



HASHING

DR. MEGHA UMMAT

HASHING

- The main operation used by the searching methods is comparison of keys.
- A different approach to searching calculates the position of the key in the table based on the value of the key. The value of the key is the only indication of the position.
- When the key is known, the position in the table can be accessed directly, without making any other preliminary tests, as required in a binary search or when searching a tree.
- This means that the search time is reduced from $O(n)$, as in a sequential search, or from $O(\lg n)$, as in a binary search, to 1 or at least $O(1)$; regardless of the number of elements being searched, the run time is always the same.

HASHING

- We need to find a function h that can transform a particular key K , be it a string, number, record, or the like, into an index in the table used for storing items of the same type as K . The function h is called a *hash function*.
- If h transforms different keys into different numbers, it is called a *perfect hash function*.
- To create a perfect hash function, which is always the goal, the table has to contain at least the same number of positions as the number of elements being hashed. But the number of elements is not always known ahead of time.

HASH FUNCTIONS

- The number of hash functions that can be used to assign positions to n items in a table of m positions (for $n \leq m$) is equal to m^n .
- The number of perfect hash functions is the same as the number of different placements of these items in the table and is equal to $m! / (m-n)!$
- Most of these functions are too unwieldy for practical applications and cannot be represented by a concise formula.
- When two keys hash to the same address it is called **collision**.

DIVISION

- A hash function must guarantee that the number it returns is a valid index to one of the table cells.
- The simplest way to accomplish this is to use division modulo $TSize = sizeof(table)$, as in $h(K) = K \bmod TSize$, if K is a number.
- It is best if $TSize$ is a prime number; otherwise, $h(K) = (K \bmod p) \bmod TSize$ for some prime $p > TSize$ can be used.
- The division method is usually the preferred choice for the hash function if very little is known about the keys.

FOLDING

- In this method, the key is divided into several parts. These parts are combined or folded together and are often transformed in a certain way to create the target address.
- There are two types of folding: *shift folding* and *boundary folding*.
- The key is divided into several parts and these parts are then processed using a simple operation such as addition to combine them in a certain way.

SHIFT FOLDING

- In shift folding, the different parts of the key are put underneath one another and then processed.
- For example, a social security number (SSN) 123-45-6789 can be divided into three parts, 123, 456, 789, and then these parts can be added. The resulting number, 1,368, can be divided modulo *TSize* or, if the size of the table is 1,000, the last three digits can be used for the address.
- Another possibility is to divide the same number 123-45-6789 into five parts (say, 12, 34, 56, 78, and 9), add them, and divide the result modulo *TSize*.

BOUNDARY FOLDING

- With boundary folding, the key is seen as being written on a piece of paper that is folded on the borders between different parts of the key. In this way, every other part will be put in the reverse order.
- Consider the same three parts of the SSN: 123, 456, and 789. The first part, 123, is taken in the same order, then the piece of paper with the second part is folded underneath it so that 123 is aligned with 654, which is the second part, 456, in reverse order. When the folding continues, 789 is aligned with the two previous parts.
- The result is $123 + 654 + 789 = 1,566$.

FOLDING

- A bit-oriented version of shift folding is obtained by applying the exclusive-or operation, \wedge .
- In the case of strings, one approach processes all characters of the string by “xor’ing” them together and using the result for the address.
- For example, for the string “abcd,” $h(\text{“abcd”}) = \text{“a”} \wedge \text{“b”} \wedge \text{“c”} \wedge \text{“d.”}$ However, this simple method results in addresses between the numbers 0 and 127.
- For better result, chunks of strings are “xor’ed” together rather than single characters, $h(\text{“abcd”}) = \text{“ab”} \text{ xor “cd”}$ (most likely divided modulo $TSize$).

MID-SQUARE FUNCTION

- In the mid-square method, the key is *squared* and the middle or *mid* part of the result is used as the address.
- If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding.
- In a mid-square hash function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys.
- For example, if the key is 3,121, then $3,121^2 = 9,740,641$, and for the 1,000-cell table, $h(3,121) = 406$, which is the middle part of $3,121^2$.

MID-SQUARE FUNCTION

- In practice, it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key.
- If we assume that the size of the table is 1,024, then, in this example, the binary representation of $3,121^2$ is the bit string 100101001010000101100001. This middle part, the binary number 0101000010, is equal to 322.

