CSCI2100B Data Structures
Hashing

Irwin King

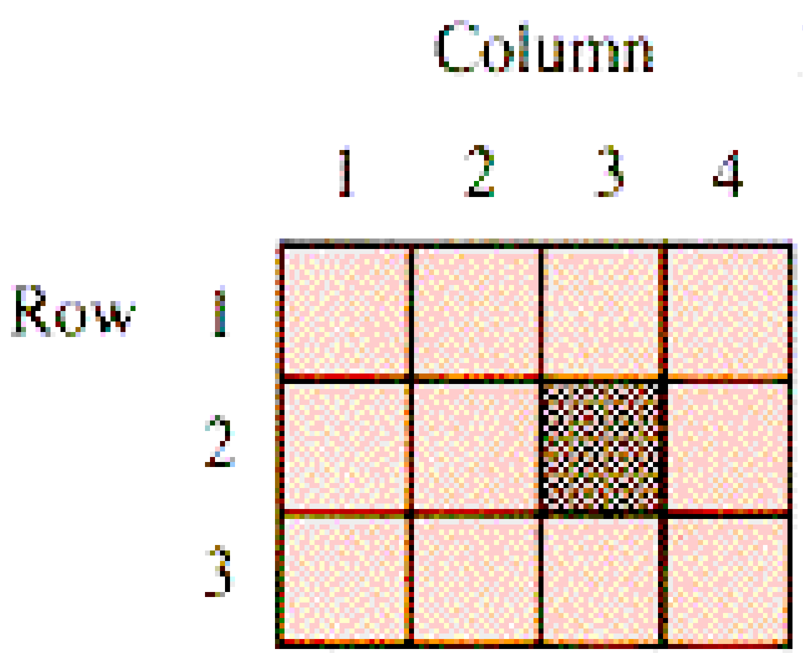
king@cse.cuhk.edu.hk
http://www.cse.cuhk.edu.hk/~king

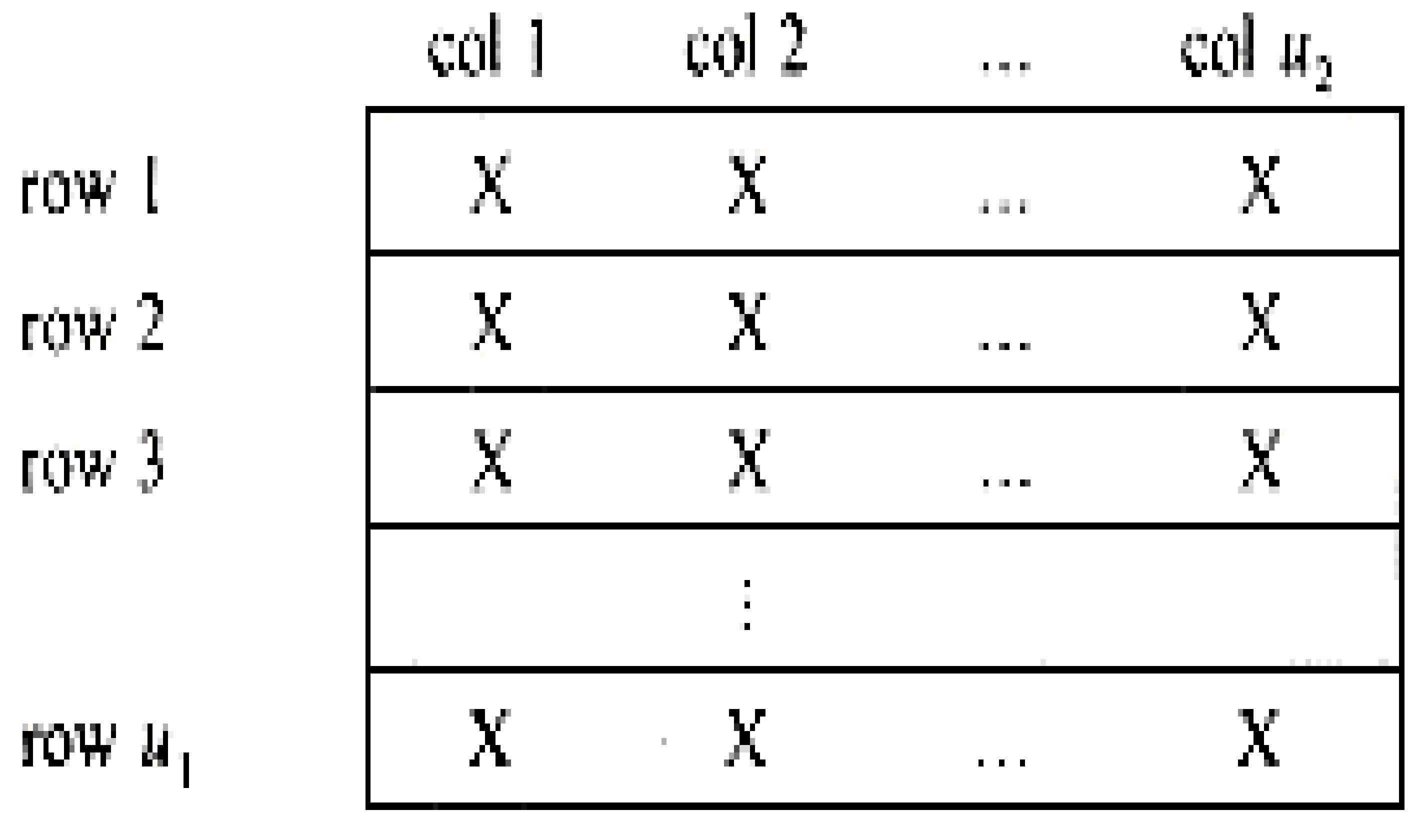
Department of Computer Science & Engineering
The Chinese University of Hong Kong

Introduction

- Hashing is a technique used for performing insertions, deletions and finds in constant **average** time.
- Tree operations that require any ordering information among the elements are not supported efficiently.
  - See several methods of implementing the hash table.
  - Compare these methods analytically.
  - Show numerous applications of hashing.
  - Compare hash tables with binary search trees.

Rectangular Arrays





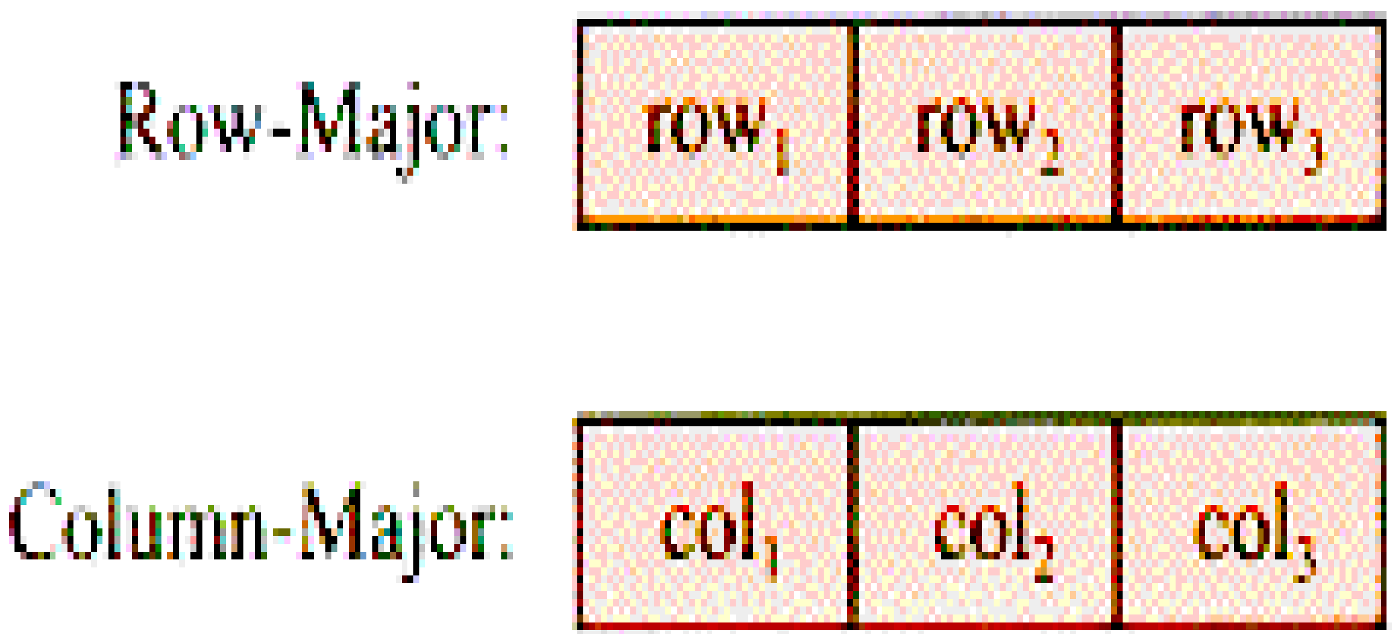
(a)

