CHAPTER 8 HASHING

8.1 The symbol table abstract data type

- Many example of *dictionaries* are found in many applications, including the spelling checker, the thesaurus.
- In computer science, we generally use the term *symbol table* rather than dictionary, when referring to the ADT.
- We define the symbol table as a *set of name-attribute pairs*.
  - Example: In a thesaurus, the name is a word, and the attribute is a list of synonyms for the word; in a symbol table for a compiler, the name is an identifier, and the attributes might include an initial value and a list of lines that use the identifier.

- Generally we would want to perform the following operations on symbol table:
  (1) determine if a particular name is in the table (search)
  (2) retrieve the attributes of that name
  (3) modify the attributes of that name
  (4) insert a new name and its attributes
  (5) delete a name and its attributes.
- To implement these operations, we could use the binary search tree which is introduced in chapter 5 and the complexity is $O(n)$, or some other binary trees which will be introduced in chapter 10 and the complexity are $O(\log n)$.
- In this chapter we examine a technique for search, insert, and delete operation that has very good expected performance. This technique is *hashing*.

– Comparisons vs. Hashing : Unlike search tree methods which rely on identifier comparisons to perform a search, hashing relies on a formula called the hash function.

– Two Types of Hashing : We divide our discussion of hashing into two part: *static hashing* and *dynamic hashing*.

8.2 Static hashing

- Hash tables
  - In static hashing, we store the identifiers in a fixed size table called a *hash table*.
    - We use an arithmetic function, *f*, to determine the address, or location, of an identifier, *x*, in the table. Thus, *f*(*x*) gives the hash, or home address, of *x* in the table.
    - The hash table *ht* is stored in sequential memory locations that are partitioned into *b* buckets, *ht*[0], …, *ht*[*b*-1]. Each bucket has *s* slots (i.e., the size of *b* is *s*).

- **Definition:**

  The *identifier density* of a hash table is the ratio n/T, where n is the number of identifiers of the table and T is the number of all distinct possible values . The *loading density* or *loading factor* of a hash table is $\alpha = n/(sb)$.

- Ex., 學生的身份証號碼的 $T=26*10^9$. Any reasonable application would never have this many identifiers.
- Two identifiers, i_1 and i_2 , are *synonyms* with respect to f if $f(i_1) = f(i_2)$.
- An *overflow* occurs when we hash a new identifier, i, into a full bucket.
- A *collision* occurs when we hash two distinct identifiers into the same bucket.
  - When the bucket size is 1, collisions and overflows occur simultaneously.

– Example 8.1:

- Consider the hash table *ht* with $b = 26$ buckets and $s = 2$. We have $n = 10$ distinct identifiers, each representing a C library function.
  - The loading factor, α, is 10/52 = 0.19.
  - The hash function must map each of the possible identifiers onto one of the numbers, 0-25. We can construct a fairly simple hash function by associating the letters, *a*-*z*, with the numbers, 0-25, respectively, and then defining the hash function, $f(x)$, as the first character of x.
  - Figure 8.1 shows the first 8 identifiers entered into the hash table.
  - Our choice of a hash function in Example 8.1 is not well suited for most applications since a large number of collisions and overflows is likely.
  - (Figure 8.1)

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

Figure 8.1: Hash table with 26 buckets and two slots per bucket

– 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).

- Hashing Functions
  - A hash function, *f*, transforms an identifier, *x*, into a bucket address in the hash table.
  - Ideally, we would like to choose a hash function that is both easy to compute and produces few collisions.
    - Unfortunately, since the ratio b/T is usually small, we cannot avoid collisions altogether. => Overload handling mechanisms are needed.
  - 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*.

– We will describe four types of uniform hash functions.
  - Mid-square
  - Division
  - Folding
  - Digit Analysis

– **Mid-square**
  - The *middle of square* hash function is frequently used in symbol table application.
  - We compute the function 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.
  - 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.
  - 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.

– Division

- We obtain a second simple hash function by using the modulus (%) operator.
- We divide the identifier x by some number *M* and use the remainder as the hash address for *x*. The hash function is:

$$f(x) = x \% M$$

  – This gives bucket addresses that range from 0 to *M* - 1, where *M* = table size.