EXTRACTION

- In the extraction method, only a part of the key is used to compute the address.
- For the social security number 123-45-6789, this method might use the first four digits, 1,234; the last four, 6,789; the first two combined with the last two, 1,289; or some other combination.
- Each time, only a portion of the key is used. If this portion is carefully chosen, it can be sufficient for hashing, provided the omitted portion distinguishes the keys only in an insignificant way.

EXTRACTION

- For example, in some university settings, all international students' ID numbers start with 999. Therefore, the first three digits can be safely omitted in a hash function that uses student IDs for computing table positions.
- Similarly, the starting digits of the ISBN code are the same for all books published by the same publisher (e.g., 1133 for the publishing company Cengage Learning). Therefore, they should be excluded from the computation of addresses if a data table contains only books from one publisher.

RADIX TRANSFORMATION

- Using the radix transformation, the key K is transformed into another number base; K is expressed in a numerical system using a different radix.
- If K is the decimal number 345, then its value in base 9 (nonal) is 423. This value is then divided modulo $TSize$, and the resulting number is used as the address of the location to which K should be hashed.
- Collisions, however, cannot be avoided. If $TSize = 100$, then although 345 and 245 (decimal) are not hashed to the same location, 345 and 264 are because 264 decimal is 323 in the nonal system, and both 423 and 323 return 23 when divided modulo 100.

UNIVERSAL HASH FUNCTIONS

- When very little is known about keys, a *universal class of hash functions* can be used.
- A class of functions is universal when for any sample, a randomly chosen member of that class will be expected to distribute the sample evenly, whereby members of that class guarantee low probability of collisions.
- Let H be a class of functions from a set of *keys* to a hash table of $TSize$. We say that H is universal if for any two different keys x and y from *keys*, the number of hash functions h in H for which $h(x) = h(y)$ equals $|H|/TSize$.
- That is, H is universal if no pair of distinct keys are mapped into the same index by a randomly chosen function h with the probability equal to $1/TSize$.
- In other words, there is one chance in $Tsize$ that two keys collide when a randomly picked hash function is applied.

UNIVERSAL HASH FUNCTIONS EXAMPLE

Example 1

For a prime number $p \geq |keys|$, and randomly chosen numbers a and b ,

$$H = \{h_{a,b}(K): h_{a,b}(K) = ((aK+b) \bmod p) \bmod TSize \text{ and } 0 \leq a, b < p\}$$

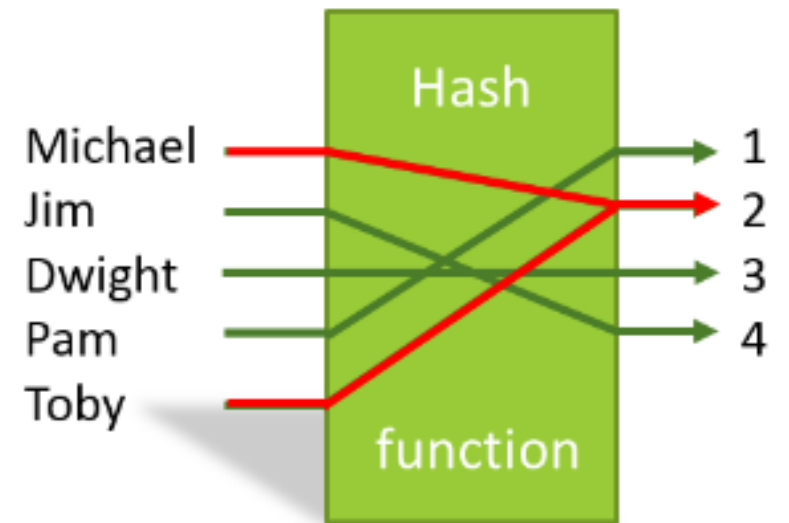
Example 2

Another example is a class H for keys considered to be sequences of bytes, $K = K_0K_1 \dots K_{r-1}$. For some prime $p \geq 2^8 = 256$ and a sequence $a = a_0, a_1, \dots, a_{r-1}$,

$$H = \{h_a(K): h_a(K) = \left(\left(\sum_{i=0}^{r-1} a_i K_i \right) \bmod p \right) \bmod TSize \text{ and } 0 \leq a_0, a_1, \dots, a_{r-1} < p\}$$

COLLISION RESOLUTION

- Almost all hash functions can hash more than one key to the same position i.e. face collision.
- If the size of the table is increased, it can contribute to avoiding conflicts between hashed keys.
- These two factors—hash function and table size—may minimize the number of collisions, but they cannot completely eliminate them.
- Some strategies to avoid collision are discussed.