Row- and Column-Major Ordering

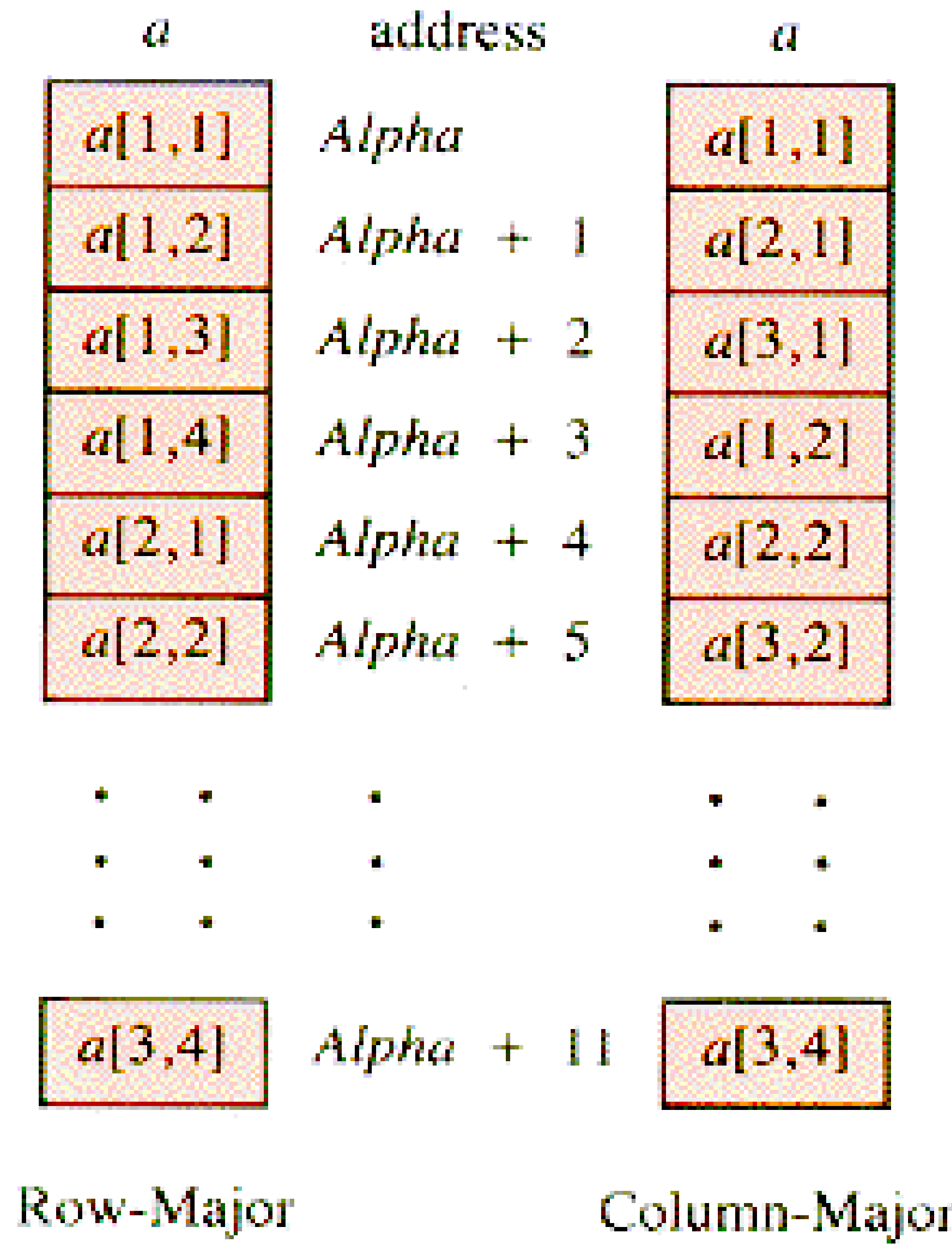
- How does one index an entry in an array?

- Entry [i,j] goes to position $ni+j$ for row-major ordering and $i+jm$ for column-major ordering when the rows are numbered from 0 to $m-1$ and the columns from 0 to $n-1$ and entry [0,0] is at position 0.

