DATA STRUCTURE

Chapter 8: Hashing

SYMBOL TABLE

- Symbol table is used widely in many applications.
  - dictionary is a kind of symbol table
  - data dictionary is database management
- In general, the following operations are performed on a symbol table
  - determine if a particular name is in the table
  - retrieve the attribute of that name
  - modify the attributes of that name
  - insert a new name and its attributes
  - delete a name and its attributes

SYMBOL TABLE (CONT'D)

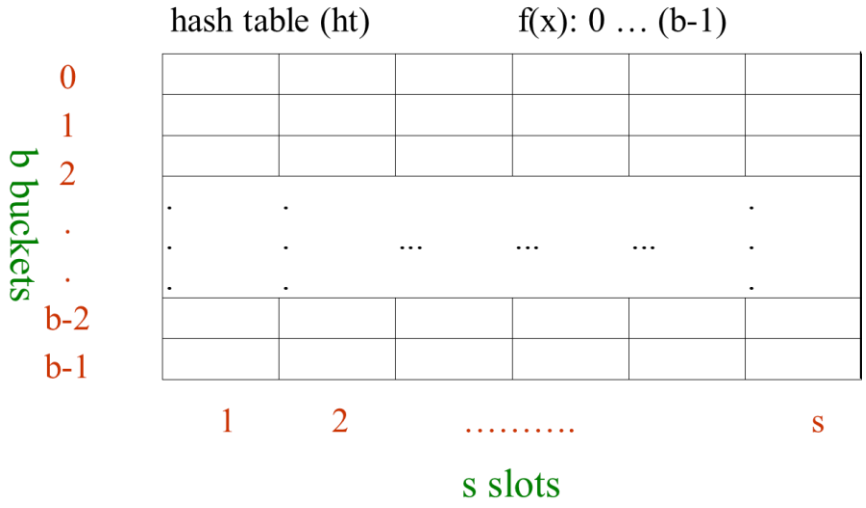
- Popular operations on a symbol table include search, insertion, and deletion
- A binary search tree could be used to represent a symbol table.
  - The worse-case complexities for the operations are O(n).
- Hashing: insertions, deletions & finds in constant time.
- General data structure of hashing
  - array of fixed size(TableSize) containing the keys
  - hash function

HASHING

- Static hashing
  - Perfect hashing without collision
  - Hashing with collision
    - Chaining
    - Open addressing
      - Linear probing
      - Quadratic probing
      - Rehashing
- Dynamic hashing
  - Extensible hashing
  - Linear 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*.



HASH TABLES

- 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 $a = n/(sb)$.

HASH TABLES (CONT'D)

- Two identifiers, *I*1, and *I*2, 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 8.1

	Slot 1	Slot 2
0	A	A2
1		
2		
3	D	
4		
5		
6	GA	G
…	…	…
25		

Assume there are 10 distinct identifiers GA, A,G,L,A2,A1,A3,A4 and E

Large number of collisions and overflows!

If no overflow occur, the time required for hashing depends only on the time required to compute the hash function h.

HASH FUNCTION

- Requirements
  - Simple to compute
  - Minimize the number of collisions
- Dependent upon all the characters in the identifiers
- Uniform hash function
  - If there are b buckets, we hope to have $h(x) = i$ with the probability being (1/b)
- Commonly used hash function
  - Mid-square, division (mod), folding, digit analysis

MID-SQUARE

- Mid-Square function
  - h_m 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.
- Table size is a power of two

DIVISION

- 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.

DIVISION (CONT'D)

- If identifiers are stored right-justified with leading zeros and $M = 2^i, i \leq 6$, the identifiers A1, B1, C1, etc., all have the same bucket.
- If a division function h_D is used as the hash function, the table size should not be a power of two.
  - Since programmers have a tendency to use many variables with the same suffix → too many collisions
- 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.

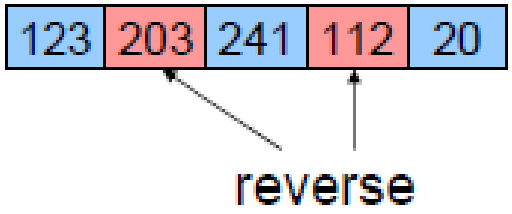
FOLDING

- 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.