- The choice of *M* is critical.
  – If *M* is divisible by 2, then odd keys are mapped to odd buckets and even keys are mapped to even buckets. (biased!!)
  – A good choice for *M* would be : *M* a prime number such that *M* does not divide $r^k \pm a$ for small *k* and *a*.

– Folding

- We partition the identifier *x* into several parts. All parts, except for the last one have the same length. We then add the parts together to obtain the hash address for *x*.
- There are two ways of carrying out this addition.
  - *Shift folding*: We 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. We then add the parts together to obtain $f(x)$.
    - » Ex: suppose that we have divided the identifier x into the following parts: $x_1 = 123$, $x_2 = 203$, $x_3 = 241$, $x_4 = 112$, and $x_5 = 20$. We would align x_1 through x_4 with x_5 and add. This gives us a hash address of 699.
  - *Folding at the boundaries*: reverses every other partition before adding.
    - » Ex suppose the identifier *x* is divided into the same partitions as in shift folding. We would reverse the second and forth partitions, that is $x_2 = 302$ and $x_4 = 211$, and add the partitions. This gives us a hash address of 897.

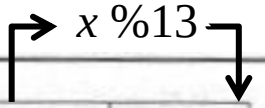
– **Digit Analysis**

- Digital analysis is used with static files.
- A *static files* is one in which all the identifiers are known in advance.
- Using this method,
  - We first transform the identifiers into numbers using some radix, *r*.
  - We then 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.

– Of these methods, 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**
  - There are two methods for detecting collisions and overflows in a static hash table. One is *linear open addressing* or *linear probing*, and the other is *chaining*.
  - Linear open addressing
    - If we insert an element into the hash table and the slot at the hash address is empty, we simply place the new element into this slot.
    - However, if the new element is hashed into a full bucket, we must find another bucket for it.
      - The simplest solution places the new element in the closest unfilled bucket. We refer to this method of resolving overflows as *linear probing* or *linear open addressing*.

- Example:



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

Figure 8.2: Additive transformation

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

Figure 8.3: Hash table with linear probing (13 buckets, 1 slot/bucket)

- Example: suppose we enter the C built-in functions acos, atoi, char, define, exp, ceil, cos, float, atol, floor, and ctime into a 26-bucket hash table in order. The hash function uses the first character in each function name.
- Our earlier example shows that when we use linear probing to resolve overflows, identifiers tend to cluster together. In addition, adjacent clusters tend to coalesce, thus increasing the search time.
- Example: suppose we enter the C built-in functions **acos**, **atoi**, **char**, **define**, **exp**, **ceil**, **cos**, **float**, **atol**, **floor**, and **ctime** into a 26-bucket hash table in order. The hash function uses the first character in each function name.
- (Figure 8.4)

bucket	x	buckets searched
0	**acos**	1
1	**atoi**	2
2	**char**	1
3	**define**	1
4	**exp**	1
5	**ceil**	4
6	**cos**	5
7	**float**	3
8	**atol**	9
9	**floor**	5
10	**ctime**	9
...		
25		

Figure 8.4: Hash table with linear probing (26 buckets, 1 slot per bucket)

– Chaining

- Linear probing and its variations perform poorly because inserting an identifier requires the comparison of identifiers with different hash values.
- We could have eliminated most of these comparisons if we had maintained a list of synonyms for each bucket.
- To insert a new element we would only have to compute the hash address $f(x)$ and 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.
- Example: Figure 8.6 shows the chained hash table corresponding to the linear table found in Figure 8.4.
- (Figure 8.6)