OPEN ADDRESSING

- In the open addressing method, when a key collides with another key, the collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed.
- If position $h(K)$ is occupied, then the positions in the probing sequence
$$\text{norm}(h(K) + p(1)), \text{norm}(h(K) + p(2)), \dots, \text{norm}(h(K) + p(i)), \dots$$
- are tried until either an available cell is found or the same positions are tried repeatedly or the table is full.
- Function p is a *probing function*, i is a *probe*, and norm is a *normalization function*, most likely, division modulo the size of the table.

LINEAR PROBING

- The simplest method is *linear probing*, for which $p(i) = i$, and for the i th probe, the position to be tried is $(h(K) + i) \bmod TSize$.
- In linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found.
- If the end of the table is reached and no empty cell has been found, the search is continued from the beginning of the table and stops—in the extreme case—in the cell preceding the one from which the search started.
- Linear probing, however, has a tendency to create clusters in the table.

LINEAR PROBING

- Figure contains an example where a key K_i is hashed to the position i .

Insert: A_5, A_2, A_3	B_5, A_9, B_2	B_9, C_2
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
(a)	(b)	(c)

LINEAR PROBING

- In this example, the empty cells following clusters have a much greater chance to be filled than other positions. This probability is equal to $(sizeof(cluster) + 1)/TSize$.
- Other empty cells have only $1/TSize$ chance of being filled.
- If a cluster is created, it has a tendency to grow, and the larger a cluster becomes, the larger the likelihood that it will become even larger.
- This fact undermines the performance of the hash table for storing and retrieving data.
- The problem at hand is how to avoid cluster buildup.

QUADRATIC PROBING

- An answer can be found in a more careful choice of the probing function p . One such choice is a quadratic function so that the resulting formula is

$$p(i) = h(K) + (-1)^{i-1}((i+1)/2)^2 \text{ for } i = 1, 2, \dots, TSize - 1$$

$$h(K) + i^2, h(K) - i^2 \text{ for } i = 1, 2, \dots, (TSize - 1)/2$$

$$h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \dots, h(K) + (TSize - 1)^2/4,$$

$$h(K) - (TSize - 1)^2/4$$

all divided modulo $TSize$.

QUADRATIC PROBING

- Ideally, the table size should be a prime $4j + 3$ of an integer j , which guarantees the inclusion of all positions in the probing sequence.
- For example, if $j = 4$, then $TSize = 19$, and assuming that $h(K) = 9$ for some K , the resulting sequence of probes is

9, 10, 8, 13, 5, 18, 0, 6, 12, 15, 3, 7, 11, 1, 17, 16, 2, 14, 4

QUADRATIC PROBING

Insert: A_5, A_2, A_3

0	
1	
2	A_2
3	A_3
4	
5	A_5
6	
7	
8	
9	

(a)

B_5, A_9, B_2

0	
1	B_2
2	A_2
3	A_3
4	
5	A_5
6	B_5
7	
8	
9	A_9

(b)

B_9, C_2

0	B_9
1	B_2
2	A_2
3	A_3
4	
5	A_5
6	B_5
7	
8	C_2
9	A_9

(c)

QUADRATIC PROBING

- Although using quadratic probing gives much better results than linear probing, the problem of cluster buildup is not avoided altogether, because for keys hashed to the same location, the same probe sequence is used.
- Such clusters are called *secondary clusters*. These secondary clusters, however, are less harmful than primary clusters.

PROBING

- Another possibility is to have p be a random number generator.
- This approach prevents the formation of secondary clusters, but it causes a problem with repeating the same probing sequence for the same keys.
- If the random number generator is initialized at the first invocation, then different probing sequences are generated for the same key K .
- Consequently, K is hashed more than once to the table, and even then it might not be found when searched. Therefore, the random number generator should be initialized to the same seed for the same key before beginning the generation of the probing sequence.