EXAMPLE 8.2

- 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:
  - h(x) = 123 + 302 + 241 + 211 + 20 = 897

OVERFLOW HANDLING

- There are two ways to handle overflow:
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Rehashing
  - Chaining

OPEN ADDRESSING

- Assumes the hash table is an array
- The hash table is initialized so that each slot contains the null identifier.
- When a new identifier is hashed into a full bucket, find the closest unfilled bucket.
  - linear probing or linear open addressing

LINEAR PROBING

- Assume 26-bucket table with one slot per bucket and the following identifiers: GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E. Let the hash function h(x) = first character of x.
- When entering G, G collides with GA and is entered at ht[7] instead of ht[6].

0	A
1	A2
2	A1
3	D
4	A3
5	A4
6	GA
7	G
8	ZA
9	E
…	…
25	Z

LINEAR PROBING (CONT'D)

- When linear open address is used to handle overflows, a hash table search for identifier x proceeds as follows:
  - compute $h(x)$
  - examine identifiers at positions $ht[h(x)], ht[h(x)+1], \ldots, ht[h(x)+j]$, in this order until one of the following condition happens:
    - $ht[h(x)+j] = x$; in this case x is found
    - $ht[h(x)+j]$ is null; x is not in the table
    - We return to the starting position $h(x)$; the table is full and x is not in the table

LINEAR PROBING (CONT'D)

- When linear probing is used to resolve overflows, identifiers tend to cluster together.
  - This increases search time.
  - e.g., to find ZA, you need to examine $ht[25], ht[0], \ldots, ht[8]$ (total of 10 comparisons).
- The number of comparisons to look up an identifier is approximated $(2 - \alpha)/(2 - 2\alpha)$, where α is the load density

0	A
1	A2
2	A1
3	D
4	A3
5	A4
6	GA
7	G
8	ZA
9	E
…	…
25	Z

QUADRATIC PROBING

- One of the problems of linear open addressing is that it tends to create clusters of identifiers.
- These clusters tend to merge as more identifiers are entered, leading to bigger clusters.
  - It is difficult to find an unused bucket.
- A quadratic probing scheme improves the growth of clusters. A quadratic function of *i* is used as the increment when searching through buckets.
- Perform search by examining bucket $h(x)$, $(h(x) \pm i^2)\%b$, $for\ 1 \leq i \leq (b-1)/2$.

REHASHING

- Another way to control the growth of clusters is to use a series of hash functions $h_1, h_2, \ldots, hm$. This is called rehashing.
- Buckets $h_i(x), 1 \leq i \leq m$ are examined in that order.
- Double hashing:
  - If $h(x)$ is occupied, then we iteratively try buckets $h(x) + j * h'(x)\ for\ j = 1, 2, \ldots$

SUMMARY OF OPEN ADDRESSING

- If collisions, alternative cells are tried until an empty cell is found.
- $H_i(K) = (h(K) + F(i)) \,\%\, TS, F(0) = 0$
- *F*: collision resolution strategy
  - Linear probing: *F* is a linear function
  - Quadratic probing: *F* is a quadratic function
  - Double hashing: *F* is a second hash function

EXAMPLE - LINEAR PROBING

- $H_i(K) = (h(K) + F(i))\%TS,\ F(0) = 0$, F is linear
- Example: $F(i) = i, \Rightarrow H_i(K) = (h(K) + i)\%10$

Content	49	58	69						18	89
Index	0	1	2	3	4	5	6	7	8	9

- Primary clustering: effect that any key hashes into the cluster will require several attempts to resolve collisions & be added to the cluster