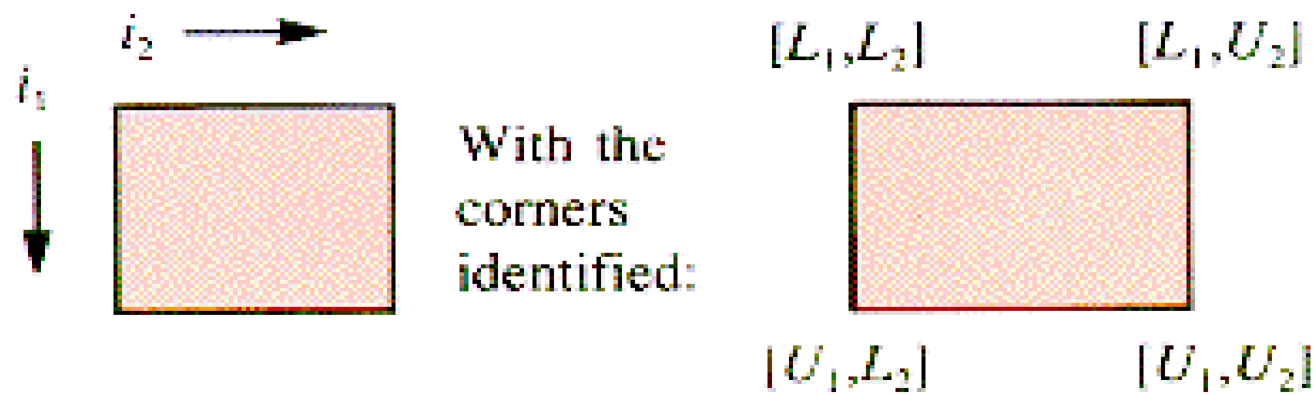
Row- and Column-Major Example

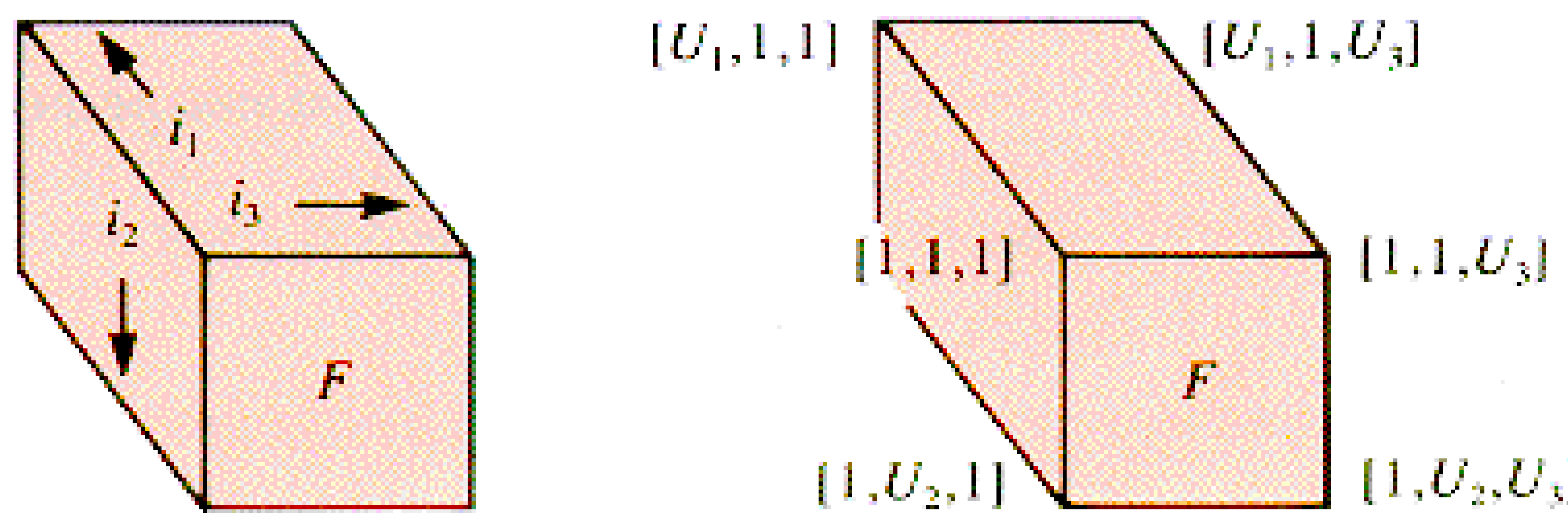


Example

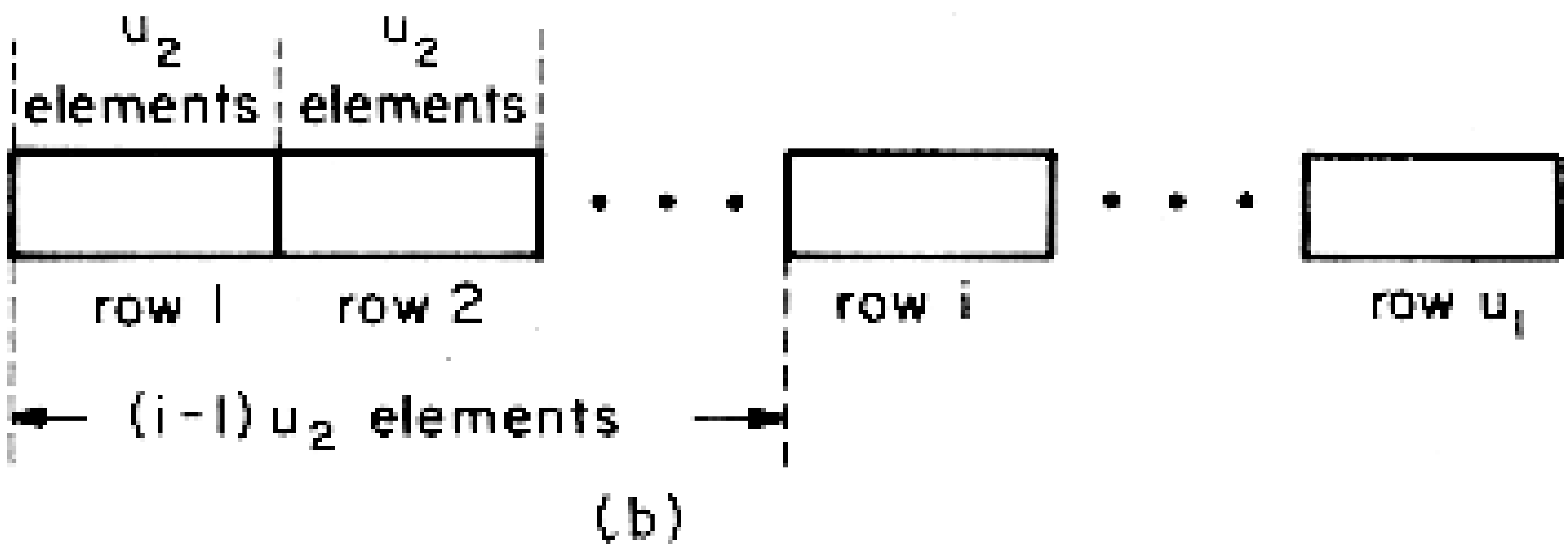

a
address
a
a[1,1]
Alpha
a[1,1]
a[1,2]
Alpha + 1
a[2,1]
a[1,3]
Alpha + 2
a[3,1]
a[1,4]
Alpha + 3
a[1,2]
a[2,1]
Alpha + 4
a[2,2]
a[2,2]
Alpha + 5
a[3,2]
a[3,4]
Alpha + 11
a[3,4]
Row-Major
Column-Major

Implementation Example

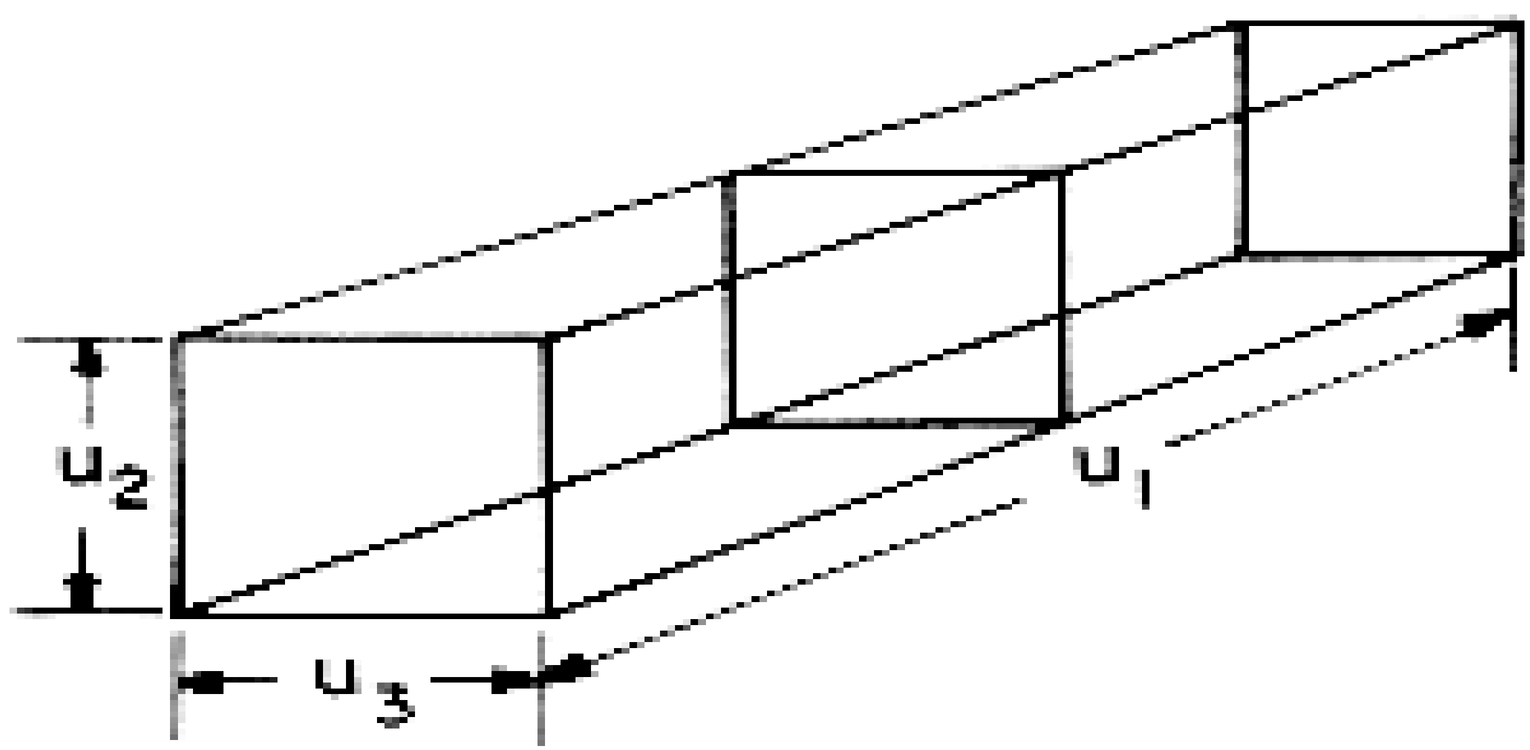




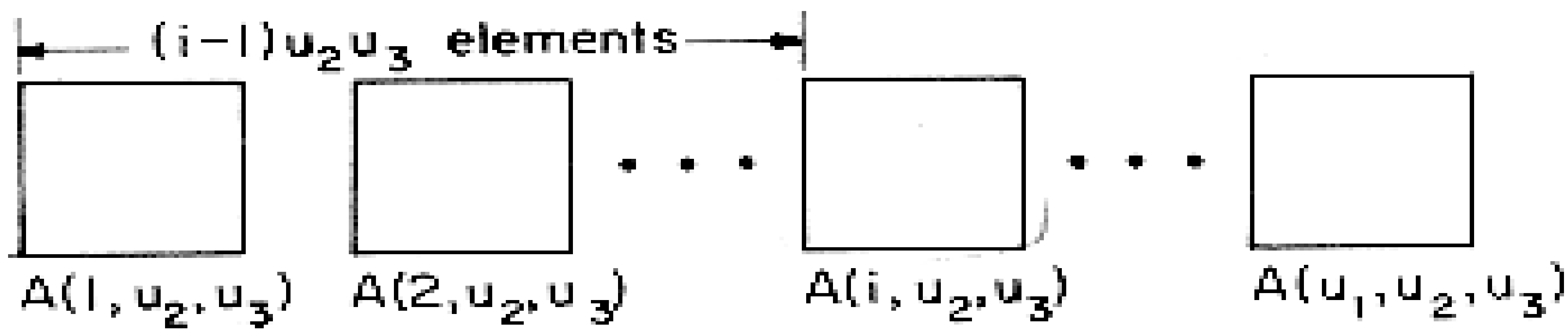
More Implementation Example



3D Array Implementation



3-D Array Implementation



Problem

- In some applications the full use of the whole array is seldom.
- This leads to sparse array or matrix representation.
- For example, population count in a 2-D grid map.