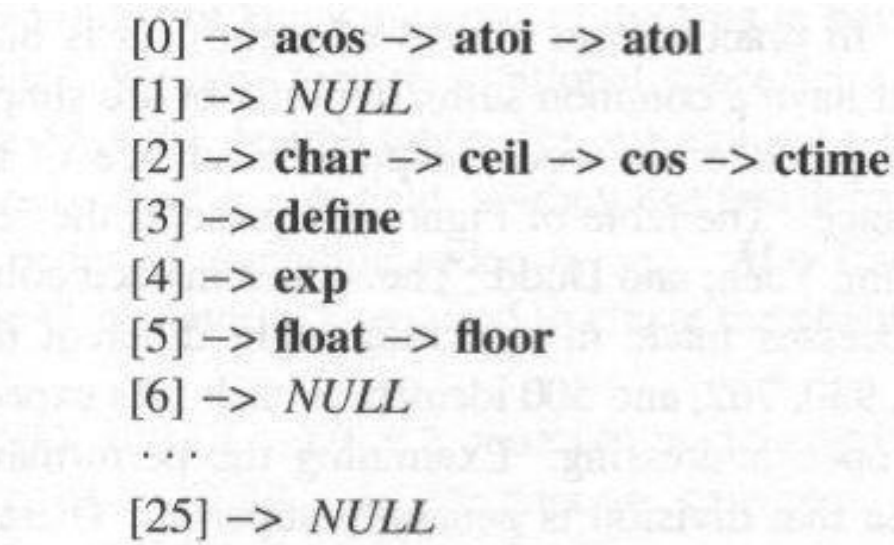


Figure 8.6: Hash chains corresponding to Figure 8.4

– Comparison

- The table of Figure 8.7 presents the results of an empirical study conducted by Lum, Yuen, and Dodd.
- 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.
- As expected, chaining performs better than linear open addressing.
- Examining the performance of the various hash functions, we can see that division is generally superior.
- (Figure 8.7)

$\alpha = \frac{n}{b}$	.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

(Adapted from V. Lum, P. Yuen, and M. Dodd, *CACM*, 1971, Vol. 14, No. 4)

Figure 8.7: Average number of bucket accesses per identifier retrieved.

8.3 Dynamic Hashing

- Motivation for dynamic hashing
  - One of the most important classes of software is the DBMS. A key characteristic of a DBMS is that the amount of information can vary a great deal over time.
    - Various data structures have been suggested for storing the data in a DBMS. In this section, we examine an extension of hashing that permits the technique to be used by a DBMS.
  - Traditional hashing schemes are not ideal because we must statically allocate a portion of memory to hold the hash table.

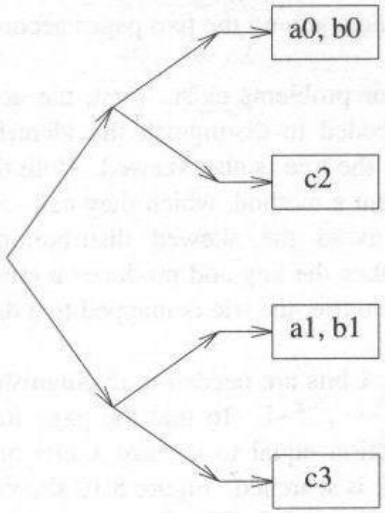
– *Dynamic hashing*, also referred to as *extendible hashing*, retains the fast retrieval time of conventional hashing, while extending the technique so the it can accommodate dynamically increasing and decreasing file size without penalty.

- Dynamic hashing using directories
  - Consider an example where an identifier consists of two characters and each character is represented by 3 bits. Figure 8.8 gives a list of some of these identifiers.
  - (Figure 8.8)

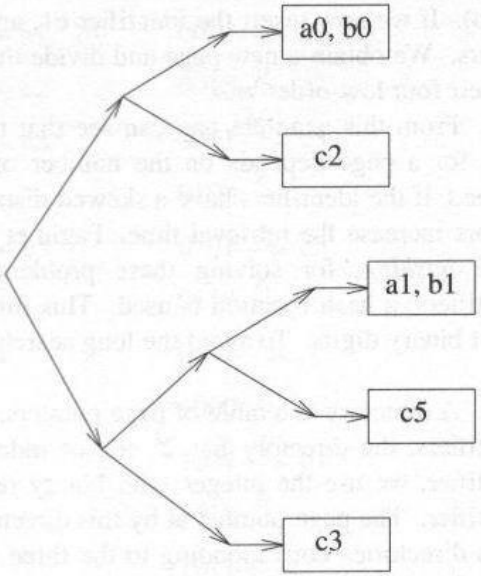
Identifiers	Binary representation
a0	100 000
a1	100 001
b0	101 000
b1	101 001
c0	110 000
c1	110 001
c2	110 010
c3	110 011