EXAMPLE - QUADRATIC PROBING

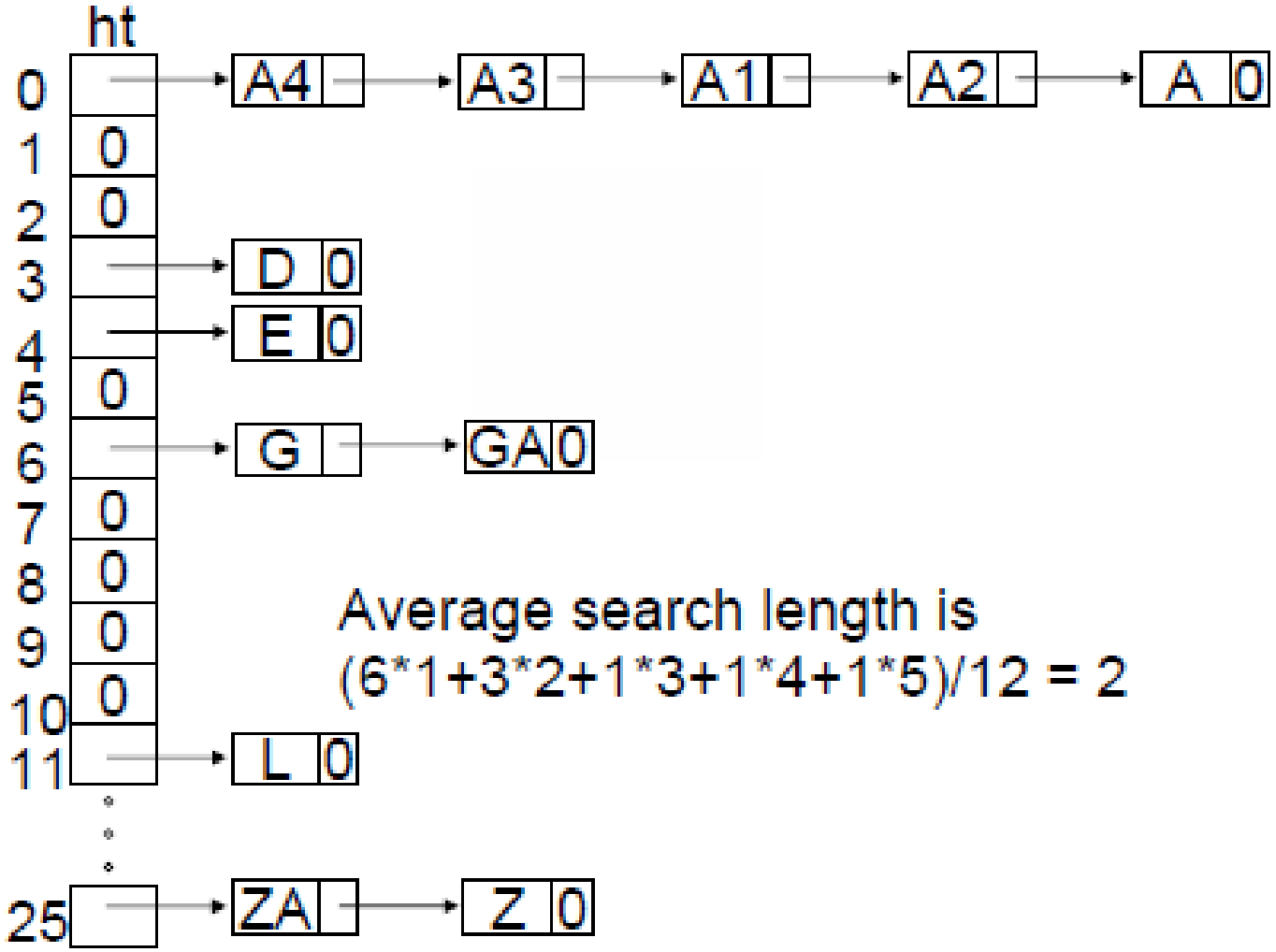
- $H_i(K) = (h(K) + F(i))\%TS, F(0) = 0$, F is quadratic
- Example: $F(i) = i^2$, $H_{i(K)} = (h(K) + i^2)\%10$

Content	49		58	69					18	89
Index	0	1	2	3	4	5	6	7	8	9

CHAINING

- Linear probing performs poorly because the search for an identifier involves comparisons with identifiers that have different hash values.
  - e.g., search of ZA involves comparisons with the buckets ht[0] – ht[7] which are not possible of colliding with ZA.
- Unnecessary comparisons can be avoided if all the synonyms are put in the same list, where one list per bucket.
- Each chain has a head node. Head nodes are stored sequentially.

HASH CHAIN EXAMPLE



HASH FUNCTIONS

- Theoretically, the performance of a hash table depends only on the method used to handle overflows and is independent of the hash function as long as an uniform hash function is used.

THEORETICAL EVALUATION OF OVERFLOW TECHNIQUES

- In general, hashing provides pretty good performance over conventional techniques such as balanced tree.
- The expected number of comparisons can be shown to be 1+a/2
  - Theorem 8.1
- However, the worst-case performance of hashing can be O(n).