An Access Table

- One method to eliminate the multiplications needed in calculating the index to an entry is to use an access table.
- The array will contain the values 0, n, 2 n, 3 n, ..., (m-1) n.
- Then for all references to the rectangular array, the index for [i,j] is calculated by taking the entry in position i of the auxiliary table, adding j, and going to the resulting position.
- Again we see a trade-off between space used and execution speed.

Example

T Columns

Rows

	1	2	3	4	5
1		12		14	
2					
3		26		40	
4					
5	17		31		
6		9			85

Example

T

Columns

Rows	1	2	3	4	5
1		12		14	
2					
3		26		40	
4					
5	17		31		
6		9			85

Columns

1

5	17

2

1	12
3	26
6	9

3

5	31

4

1	14
3	40

5

6	85

Example

T

Rows \ Columns	1	2	3	4	5
1		12		14	
2					
3		26		40	
4					
5	17		31		
6		9			85

Columns

1		2		3		4		5	
5	17	1	12	5	31	1	14	6	85
		3	26			3	40		
		6	9						

Value	17	12	26	9	31	14	40	85	
Row	5	1	3	6	5	1	3	6	
	1	2			5	6		8	9

FirstInCol	1	2	5	6	8	9
	1	2	3	4	5	6

Tables: A New Abstract Data Type

- Functions: a function is defined in terms of two sets and a correspondence from elements of the first set to elements of the second.
- If f is a function from a set A to a set B, then f assigns to each element of A a unique elements of B.
- The set A is called the domain of f, and the set B is called the codomain of f. The subset of B containing just those element that occur as values of f is called the range of f.

Example

- For a table, we call the domain the **index** set, and we call the codomain the **base type** or **value type**.
- For example, to index into the cell [2,3] the offset value may be 13 if the matrix size is [10,10].

An Abstract Data Type

- A table with index set I and base type T is a function from I into T together with the following operations.
  - Table access: Evaluate the function at any index in I.
  - Table assignment: Modify the function by changing its value at a specified index in I to the new value specified in the assignment.

An Abstract Data Type

- Insertion: Adjoin a new element x to the index set I and define a corresponding value of the function at x.
- Deletion: Delete an element x from the index set I and restrict the function to the resulting smaller domain.

Why Hash Table?

- Often, array indices are not natural identifiers for items that are to be stored, accessed, and retrieved.
- For example, let's try to store the list in an array.

`beef`	`bellpepper`	`blackpepper`	`dillweed`
`onion`	`potato`	`olive`	`salt`
`cumin`	`carrot`	`mushroom`	`tomatopaste`

Problem

- While it is true that STORE and RETRIEVE are O(1) operations for arrays, that is only so if the indices are **known** and the value in the target of a STORE can be discarded.
- Without a complete set in hand it cannot be known that potato has index 10 in the sorted list of items.

Solution

- Use item as a **KEY**--Because an index integer is not known on the entry of one of the items, it would be helpful <u>if the item itself could be used as a key to index the cell</u> where it will be stored.

Solution

- A solution would be to convert the keys (the list items, here) into unique integers and use them as array indices.
- A function that does so is called a hash function.
- The conversion process is called hashing
- The storage structure is called a hash table or scatter-storage.

Example Solution

- We may sum up the ASCII value from each character in the key, e.g., a = 1, b=2, …, z = 26, so beef = 2+5+5+6=18.

Item	HF1(Item)	Item	HF1(Item)
beef	18	carrot	75
onion	67	salt	52
cumin	60	blackpepper	105
dillweed	74	olive	63
bellpepper	107	tomatopaste	145
potato	87	mushroom	122

Introduction to Hashing

- Hashing is a technique used for performing insertions, deletions and finds in constant average time.
- Tree operations that require any ordering information among the elements are not supported efficiently.
  - See several methods of implementing the hash table.
  - Compare these methods analytically.
  - Show numerous applications of hashing.
  - Compare hash tables with binary search trees.

General Idea

- Hash table data structure is merely an array of some fixed size, containing the keys.
- A key is a string with an associated value (for instance, salary information).
- Each key is mapped into some number in the range 0 to H_SIZE - 1 and placed in the appropriate cell.

$$f : \text{key} \rightarrow \text{value}$$

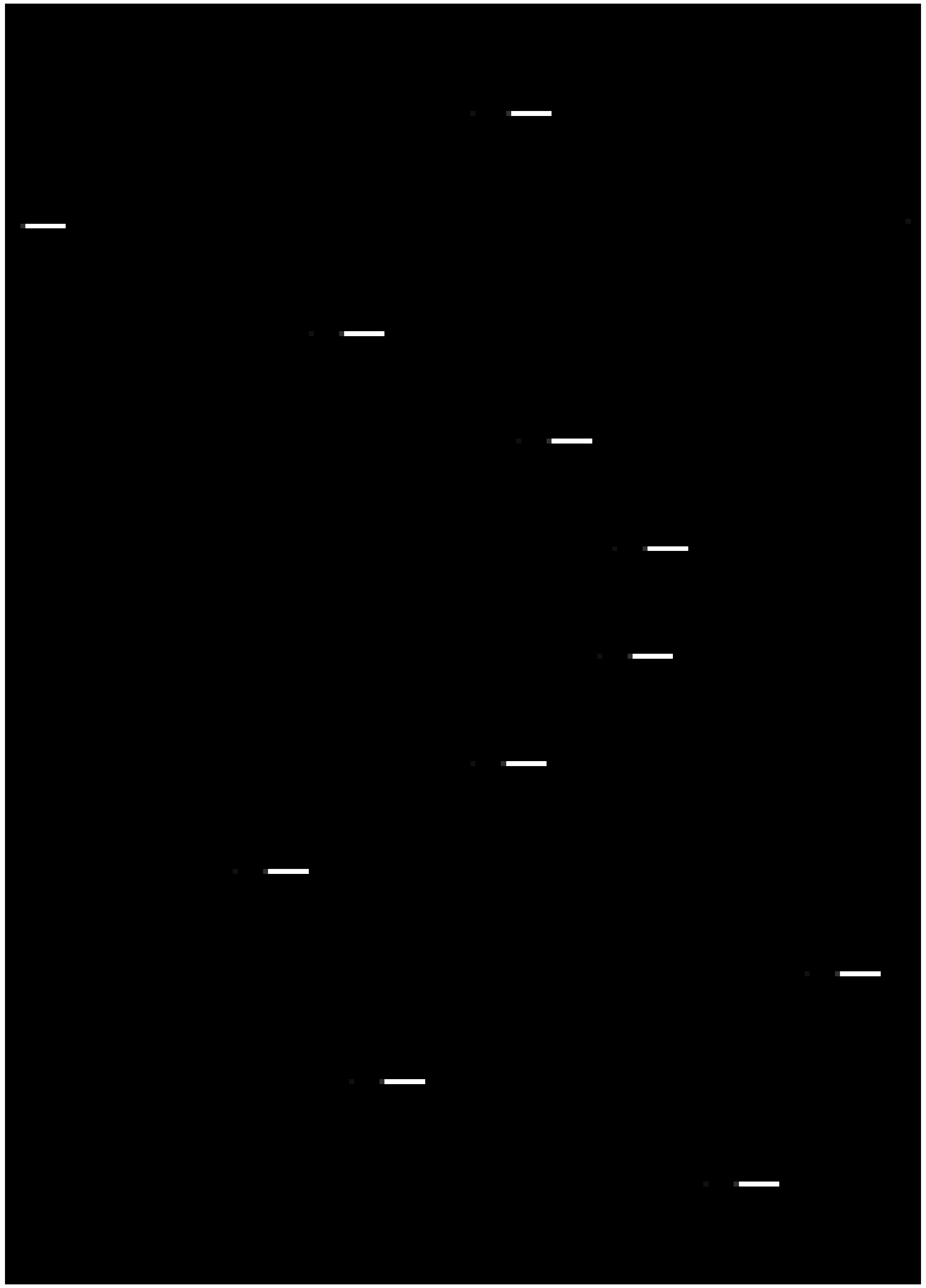
General Idea

- The mapping is called a **hash function**, which ideally should be **simple** to compute and should ensure that any two distinct keys get **different** cells.
- This is difficult to achieve in reality since there are a finite number of cells and a virtually inexhaustible supply of keys.
- We seek a hash function that distributes the keys **evenly** among the cells.