PROBING

- This can be achieved in C++ by using the `srand()` function with a parameter that depends on the key; for example, $p(i) = \text{srand}(\text{sizeof}(K)) \cdot i$ or $\text{srand}(K[0]) + i$.
- To avoid relying on `srand()`, a random number generator can be written that assures that each invocation generates a unique number between 0 and $TSize - 1$.
- The following algorithm was developed by Robert Morris for tables with $TSize = 2^n$ for some integer n :

```
generateNumber()  
    static int r = 1;  
    r = 5*r;  
    r = mask out n + 2 low-order bits of r;  
    return r/4;
```

DOUBLE HASHING

- The problem of secondary clustering is best addressed with *double hashing*.
- This method utilizes two hash functions, one for accessing the primary position of a key, h , and a second function, h_p , for resolving conflicts. The probing sequence becomes

$$h(K), h(K) + h_p(K), \dots, h(K) + i \cdot h_p(K), \dots$$

(all divided modulo $TSize$).

- Experiments indicate that secondary clustering is generally eliminated because the sequence depends on the values of h_p , which, in turn, depend on the key.

DOUBLE HASHING

- If the key K_1 is hashed to the position j , the probing sequence is

$$j, j + h_p(K_1), j + 2 \cdot h_p(K_1), \dots \quad (\text{all divided modulo } TSize).$$

- If another key K_2 is hashed to $j + h_p(K_1)$, then the next position tried is $j + h_p(K_1) + h_p(K_2)$, not $j + 2 \cdot h_p(K_1)$, which avoids secondary clustering if h_p is carefully chosen.
- Also, even if K_1 and K_2 are hashed primarily to the same position j , the probing sequences can be different for each.

DOUBLE HASHING

- Using two hash functions can be time-consuming, especially for sophisticated functions.
- Therefore, the second hash function can be defined in terms of the first, as in $h_p(K) = i \cdot h(K) + 1$.
- The probing sequence for K_1 is

$$j, 2j + 1, 5j + 2, \dots$$

(modulo $TSize$).

- If K_2 is hashed to $2j + 1$, then the probing sequence for K_2 is

$$2j + 1, 4j + 3, 10j + 7, \dots$$

NUMBER OF TRIALS FOR SUCCESSFUL & UNSUCCESSFUL SEARCH

	linear probing	quadratic probing ^a	double hashing
successful search	$\frac{1}{2} \left(1 + \frac{1}{1 - LF} \right)$	$1 - \ln(1 - LF) - \frac{LF}{2}$	$\frac{1}{LF} \ln \frac{1}{1 - LF}$
unsuccessful search	$\frac{1}{2} \left(1 + \frac{1}{(1 - LF)^2} \right)$	$\frac{1}{1 - LF} - LF - \ln(1 - LF)$	$\frac{1}{1 - LF}$
Loading factor $LF = \frac{\text{number of elements in the table}}{\text{table size}}$			

^a The formulas given in this column approximate any open addressing method that causes secondary clusters to arise, and quadratic probing is only one of them.