DYNAMIC HASHING

- Retain the fast retrieval time
  - Dynamically increasing and decreasing file size without penalty
- Assume a file F is a collection of records R. Each record has a key field K
  - Records are stored in pages or buckets whose capacity is p
- Dynamic hashing is to minimize access to pages

DYNAMIC HASHING USING DIRECTORIES

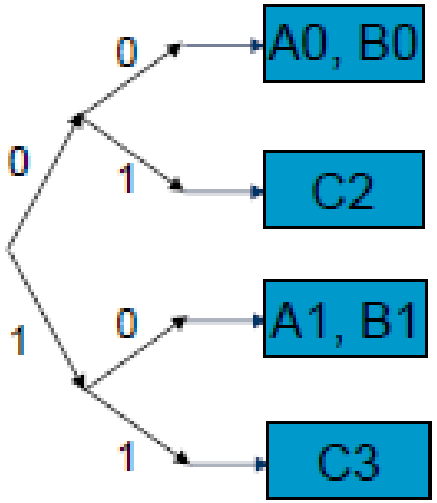
- Given a list of identifiers in the following:

Identifiers	Binary representation
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C0	110 000
C1	110 001
C2	110 010
C3	110 011
C5	110 101

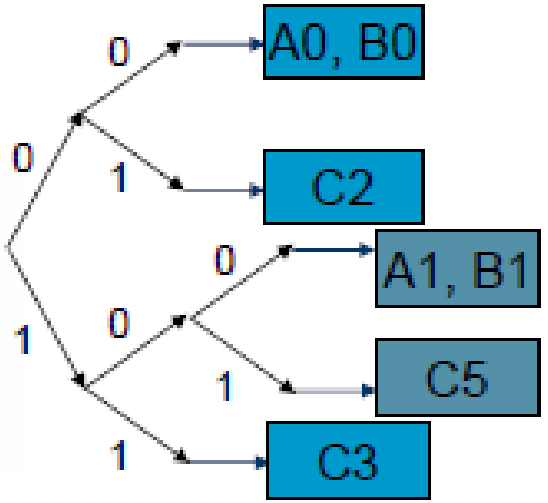
DYNAMIC HASHING USING DIRECTORIES (CONT'D)

- Now put these identifiers into a table of four pages. Each page can hold at most two identifiers, and each page is indexed by two-bit sequence 00, 01, 10, 11.
- Now place A0, B0, C2, A1, B1, and C3 in a binary tree, called Trie, which is branching based on the last significant bit at root. If the bit is 0, the upper branch is taken; otherwise, the lower branch is taken. Repeat this for next least significant bit for the next level.