Figure 8.8: Some identifiers requiring 3 bits per character

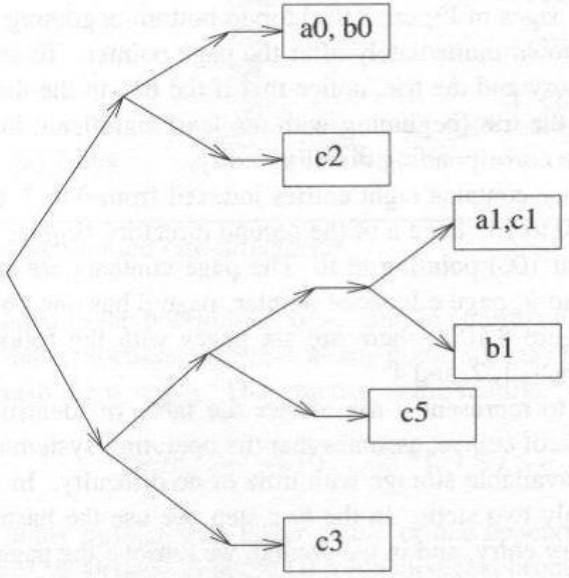
–We would like to place these identifiers into a table that has four page. Each page can hold no more than two identifiers, and the pages are indexed by the 2 bit sequence 00, 01, 10, 11, respectively. We use the two low-order bits of each identifier to determine the page address of the identifier. (Figure 8.9)



(a) two level trie on four pages



(b) inserting c5 with overflow



(c) inserting c1 with overflow

Figure 8.9: A trie to hold identifiers

- We use the term trie to denote a binary tree in which we locate an identifier by following its bit sequence.
  - Notice that this trie has nodes that always branch in two directions corresponding to 0 or 1. Only the leaf nodes of the trie contain a pointer to a page.
- From this example we can see two major problems exist.
  - The access time for a page depends on the number of bits needed to distinguish the identifiers.
  - If the identifiers have a skewed distribution, the tree is also skewed.

– A *directory* is a table of page pointer.

- Figure 8.10 shows the three directories corresponding to the three tries in Figure 8.9.