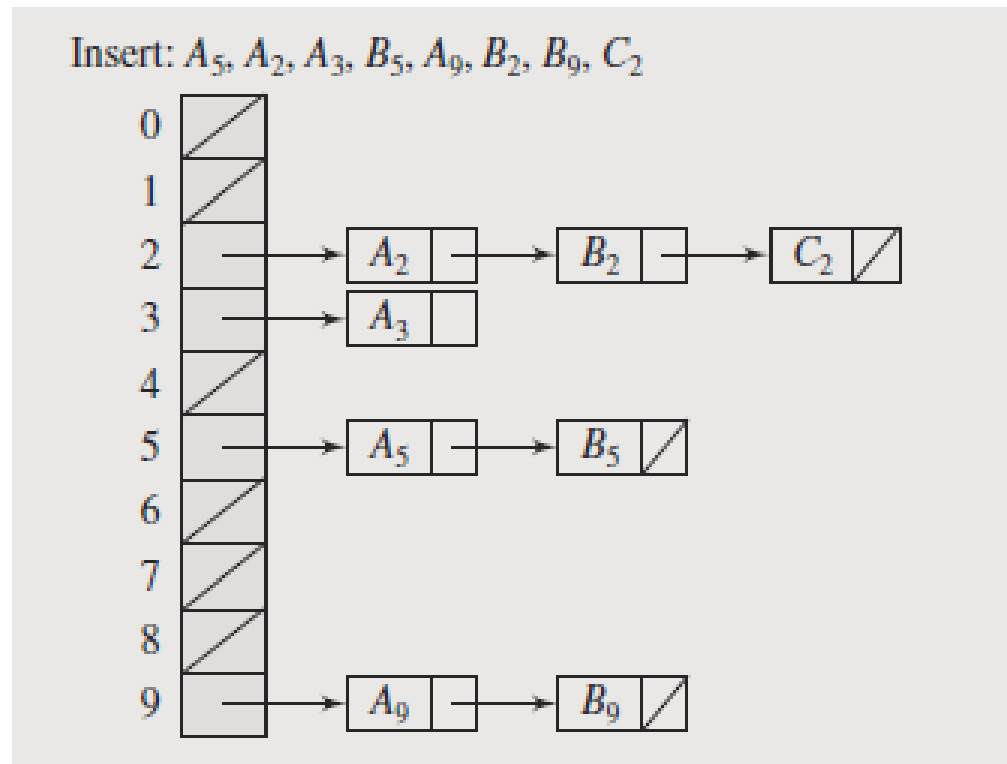
CHAINING

- Keys do not have to be stored in the table itself.
- In *chaining*, each position of the table is associated with a linked list or *chain* of structures whose info fields store keys or references to keys.
- This method is called *separate chaining*, and a table of references (pointers) is called a *scatter table*.
- In this method, the table can never overflow, because the linked lists are extended only upon the arrival of new keys.
- For short linked lists, this is a very fast method, but increasing the length of these lists can significantly degrade retrieval performance.
- Performance can be improved by maintaining an order on all these lists so that, for unsuccessful searches, an exhaustive search is not required.



CHAINING

- In chaining, colliding keys are put on the same linked list.



CHAINING

- This method requires additional space for maintaining pointers.
- The table stores only pointers, and each node requires one pointer field.
- Therefore, for n keys, $n + TSize$ pointers are needed, which for large n can be a very demanding requirement.



COALESCED CHAINING

- A version of chaining called *coalesced hashing* (or *coalesced chaining*) combines linear probing with chaining.
- In this method, the first available position is found for a key colliding with another key, and the index of this position is stored with the key already in the table.
- In this way, a sequential search down the table can be avoided by directly accessing the next element on the linked list.
- Each position *pos* of the table stores an object with two members: info for a key and next with the index of the next key that is hashed to *pos*.

COALESCED CHAINING

- Available positions can be marked by, say, -2 in next; -1 can be used to indicate the end of a chain.
- This method requires $TSize \cdot sizeof(next)$ more space for the table in addition to the space required for the keys. This is less than for chaining, but the table size limits the number of keys that can be hashed into the table.
- An overflow area known as a *cellar* can be allocated to store keys for which there is no room in the table. This area should be located dynamically if implemented as a list of arrays.

COALESCED HASHING

- Coalesced hashing puts a colliding key in the last available position of the table.

Insert: A_5, A_2, A_3

0		
1		
2	A_2	
3	A_3	
4		
5	A_5	
6		
7		
8		
9		

(a)

B_5, A_9, B_2

0		
1		
2	A_2	
3	A_3	
4		
5	A_5	
6		
7	B_2	
8	A_9	
9	B_5	

(b)

B_9, C_2

0		
1		
2	A_2	
3	A_3	
4	C_2	
5	A_5	
6	B_9	
7	B_2	
8	A_9	
9	B_5	

(c)

COALESCED HASHING WITH CELLAR


Insert: A_5, A_2, A_3

0		
1		
2	A_2	
3	A_3	
4		
5	A_5	
6		
7		
8		
9		
10		
11		
12		

(a)

B_5, A_9, B_2


0		
1		
2	A_2	
3	A_3	
4		
5	A_5	
6		
7		
8		
9	A_9	
10		
11	B_2	
12	B_5	



(b)

B_9, C_2

0		
1		
2	A_2	
3	A_3	
4		
5	A_5	
6		
7		
8	C_2	
9	A_9	
10	B_9	
11	B_2	
12	B_5	



(c)

BUCKET ADDRESSING

- Another solution to the collision problem is to store colliding elements in the same position in the table. This can be achieved by associating a *bucket* with each address.
- A bucket is a block of space large enough to store multiple items.
- By using buckets, the problem of collisions is not totally avoided.
- If a bucket is already full, then an item hashed to it has to be stored somewhere else.
- By incorporating the open addressing approach, the colliding item can be stored in the next bucket if it has an available slot when using linear probing, or it can be stored in some other bucket when, say, quadratic probing is used.