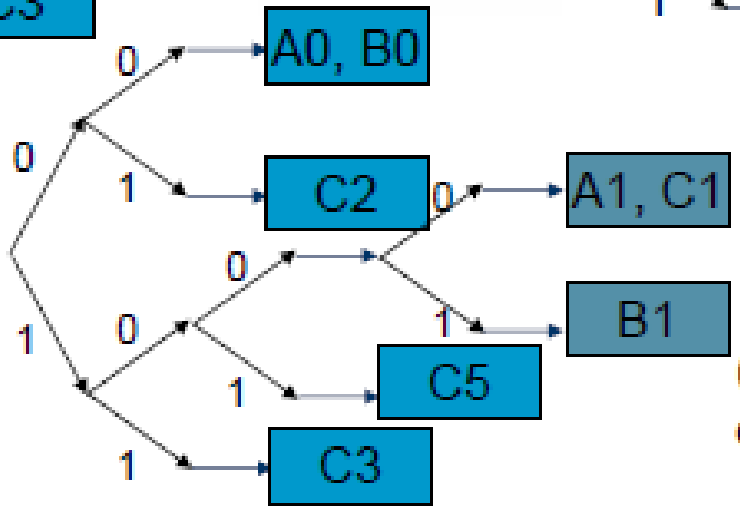
A TIE TO HOLD IDENTIFIERS



(a) two-level trie on four pages



(b) inserting C5 (110 101) with overflow



(c) inserting C1 with overflow

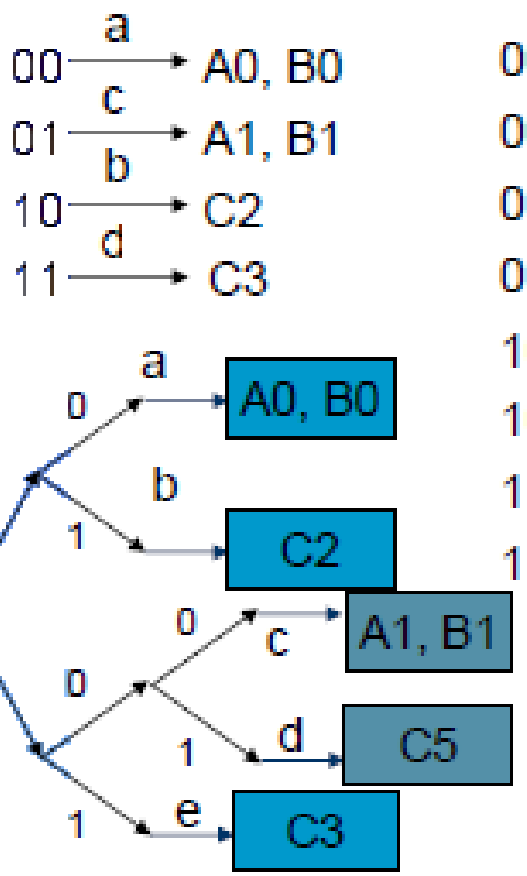
ISSUES OF TRIE REPRESENTATION

- From the example, we find that two major factors that affects the retrieval time.
- Access time for a page depends on the number of bits needed to distinguish the identifiers.
- If identifiers have a skewed distribution, the tree is also skewed.

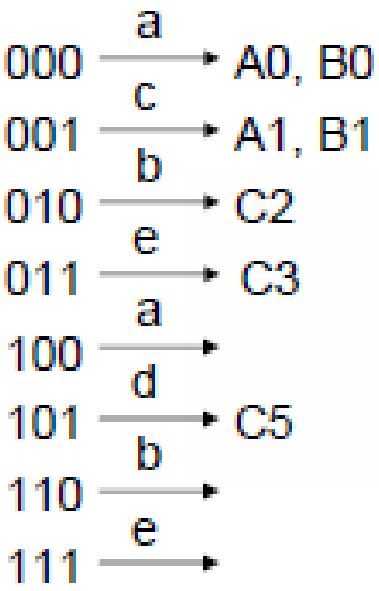
EXTENSIBLE HASHING

- Fagin et al. present a method, called *extensible hashing*, for solving the above issues.
  - A hash function is used to avoid skewed distribution. The function takes the key and produces a random set of binary digits.
  - To avoid long search down the trie, the trie is mapped to a directory, where a directory is a table of pointers.
  - If k bits are needed to distinguish the identifiers, the directory has 2k entries indexed 0, 1, …, 2k-1
  - Each entry contains a pointer to a page.

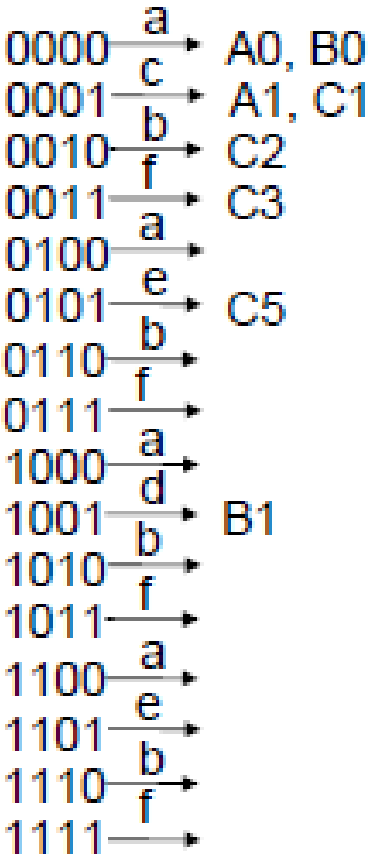
TRIE COLLAPSED INTO DIRECTORIES



(a) 2 bits



(b) 3 bits



(c) 4 bits

HASHING WITH DIRECTORY

- Using a directory to represent a trie allows table of identifiers to grow and shrink dynamically.
- Accessing any page only requires two steps:
  - First step: use the hash function to find the address of the directory entry.
  - Second step: retrieve the page associated with the address