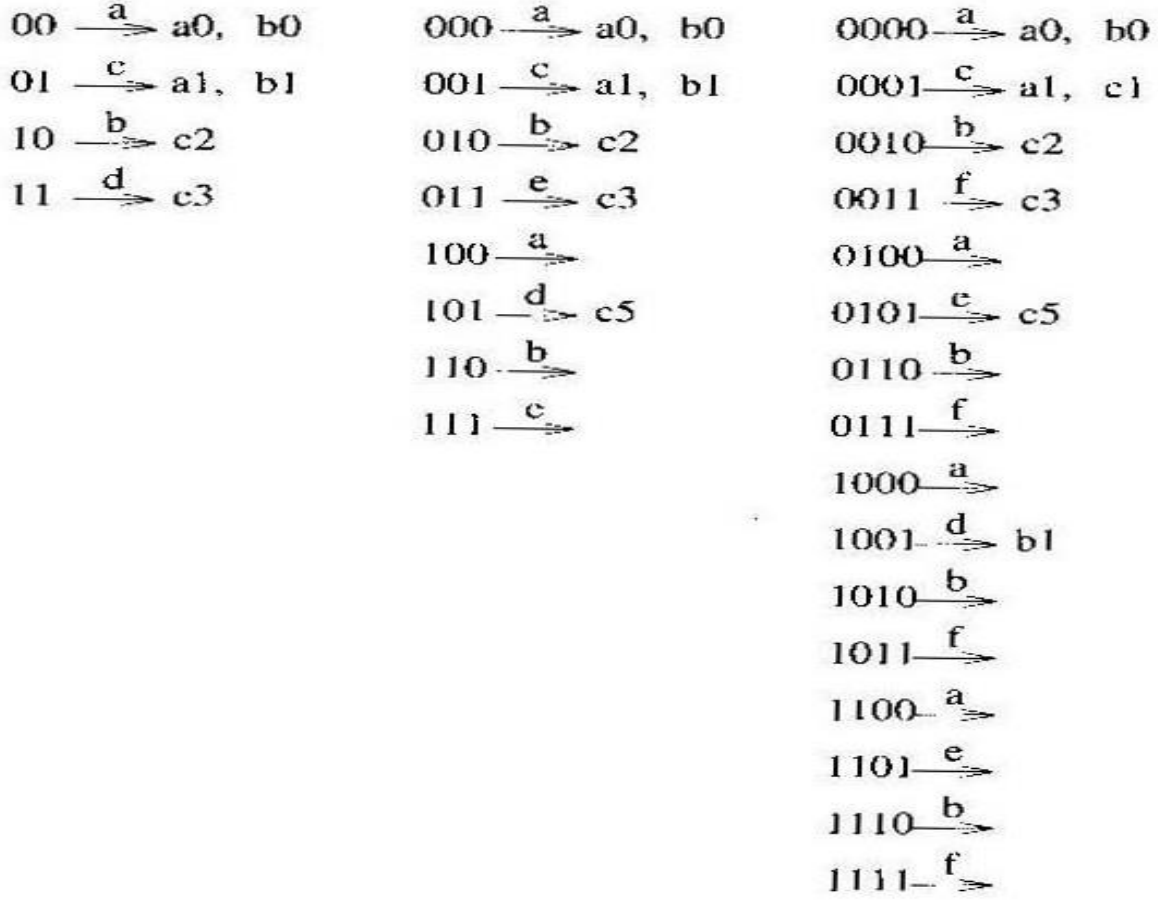


Figure 8.10: Tries collapsed into directories

– Advantage of a directory:

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

– Disadvantage of a directory:

- If the keys are not uniformly divided among the pages, the directory can grow quite large. However, most of the entries point to the same pages.
  - To prevent this from happening, we cannot use the bit sequence of the keys themselves. Instead we translate the bits into a random sequence using a *uniform hash function* as discussed in the previous section.
  - In contrast to the previous section, however, we need a *family* of hash functions, because, at any point, we may require a different number of bits to distinguish the new key.

- Directoryless dynamic hashing
  - One criticism of the dynamic hashing using directories approach is that it always requires at least one level of indirection.
  - Improvement idea:
    - if we assume that we have a contiguous address space which is large enough to hold all the records, we will eliminates the directory. This scheme is referred as *directoryless hashing* or *linear hashing*.
  - Then, we use an example to demonstrate this approach.
    - (Figure 8.12, 8.13)

