy to compute and produces few collisions.
  - Unfortunately, since the ration b/T is usually small, we cannot avoid collisions altogether.
    => Overload handling mechanisms are needed

Hashing Functions (1/8)

- Hashing functions should be **unbiased.**
  - That is, if we randomly choose an identifier, *x*, from the identifier space, the probability that **$f(x) = i$ is $1/b$ for all buckets *i*.**
  - We call a hash function that satisfies unbiased property a ***uniform hash function*.**
    Mid-square, Division, Folding, Digit Analysis

Hashing Functions (2/8)

- **Mid-square** $f_m(x)=middle(x^2)$:
  - We compute f_m by squaring the identifier and then using an appropriate number of bits from the middle of the square to obtain the bucket address.
  - The number of bits used to obtain the bucket address depends on the table size. If we use r bits, the range of the value is 2^r.

Hashing Functions (2/8)

- **Mid-square** $f_m(x)=middle(x^2)$:
  - Since the middle bits of the square usually depend upon all the characters in an identifier, there is high probability that different identifiers will produce different hash addresses, i.e., unbiased.

Hashing Functions (3/8)

- **Division** $f_D(x) = x \% M$:
  - Using the modulus (%) operator.
  - We divide the identifier x by some number M and use the remainder as the hash address for x.
    - This gives bucket addresses that range from 0 to M - 1, where M = that table size.
  - The choice of M is critical.
    - If M is divisible by 2, then odd keys to odd buckets and even keys to even buckets. (biased!!)

Hashing Functions (4/8)

- Biased Example: $X = x_1 x_2$ and $Y = x_2 x_1$

  Internal binary representation: x_1 --> $C(x_1)$ and x_2 --> $C(x_2)$

  Each character is represented by six bits

  X: $C(x_1) * 2^6 + C(x_2)$,

  Y: $C(x_2) * 2^6 + C(x_1)$

$(f_D(X) - f_D(Y))$ % M (where M is a prime number)

$= (C(x_1) * 2^6 \% M + C(x_2) \% M - C(x_2) * 2^6 \% M - C(x_1) \% M) \% M$

$M = 3$, $2^6 = 64$

$(64 \% 3 * C(x_1) \% 3 + C(x_2) \% 3 - 64 \% 3 * C(x_2) \% 3 - C(x_1) \% 3) \% 3$

$= C(x_1) \% 3 + C(x_2) \% 3 - C(x_2) \% 3 - C(x_1) \% 3 = 0 \% 3$… **biased !!!**

Hashing Functions (5/8)

- **Folding**
  - Partition identifier ***x* =12320324111220** into several parts
  - All parts except for the last one have the same length
  - Add the parts together to obtain the hash address

Hashing Functions (5/8)

- Two possibilities (divide x into several parts)
  - ***Shift folding***:
    Shift all parts except for the last one, so that the least significant bit of each part lines up with corresponding bit of the last part.
    - $x_1=123$, $x_2=203$, $x_3=241$, $x_4=112$, $x_5=20$, address=699

Hashing Functions (6/8)

- Folding example:

123	203	241	112	20
P1	P2	P3	P4	P5

shift folding

123
203
241
112
20

699

Hashing Functions (5/8)

- Two possibilities (divide x into several parts)
  - ***Folding at the boundaries***:
    reverses every other partition before adding
    - **x_1=123, x_2=203, x_3=241, x_4=112, x_5=20,**
    - x_1=123, x_2=302, x_3=241, x_4=211, x_5=20, address=897

Hashing Functions (6/8)

- Folding example:

123	203	241	112	20
P1	P2	P3	P4	P5
123	**302**	**241**	**211**	**20**

folding at the boundaries

MSD ---> LSD

LSD <--- MSD

Hashing Functions (7/8)

- Digit Analysis
  - Used with static files
    - A *static files* is one in which all the identifiers are known in advance.
  - Using this method,
    - First, transform the identifiers into numbers using some radix, ***r***.
    - Second, examine the digits of each identifier, deleting those digits that have the most skewed distribution.
    - We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hash table.