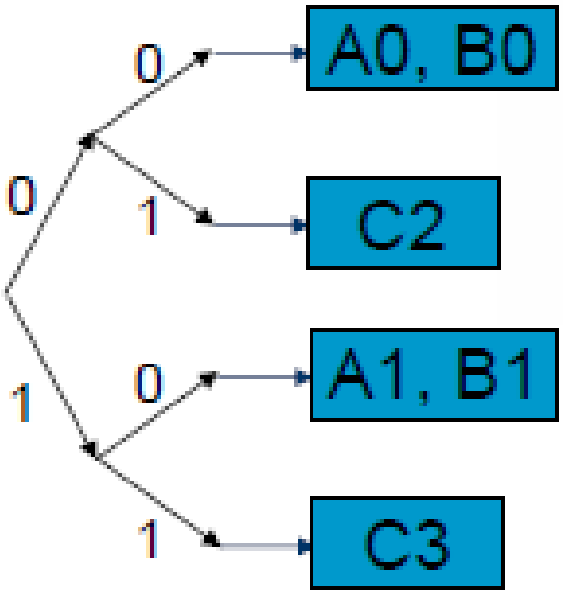
DIRECTORYLESS DYNAMIC HASHING

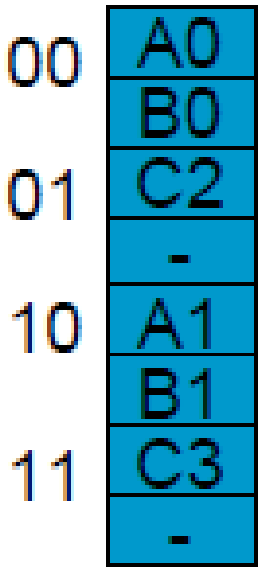
- Hashing with directory requires at least one level of indirection
- Directoryless hashing (or linear hashing) assume a continuous address space in the memory to hold all the records. Therefore, the directory is not needed.
  - Without indirection
- Thus, the hash function must produce the actual address of a page containing the key.

DIRECTORYLESS DYNAMIC HASHING (CONT'D)

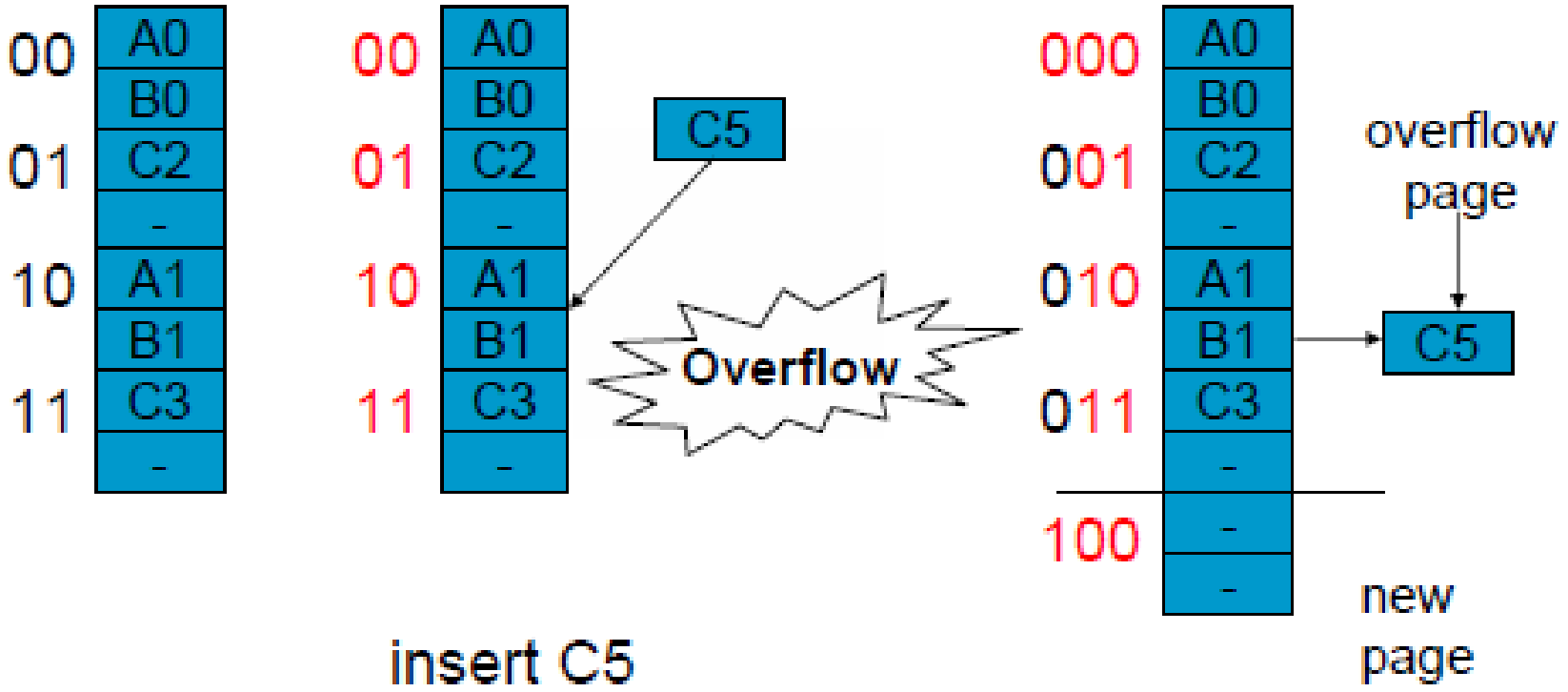
- Contrasting to the directory scheme in which a single page might be pointed at by several directory entries, in the directoryless scheme one unique page is assigned to every possible address.