Issues

- Choosing the hashing function
  - How to make sure that one has selected a good function for the application
- Collision handling
  - How to handle conflict when two keys have the same location
- Deletion handling
  - How to deal with the table when items are being removed

Example



Hash Tables

- We can continue to exploit table lookup even in situations where the key is no longer an index that can be used directly as in array indexing.
- What we can do is to set up a **one-to-one** correspondence between the keys by which we wish to retrieve information and indices that we can use to access an array.

Hash Tables

- The idea of a **hash table** is to allow many of the different possible keys that might occur to be mapped to the same location in an array under the action of the index function.
- Others have called **scatter-storage** or **key-transformation**.

Hash Function

- A **hash function** that takes a key and maps it to some index in the array.
- Often, two records may want to go to the same location.
- Therefore, a **collision** may occur and a **collision procedure** must be devised to handle this.

Choosing a Hash Function

- The two principal criteria in selecting a hash function are that
  - it should be **easy** and **quick** to compute and that
  - it should achieve an **even distribution** of the keys that actually occur across the range of indices.

Hash Function

- If the input keys are integers, then simply returning key mod H_SIZE is generally a reasonable strategy.
- For example, student ID mod 10000 would be a reasonable strategy.
- It is usually a good idea to ensure that the table size is **prime**.
- When the input keys are random integers, then this function is **simple** to compute and also distributes the keys **evenly**.

A Simple Hash Function

```
INDEX
hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = 0;

/*1*/       while( *key != '\0' )
/*2*/            hash_val += *key++;

/*3*/       return( hash_val % H_SIZE );
}
```

Another Hash Function

```
INDEX

hash( char *key, unsigned int H_SIZE )

{

return ( ( key[0] + 27*key[1] + 729*key[2] ) %
H_SIZE );

}
```

Notes

- Assuming key has at least two characters plus the NULL terminator.
  - 27 represents the number of letters in the English alphabet, plus the blank.
  - 729 is 27^2.
- This function only examines the first three characters, but if these are random, and the table size is 10,007, as before, then we would expect a reasonably equitable distribution.

Quick Analysis

- Unfortunately, English is not random.
- Although there are 26.26.26 = **17,576** possible combinations of three characters (ignoring blanks), a check of a reasonably large on-line dictionary reveals that the number of different combinations is actually only **2,851**.
- Even if none of these combinations collide, only 28% of the table can actually be hashed to.

A Good Hash Function

```
INDEX
hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = 0;

/*1*/       while( *key != '\0' )
/*2*/             hash_val = ( hash_val << 5 ) + *key++;

/*3*/       return( hash_val % H_SIZE );
}
```

Notes

- This hash function involves all characters in the key .
- It computes $\sum_{i=0}^{Key\_Size-1} Key[Key\_Size - i] \cdot 32^i$
- The code computes a polynomial function (of 32) by use of Horner's rule.
- For instance, another way of computing $h_k = k_1 + 27\ k_2 + 272\ k_3$ is by the formula $h_k = ((k_3) * 27 + k_2) * 27 + k_1$.
- Horner's rule extends this to an nth degree polynomial.

Notes

- It is common to not use all the characters.
- The length and properties of the keys would influence the choice.
- The hash function might include a couple of characters from each field.

Truncation

- Ignore part of the key, and use the remaining part directly as the index (considering non-numeric fields as their numerical codes).
  - Example: If the keys are eight-digit integers and the hash table has 1000 locations, then the first, second, and fifth digits from the right make the hash function, so that 62538194 maps to 394.
- Truncation is a very fast method, but it often fails to distribute the keys evenly through the table.

Folding

- Partition the key into several parts and combine the parts in a convenient way (often using addition or multiplication) to obtain the index.
  - For example, 62538194 maps to 625+381+94 = 1100, which is then truncated to 100.

Modular Arithmetic

- Convert the key to an integer (using the preceding devices as desired), divide by the size of the index range, and take the reminder as the result.
  - For example, ‘abcd’ = 64+65+66+67 mod 100 = 62.

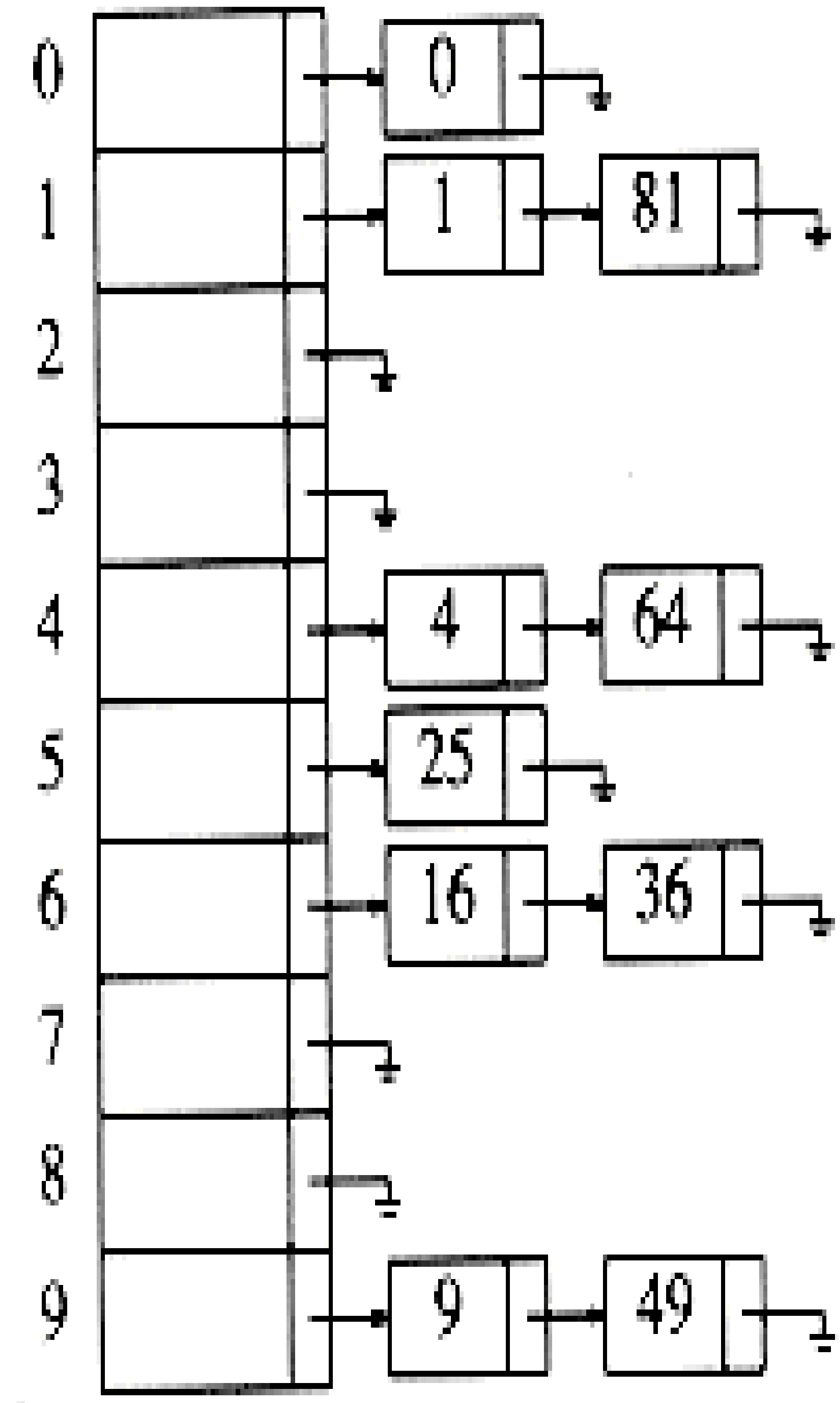
Collision Resolution

- Open Hashing (Separate Chaining)
- Closed Hashing (Open Addressing)
  - Linear probing
  - Quadratic probing
  - Double hashing
- Rehashing

Open Hashing

- The first strategy, commonly known as either **open hashing**, or **separate chaining**, is to keep a **list** of all elements that hash to the same value.
- We assume for this section that the keys are the first 10 perfect squares and that the hashing function is simply hash(x) = x mod 10. (The table size is not prime, but is used here for simplicity.)