Hashing Functions (8/8)

- Digital Analysis example:

X_1 : d_{11} d_{12} … d_{1n}

X_2 : d_{21} d_{22} … d_{2n}

…

X_m : d_{m1} d_{m2} … d_{mn}

  - Select 3 digits from *n*
    - Criterion: Delete the digits having the most skewed distributions

Hashing Functions (8/8)

- The one most suitable for general purpose applications is the division method with a divisor, *M*, such that *M* has no prime factors less than 20.

Overflow Handling (1/8)

- Open addressing
  - Linear probing, quadratic probing, rehashing
- Chaining

Overflow Handling (1/8)

- Linear open addressing (Linear probing)
  - Compute $f(x)$ for identifier x
  - Examine the buckets:

$$ht[(f(x)+j)\%TABLE\_SIZE],\ 0 \le j \le TABLE\_SIZE$$

Overflow Handling (2/8)

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

Identifier	Additive Transformation	x	Hash
for	102 + 111 + 114	327	2
do	100 + 111	211	3
while	119 + 104 + 105 + 108 + 101	537	4
if	105 + 102	207	12
else	101 + 108 + 115 + 101	425	9
function	102 + 117 + 110 + 99 + 116 + 105 + 111 + 110	870	12

Overflow Handling (3/8)

- Problem of Linear Probing
  - Identifiers tend to cluster together
  - Adjacent cluster tend to coalesce
    - Increase the search time

Overflow Handling (3/8)

- Example:

Enter sequence: **acos**, **atoi**, **char**, **define**, **exp**, **ceil**, **cos**, **float**, **atol**, **floor**, **ctime**

# of key comparisons=41/11=3.72

Enter :

bucket	x	buckets searched
→ 0		
→ 1		
→ 2		
→ 3		
→ 4		
→ 5		
→ 6		
→ 7		
→ 8		
→ 9		
→ 10		
...		
25		

Overflow Handling (4/8)

- Alternative techniques to improve open addressing approach:
  - *Rehashing*
  - *Quadratic probing*

Overflow Handling (4/8)

Rehashing

- Try h_1, h_2, …, h_m in sequence if collision occurs
- disadvantage
  - comparison of identifiers with different hash values

Overflow Handling (5/8)

- Quadratic Probing
  - Linear probing searches buckets $(f(x)+i)\%b$
  - Quadratic probing uses a quadratic function of i as the increment
  - Examine buckets $f(x)$, $(f(x)+i^2)\%b$, $(f(x)-i^2)\%b$, for $1<=i<=(b-1)/2$
  - When b is a prime number of the form $4j+3$, j is an integer, the quadratic search examines every bucket in the table

Prime	j	Prime	j
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

Overflow Handling (6/8)

- Chaining
  - Linear probing and its variations perform poorly because inserting an identifier requires the comparison of identifiers with different hash values.
  - In this approach we maintained a list of synonyms for each bucket.
  - To insert a new element
    - Compute the hash address $f(x)$
    - Examine the identifiers in the list for $f(x)$.
  - Since we would not know the sizes of the lists in advance, we should maintain them as lined chains

Overflow Handling (7/8)

- Results of Hash Chaining

acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime

f (*x*)=first character of x

[0] –> **acos** –> **atoi** –> **atol**
[1] –> *NULL*
[2] –> **char** –> **ceil** –> **cos** –> **ctime**
[3] –> **define**
[4] –> **exp**
[5] –> **float** –> **floor**
[6] –> *NULL*
. . .
[25] –> *NULL*

# of key comparisons=21/11=1.91

Overflow Handling (8/8)

- Comparison:
  - In Figure 8.7, The values in each column give the average number of bucket accesses made in searching eight different table with 33,575, 24,050, 4909, 3072, 2241, 930, 762, and 500 identifiers each.
  - Chaining performs better than linear open addressing.
  - We can see that division is generally superior

$\alpha = \frac{n}{b}$	.50		.75		.90		.95	
Hash Function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid 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 fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