A TRIE MAPPED TO DIRECTORYLESS, CONTINUOUS STORAGE

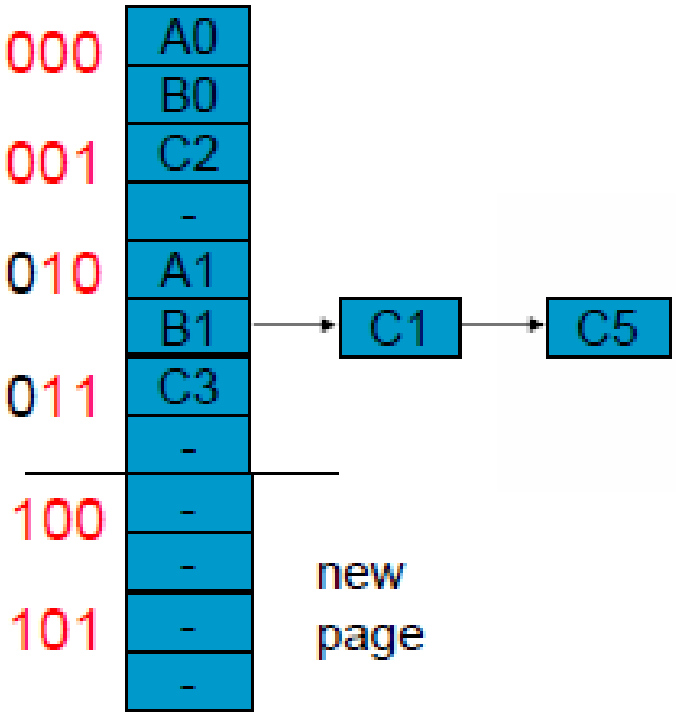




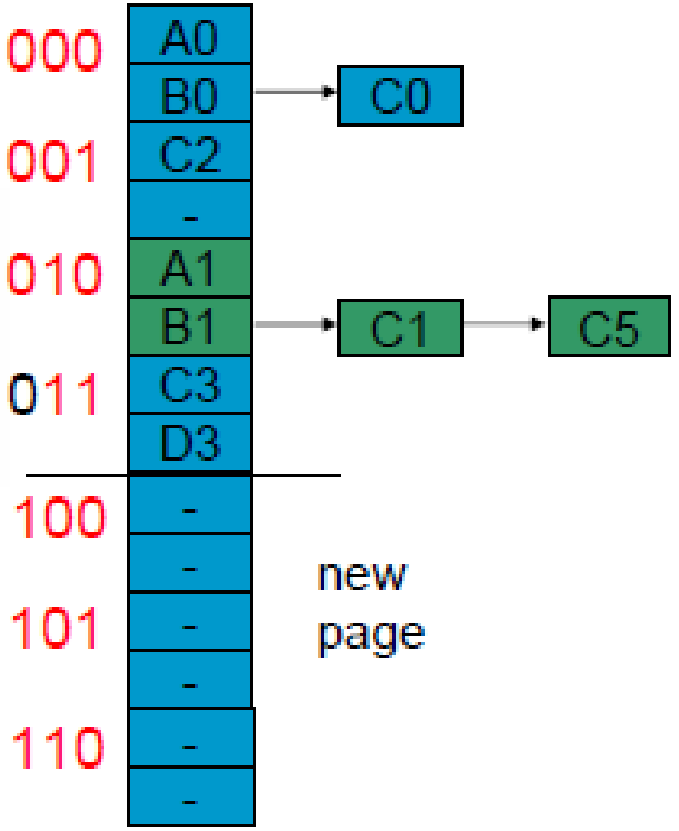
DIRECTORYLESS SCHEME OVERFLOW HANDLING