Open Hashing Example



Find in Open Hashing

- Find
  - We use the hash function to determine which list to traverse.
  - We then traverse this list in the normal manner, returning the position where the item is found.

Insert in Open Hashing

- Insert
- we traverse down the appropriate list to check whether the element is already in place.
- If the element turns out to be new, it is inserted either at the front of the list or at the end of the list.
- Sometimes new elements are inserted at the **front** of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.

Deletion in Open Hashing

- Deletion
  - The deletion routine is a straightforward implementation of deletion in a linked list.
  - First perform a FIND operation and then perform a delete operation of an item in a linked list.

Advantages of Linked Storage

- Considerable space may be saved.
- It allows simple and efficient collision handling.
- It is no longer necessary that the size of the hash table exceed the number of records.
- Deletion becomes a quick and easy task in a chained hash table.

Disadvantage of Linked Storage

- All the links require space.
- If the records are small this space usage is large when compared with the records.

Closed Hashing (Open Addressing)

- Open hashing has the disadvantage of requiring pointers.
- This tends to slow the algorithm:
  - The time required to allocate new cells.
  - It requires the implementation of a second data structure.
- In a **closed hashing** system, if a collision occurs, **alternate cells** are tried until an empty cell is found.

Closed Hashing

- For example, cells $h_0(x)$, $h_1(x)$, $h_2(x)$, . . . are tried in succession where $h_i(x) = (hash(x) + f(i)) \bmod H\_SIZE$, with $f(0) = 0$.
- The function, f, is the **collision resolution** strategy.
- Because all the data goes inside the table, a bigger table is needed for closed hashing than for open hashing.
- Generally, the load factor should be below l = 0.5 for closed hashing.

Insertion Operation Outline

- An array must be declared that will hold the hash table.
- Initializing all locations in the array to show that they are empty.
- To insert a record into the hash table, the hash function for the key is first calculated.
- If the corresponding location is **empty**, then the record can be inserted, or else
- if the keys are **equal**, then insertion of the new record would not be allowed. In this case, it becomes necessary to resolve the collision.

Find Operation Outline

- To retrieve the record with a given key is entirely similar. The hash function for the key is computed.
- If the desired record is in the corresponding location, then the retrieval has succeeded;
- otherwise,
- while the location is **nonempty** and not all locations have been examined, follow the same steps used for collision resolution.
- If an **empty** position is found, or h_0 have been considered, then no record with the given key is in the table, and the search in unsuccessful.

Linear Probing

- The simplest method to resolve a collision is to start with the hash address (the location where the collision occurred) and do a **sequential search** for the desired key or an empty location.
- The problem with the above method is that the data become **clustered**:
- Records start to appear in long strings of adjacent positions with gaps between the strings.

Example

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Problems

- The time to search for an empty cell may be long.
- The problem of **primary clustering** is essentially one of **instability**.
- If a few keys happen randomly to be near each other, then it becomes more and more likely that other keys will join in the cluster.
- Furthermore, the distribution will become progressively more unbalanced.

Analysis

- It can be shown that the expected number of probes using linear probing is roughly
  - Insertions and unsuccessful searches

$$\frac{1}{2}(1+1/(1-\lambda)^2)$$

  - Successful searched

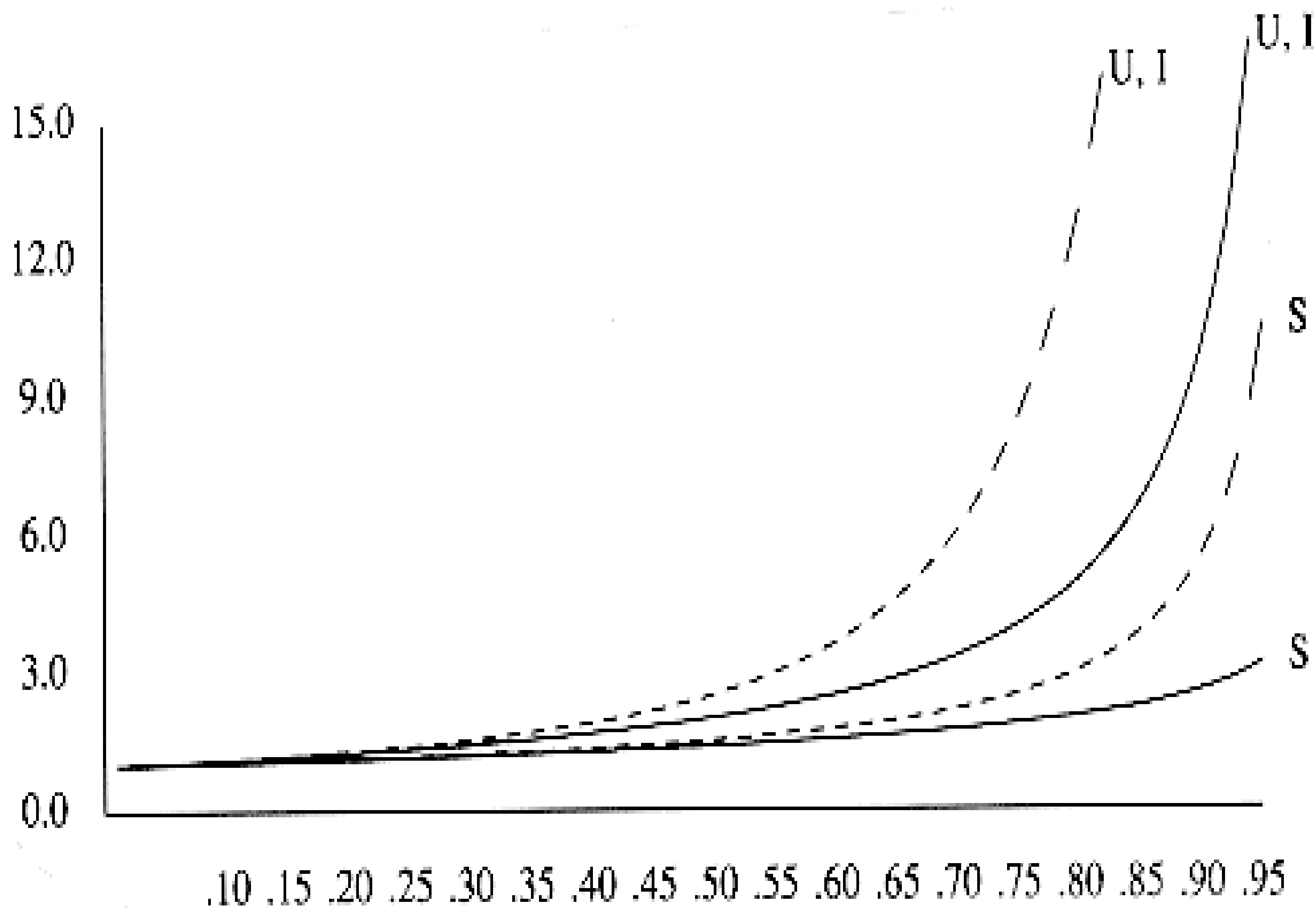
$$\frac{1}{2}(1+1/(1-\lambda))$$

  - λ, of a hash table is the ratio of the number of elements in the hash table to the table size.

Analysis

- We assume a very large table and that each probe is independent of the previous probes.
- The expected number of probes in an unsuccessful search.
    - The number of probes for a successful search = the number of probes required when the particular element was inserted.
    - When an element is inserted, it is done as a result of an unsuccessful search.
    - We can use the cost of an unsuccessful search to compute the average cost of a successful search.
    - Since the fraction of empty cells is $1 - \lambda$, the number of cells we expect to probe is $1/(1- \lambda)$.

Probes vs. Load Factor



- **Dashed curves**-- linear probing
- **Solid curves**- random collision resolution
- **S**-successful
- **U**-unsuccessful
- **I**-insert

- What it is saying is that the linear probing is not a very good method to handle collision.