Average number of bucket accesses per identifier retrieved

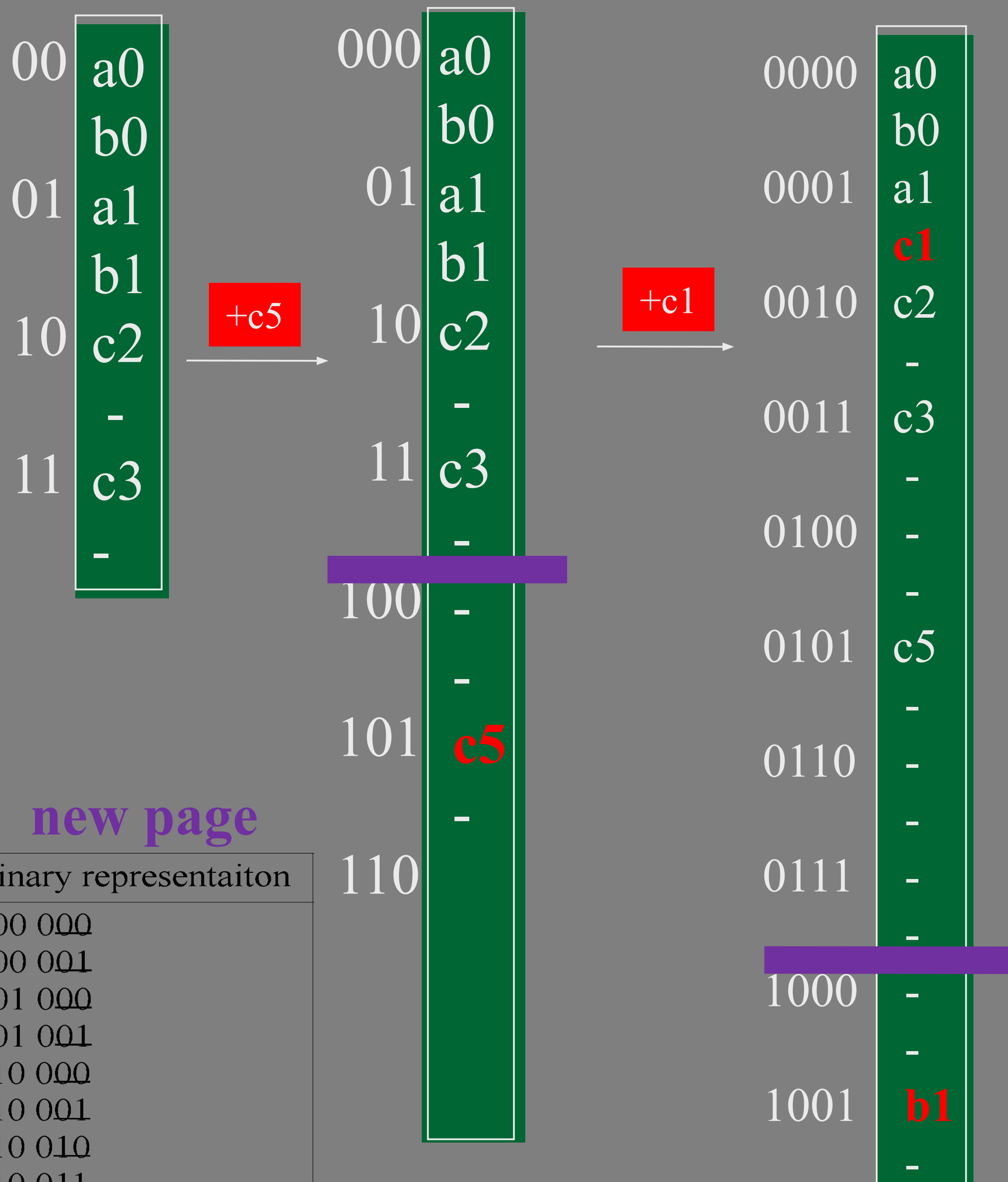
Dynamic Hashing (extensible hashing)