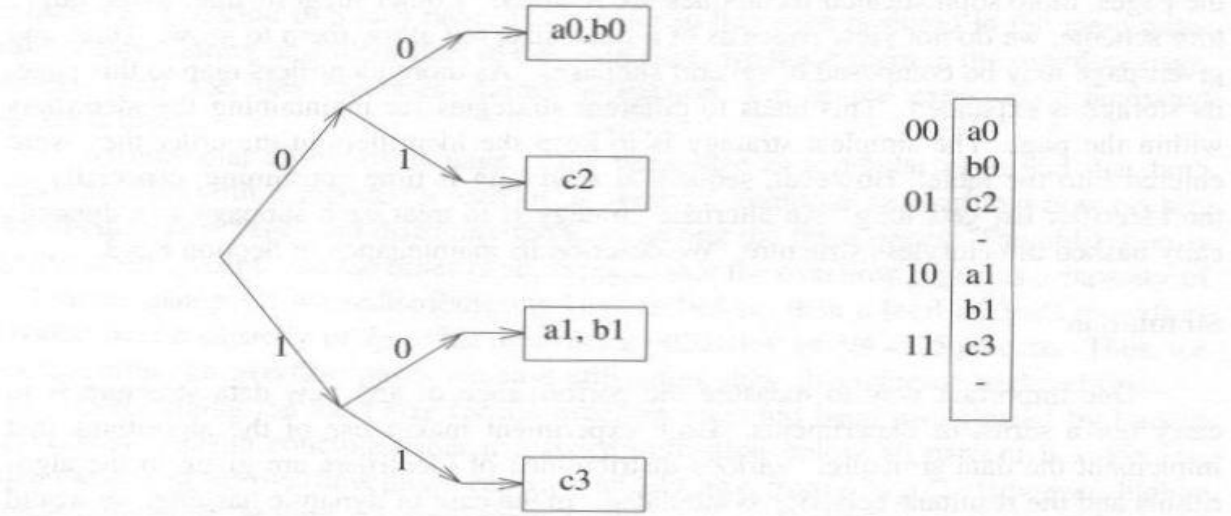


Figure 8.12: A trie mapped to a directoryless, contiguous storage

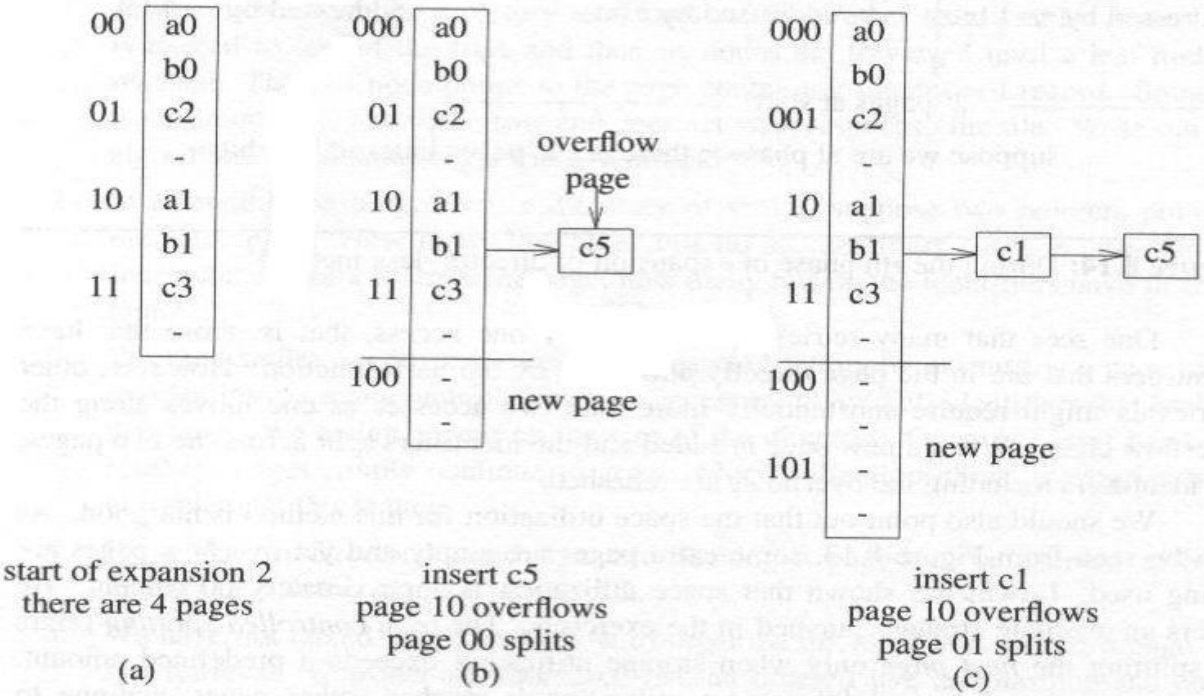


Figure 8.13: An example with two insertions

– Another condition:
  - Figure 8.14 shows the state of file expansion during the *r*th phase at some time *q*.
  - (Figure 8.14)