Notes

- If $\lambda = 0.75$, then the formula above indicates that 8.5 probes are expected for an insertion in linear probing.
- If $\lambda = 0.9$, then 50 probes are expected.
- This compares with 4 and 10 probes for the respective load factors if clustering were not a problem.
- We see from these formulas that linear probing can be a bad idea if the table is expected to be more than half full.
- If $\lambda = 0.5$, however, only 2.5 probes for insertion and only 1.5 probes are required for a successful search.

Quadratic Probing

- Quadratic probing avoid the primary clustering problem of linear probing.
- If there is a collision at hash address H, the method call **quadratic probing** looks in the table at locations h+1, h+4, h+9, …, that is, at locations $h + i^2$ (mod hashsize) for i=1, 2, ...
- This reduces clustering, but it is not obvious that it will probe all locations in the table, and in fact <u>it does not</u>.

Observation

- Theorem-- If quadratic probing is used and the table size is prime, then a new element can always be inserted if the table is at least half empty. (see book for more details)

Example

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Notes

- If the table is even one more than half full, the insertion could fail (although this is extremely unlikely).
- It is also crucial that the table size be prime.
- If the table size is not prime, the number of alternate locations can be severely reduced.
- Standard deletion cannot be performed in a closed hash table, because the cell might have caused a collision to go past it.
- Closed hash tables require lazy deletion.

About Lazy Deletion

- Deletion in a hash table is not an easy task. One method to delete an entry is to invent another **special key**, to be placed in any deleted position.
- This special key would indicate that this position is free to receive an insertion when desired but that is **<u>should not</u>** be used to terminate the search for some other item in the table.

Key-Dependent Increments

- Rather than having the increment depend on the number of probes already made, we can let it be some simple function of the key itself.
- For example, we could truncate the key to a single character and use its code as the increment.

Random Probing

- Use a pseudorandom number generator to obtain the increment.
- The generator used should be one that always generates the same sequence provided it starts with the same seed.
- This method is excellent in avoiding clustering, but is likely to be slower than the others.

Double Hashing

- For double hashing, one popular choice is $f(i) = i * h_2(x)$.
- We apply a second hash function to x and probe at a distance $h_2(x)$, 2 $h_2(x)$, . . ., and so on.
- A poor choice of $h_2(x)$ would be disastrous.
- The function must never evaluate to zero.
- Make sure all cells can be probed.

Double Hashing

- For instance, the obvious choice $h_2(x) = x \bmod 9$ would not help if 99 were inserted into the input in the previous examples.
- A function such as $h_2(x) = R - (x \bmod R)$, with R a prime smaller than H_SIZE, will work well.
- One may continue to perform triple hashing, and so on.

Example

$h_2(x) = R - (x \bmod R)$, $R = 7$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Rehashing

- When the table gets too full, the running time for the operations will deteriorate, specially when there are too many removals intermixed with insertions.
- Solution
  - Build another table that is about twice as big (with associated new hash function).
  - Scan down the entire original hash table, computing the new hash value for each element and inserting it in the new table.

Example

- Table size = 7
- Insert 13, 15, 24, and 6.
- h(x) = x mod 7.

0	6
1	15
2	
3	24
4	
5	
6	13

Closed Hash Table, Insert 23

0	6
1	15
2	23
3	24
4	
5	
6	13

- After 23 is inserted, the resulting table will be over 70% full.
- A new table is created with size = 17 since this is the first prime that is twice as large as the old table size.
- The new hash function is then h(x) = x mod 17.

After Rehashing

The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table.

- The running time is O(n).
- It is expensive.
- It should not be done so frequently.

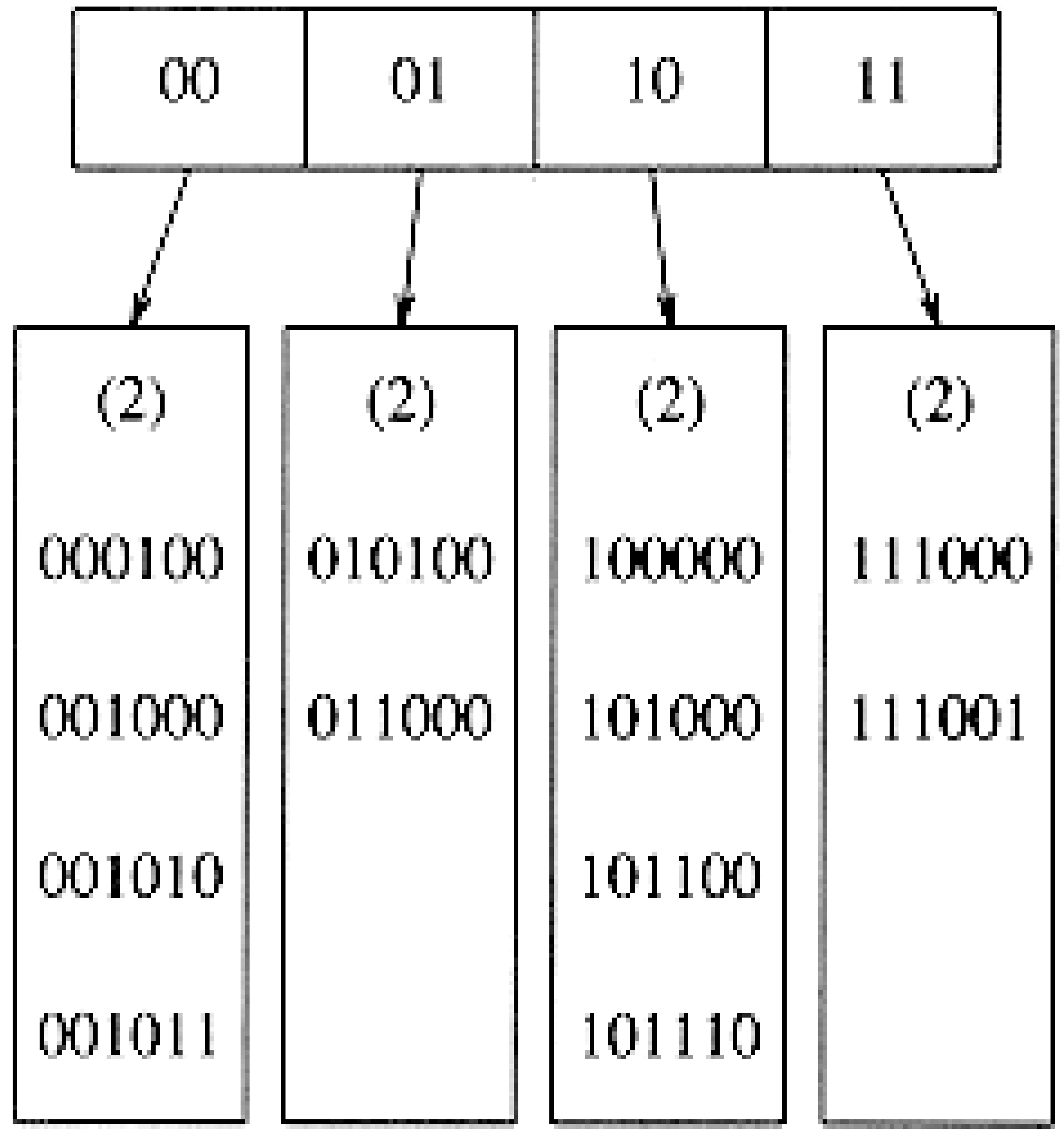
0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Extendible Hashing

- What happens when the amount of data is too large to fit in main memory and must be stored on the disk?
- How can we minimize the disk access?

- Suppose that our data consists of several 6 bit integers.
- The root of the tree contains 4 pointers determined by the leading two bits of the data.