- dynamically increasing and decreasing file size
- concepts
  - file: a collection of records
  - record: a key + data, stored in pages (buckets)
  - space utilization

$$\frac{NumberOfRecord}{NumberOfPages * PageCapacity}$$

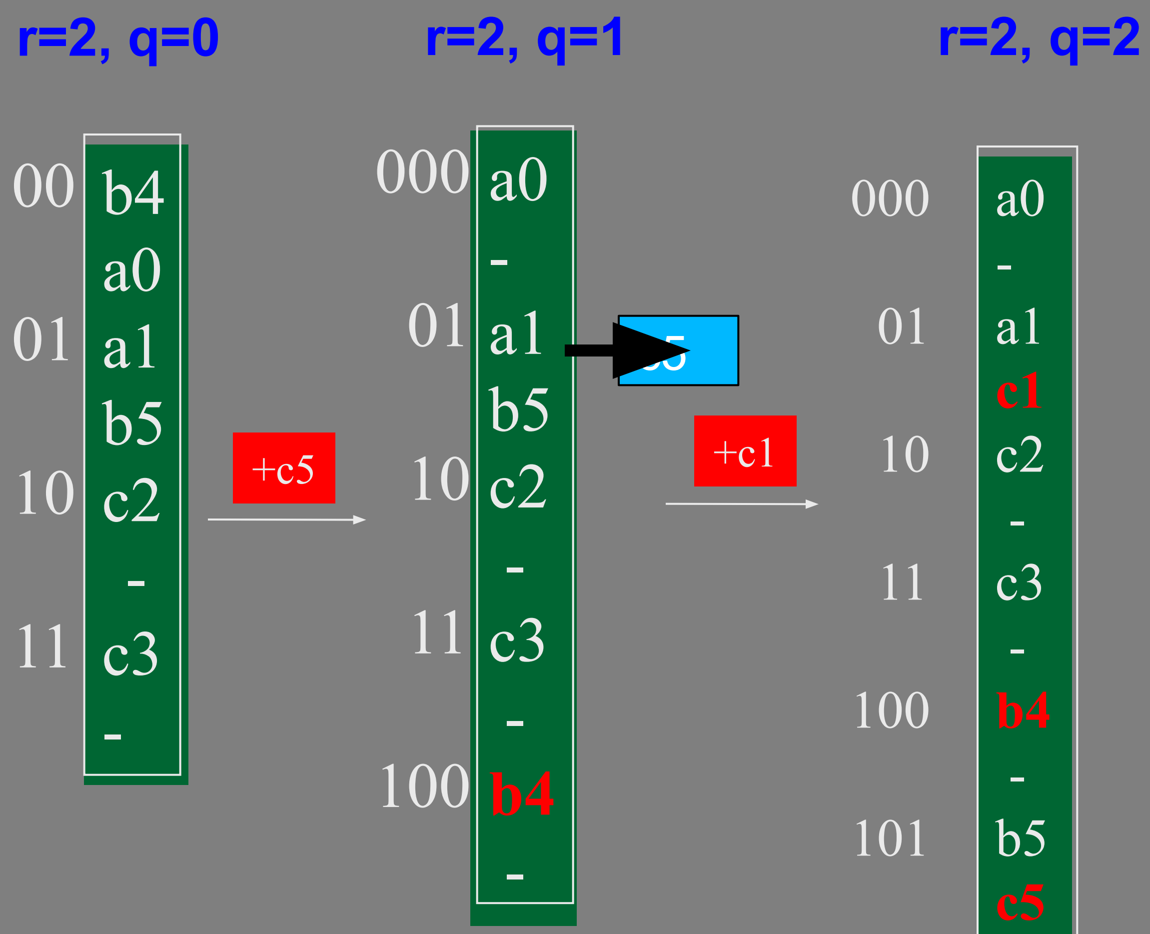
Identifiers	Binary representaiton
a0	100 000
a1	100 001
b0	101 000
b1	101 001
c0	110 000
c1	110 001
c2	110 010
c3	110 011

Dynamic Hashing Using Directories



Identifiers	Binary representaiton
a0	100 000
a1	100 001
b0	101 000
b1	101 001
c0	110 000
c1	110 001
c2	110 010
c3	110 011

Directoryless Dynamic Hashing



```
if ( hash(key,r) < q)
    page = hash(key, r+1);
else
    page = hash(key, r);
```

if needed, then follow overflow pointers;