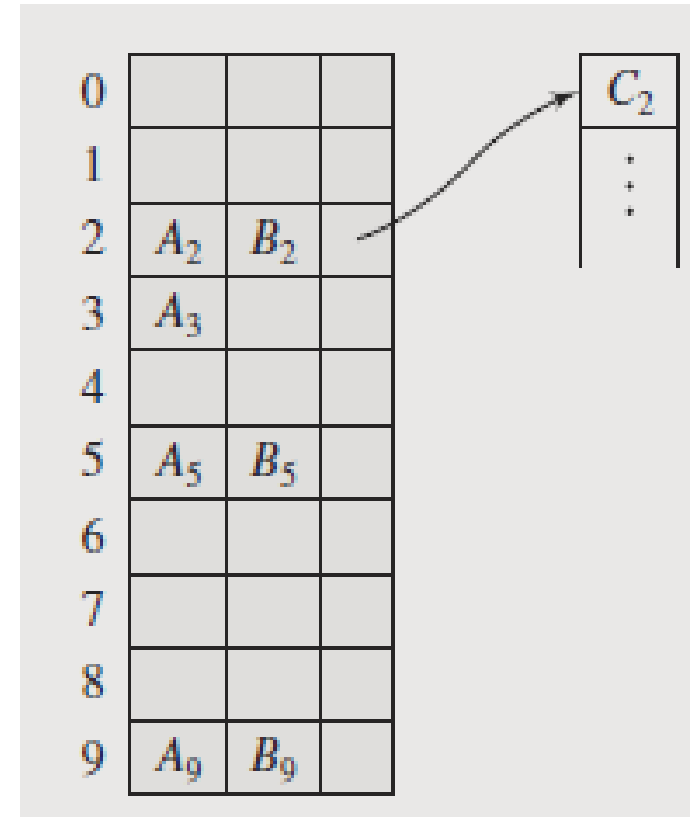
BUCKET ADDRESSING & LINEAR PROBING

Insert: $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$

0		
1		
2	A_2	B_2
3	A_3	C_2
4		
5	A_5	B_5
6		
7		
8		
9	A_9	B_9

BUCKET ADDRESSING & OVERFLOW AREA

- The colliding items can also be stored in an overflow area.
- In this case, each bucket includes a field that indicates whether the search should be continued in this area or not.
- It can be simply a yes/no marker. In conjunction with chaining, this marker can be the number indicating the position in which the beginning of the linked list associated with this bucket can be found in the overflow area.



DELETION

- How can we remove data from a hash table?
- With a chaining method, deleting an element leads to the deletion of a node from a linked list holding the element.
- For other methods, a deletion operation may require a more careful treatment of collision resolution.
- Consider the table in Figure a in which the keys are stored using linear probing.
- The keys have been entered in the following order: A_1 , A_4 , A_2 , B_4 , B_1 . After A_4 is deleted and position 4 is freed (Figure b), we try to find B_4 by first checking position 4.

Insert: A_1, A_4, A_2, B_4, B_1

0	
1	A_1
2	A_2
3	B_1
4	A_4
5	B_4
6	
7	
8	
9	

(a)

Delete: A_4

0	
1	A_1
2	A_2
3	B_1
4	
5	B_4
6	
7	
8	
9	

(b)

DELETION

- But this position is now empty, so we may conclude that B_4 is not in the table. The same result occurs after deleting A_2 and marking cell 2 as empty (Figure c).
- Then, the search for B_1 is unsuccessful, because if we are using linear probing, the search terminates at position 2.
- The situation is the same for the other open addressing methods.

Delete: A_2

0	
1	A_1
2	
3	B_1
4	
5	B_4
6	
7	
8	
9	

(c)

DELETION

- If we leave deleted keys in the table with markers indicating that they are not valid elements of the table, any subsequent search for an element does not terminate prematurely.
- When a new key is inserted, it overwrites a key that is only a space filler.
- However, for a large number of deletions and a small number of additional insertions, the table becomes overloaded with deleted records, which increases the search time because the open addressing methods require testing the deleted elements.
- Therefore, the table should be purged after a certain number of deletions by moving undeleted elements to the cells occupied by deleted elements.
- Cells with deleted elements that are not overwritten by this procedure are marked as free.

MINIMAL PERFECT HASH FUNCTIONS

- In many situations, when the body of data is fixed, a hash function can be devised after the data are known. Such a function may really be a perfect hash function if it hashes items on the first attempt.
- In addition, if such a function requires only as many cells in the table as the number of data so that no empty cell remains after hashing is completed, it is called a *minimal perfect hash function*.
- Wasting time for collision resolution and wasting space for unused table cells are avoided in a minimal perfect hash function.