DIRECTORYLESS SCHEME OVERFLOW HANDLING (CONT'D)

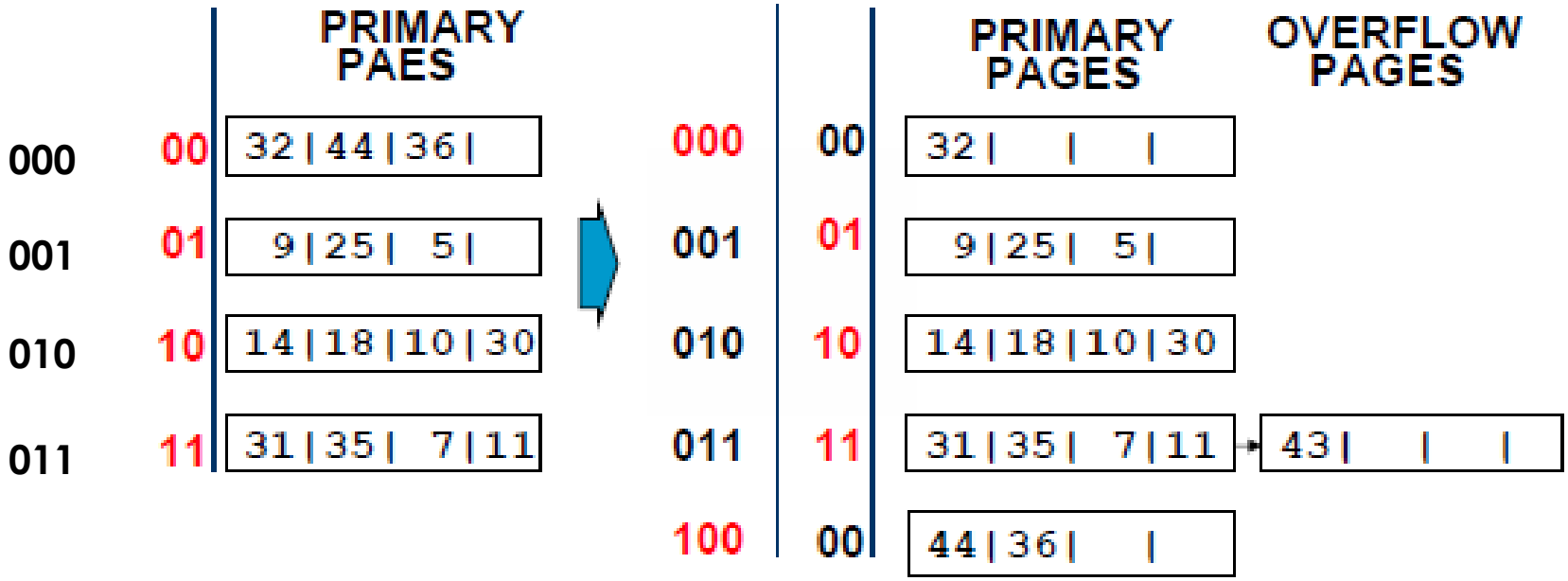


insert C1



insert C0

EXAMPLE OF LINEAR HASHING



insert 43→overflow→page 00 is split
→records in page 00 should be redistributed