Example


00
01
10
11
(2)
000100
001000
001010
001011
(2)
010100
011000
(2)
100000
101000
101100
101110
(2)
111000
111001

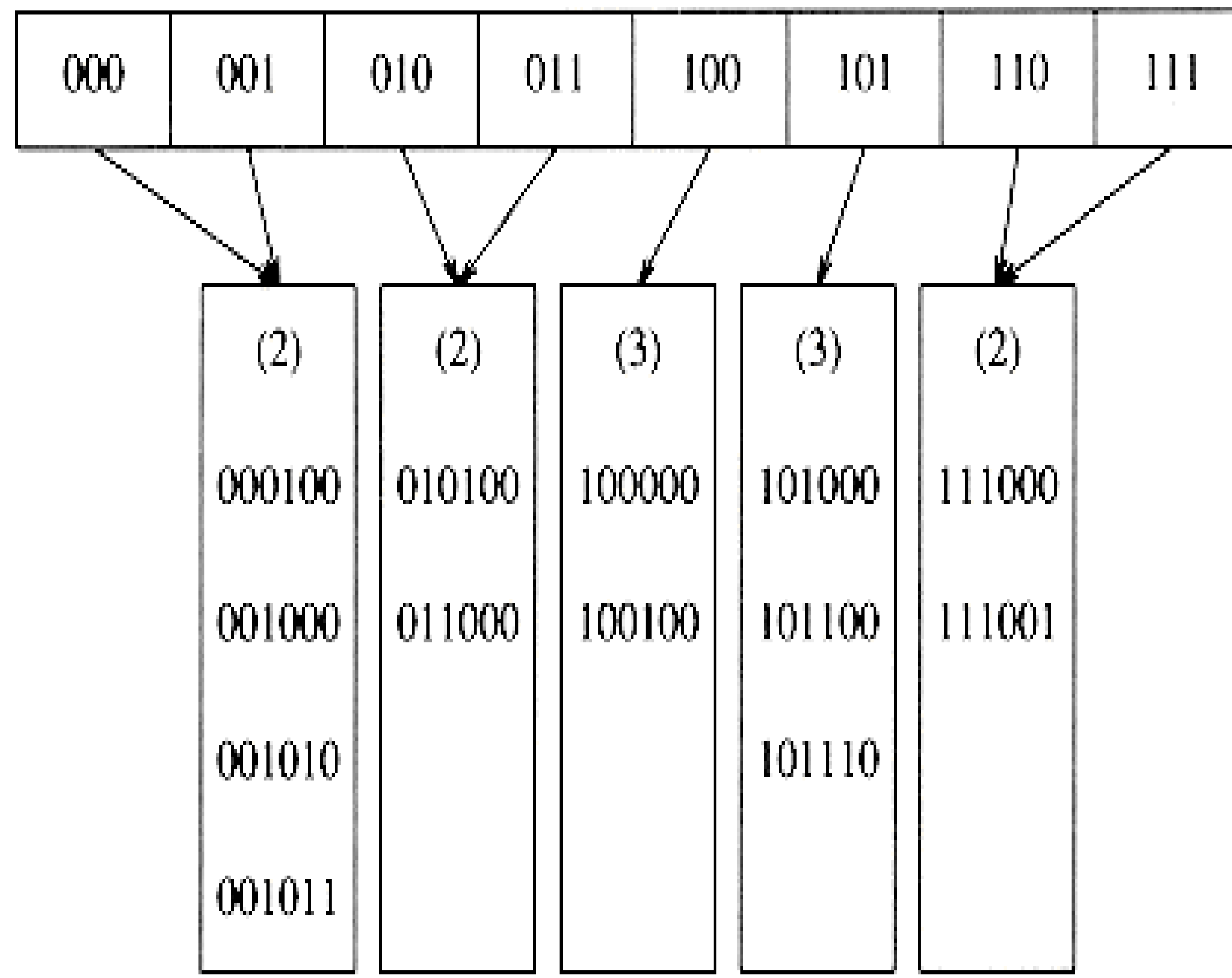
Extendible Hashing

- Each leaf has up to m = 4 elements.
- D represents the number of bits used by the root, which is sometimes known as the **directory**.
- The number of entries in the directory is thus 2D dl (the number of leading bits that all the elements of some leaf l have in common.
- dl will depend on the particular leave.

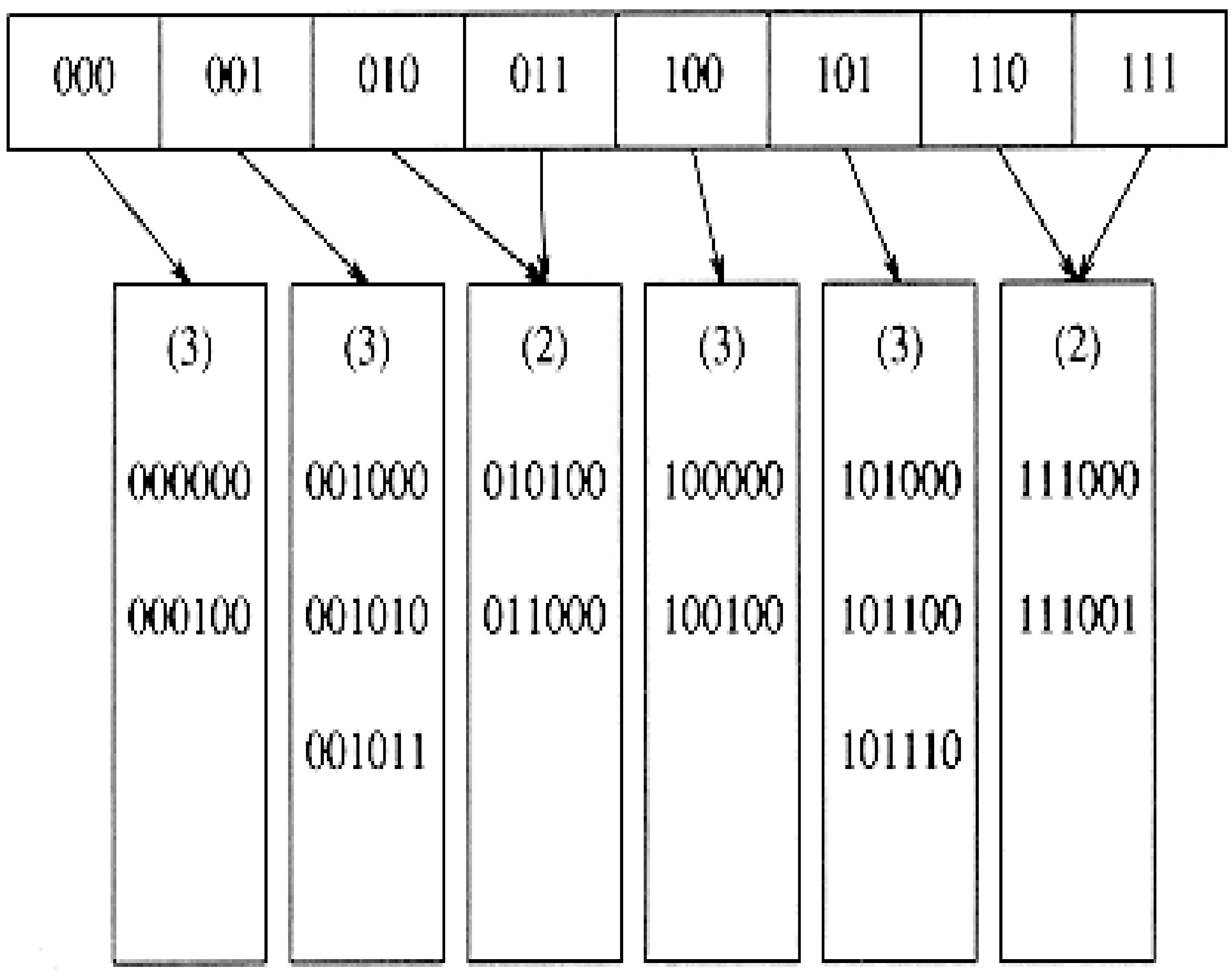
Example, insert 100100

This will go into the third leaf and cause a split.

Now, the leaves are now determined by the first 3 bits.



Example, insert 000000



Notes

- It is possible that several directory splits will be required if the elements in a leaf agree in more than D+1 leading bits.
  - For example, 111010, 111011, and 111100 are inserted, the directory size must be increased to 4.
- The possibility of duplicate keys. This algorithm does not work when there are more than m duplicates.
- It is important for the bits to be fairly random.

Summary

- Hash tables can be used to implement the **insert** and find operations in constant average time.
- It is especially important to pay attention to details such as load factor when using hash tables.
- It is also important to choose the hash function carefully when the key is not a short string or integer.

Summary

- For open hashing, the load factor should be close to 1.
- For closed hashing, the load factor should not exceed 0.5, unless this is completely unavoidable.
- Using a hash table, it is not possible to find the minimum element.
- It is not possible to search efficiently for a string unless the exact string is known.

Summary

- Compilers use hash tables to keep track of declared variables in source code. The data structure is known as a symbol table.
- A hash table is useful for any graph theory problem where the nodes have real names instead of numbers.
- A third common use of hash tables is in programs that play games.
- Another use of hashing is in online spelling checkers.