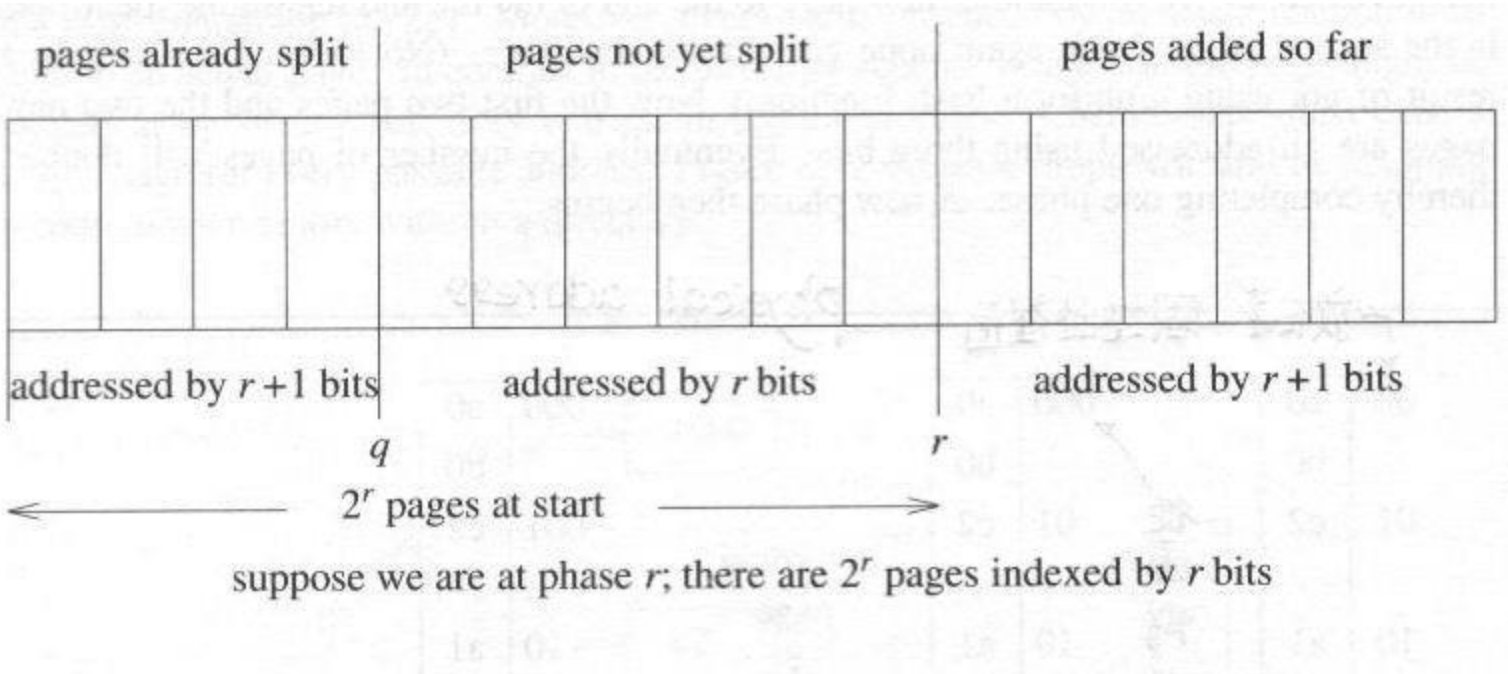


Figure 8.14: During the *r*th phase of expansion of directoryless method

– Some conclusions of the directoryless dynamic hashing:
  - Many retrievals require only one access, that is, those that identifiers that are in page directly address by the hash function.
  - However, other retrievals might require substantially more than two accesses as one moves along the overflow chain
  - When a new page is added and the identifier split across the two pages, all identifiers including the overflows are rehashing.
  - The space utilization for this method is not good.
    – As can be seen from Figure 8.13, some extra pages are empty and yet overflow pages are been used.
    – Litwin has shown that the space utilization is approximately 60 percent. He offers an alternate strategy. The term controlled splitting refers to splitting the next page only when storage utilization exceeds a predefined amount. Litwin suggest that until 80 percent utilization is reached, other pages continue to overflow.

- A natural way to handle overflow is to use one of traditional hashing scheme discussed earlier, such as open addressing.
- From the example, one see that the longest overflow chains occur for those page that are near the end of expansion phase since they are last to split.
  - In contrast, those pages that are split early are generally underfull.