MALLA REDDY UNIVERSITY

MR22-1CS0104

ADVANCED DATA STRUCTURES

II YEAR B.TECH. (CSE) / II – SEM

Unit-3

Hashing: Hash Tables - Hash Functions - Collision-Handling Schemes - Separate Chaining - Open Addressing – Linear Probing - Quadratic Probing- Double Hashing, Rehashing.

String Matching: The naive string-matching algorithm - Knuth-Morris-Pratt algorithm. Binary Tries, Compressed Binary Trie.

Hashing

Hashing:

- Hashing is an approach in which the time complexity of insert, delete and search operations have constant time complexity i.e., **O(1).**

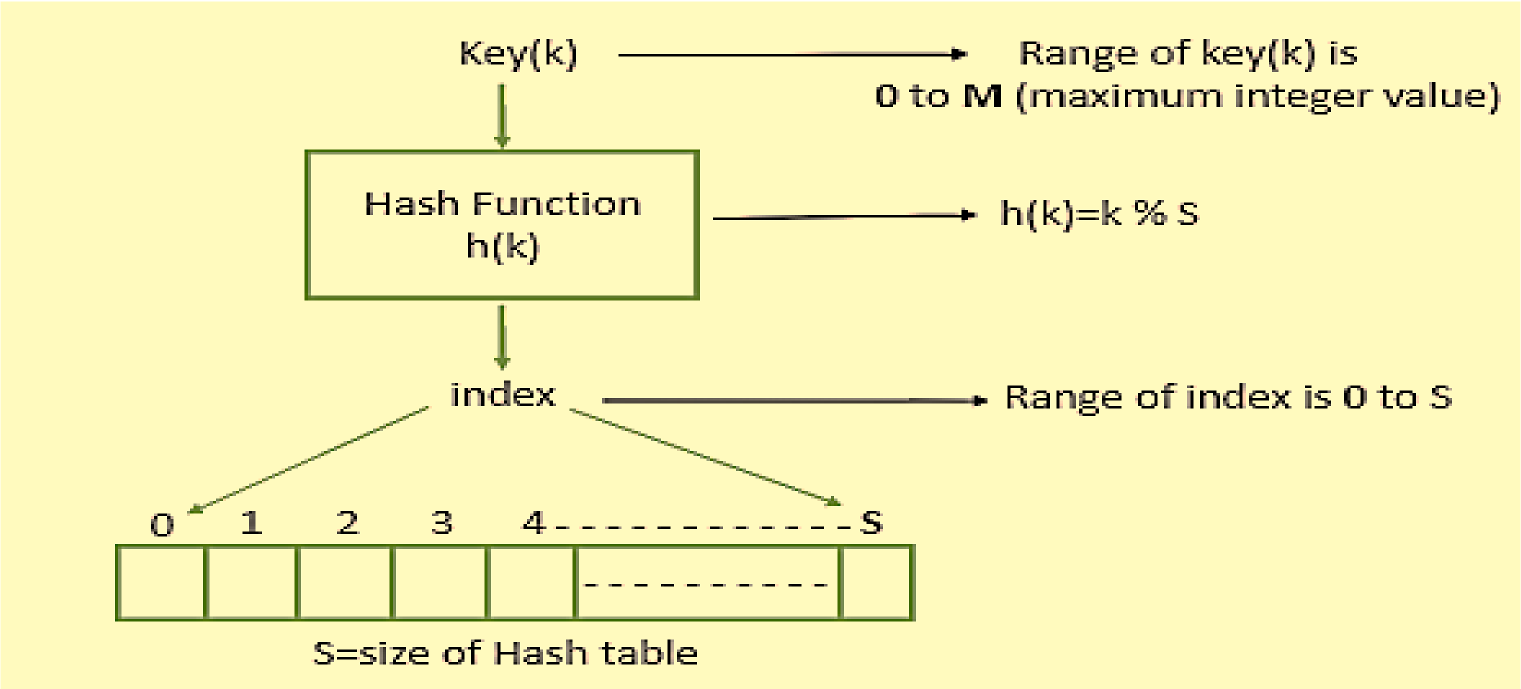
Hash Table:

- We use a table called **Hash Table** to store the data.
- The size of the hash table is less than the range of the actual data.
- the data values are inserted into the hash table based on the **Hash(key)** value.

Hashing

Hash Table:

- is similar to an array, but the data is added to the hash table according to the index obtained after performing the hashing function.



Hashing

Hash(key) value:

- index of the hash table at which the actual data is stored.
- provides the mapping of the actual data and the index of the hash table.

Hash function:

- a function which takes the key i.e the actual data to be inserted as input and gives the Hash(key) value as output.
- the output of the hash function is always within the range of the index of the hash table.

Hashing

Hash function types:

Standard hash functions are,

- Division Method.
- Mid Square Method.
- Folding Method.
- Multiplication Method.

Hashing

Division Method:

- simplest and easiest method used to generate a hash value.

h(K) = k mod M

(where k = key value and M = the size of the hash table)

Example:

k = 1320 , M = 11

h (1320) = 1320 mod 11 = 0

Mid Square Method:

The steps involved in computing this hash method are:

- Squaring the value of k (like k*k)
- Extract the hash value from the middle r digits.

h(k) = h(k x k) , (where k = key value)

Example:

Element (k) = 87431 ⇒ k2 = 7644179761

The possible 3 digit mids of 7644179761 are **417** or **179**

Hashing

Folding Method:

The steps involved in computing this hash method are:

- k should be divided into a specific number of parts
- Add each component separately.

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(k)= s mod M

Example:

k is 452378912 and the table size (i.e., M = 10).

- Therefore, a = 452, b = 378, c = 912.
- h(k) = (a + b + c) mod M i.e., H(452378912) = 1742 mod 10 = 2.

Multiplication Method:

h(k) = floor (M (kA mod 1))

Example:

k = 1234 , A = 0.35784 , M = 100 **(0<A<1)**

- h (1234) = floor [100(1234 x 0.35784 mod 1)] = **57**

Collision & Collision-Handling

- There must be some keys that hash into the same slot is called **collision**.
- During collision, a newly inserted key maps to an already occupied slot in hash table and must be handled using some collision handling technique.

There are mainly two methods to handle collision:

- Open Addressing (Closed Hashing)
  - linear probing or
  - quadratic probing or
  - double hashing.
- Separate Chaining (Open Hashing).

Collision-Handling Schemes

Linear probing:

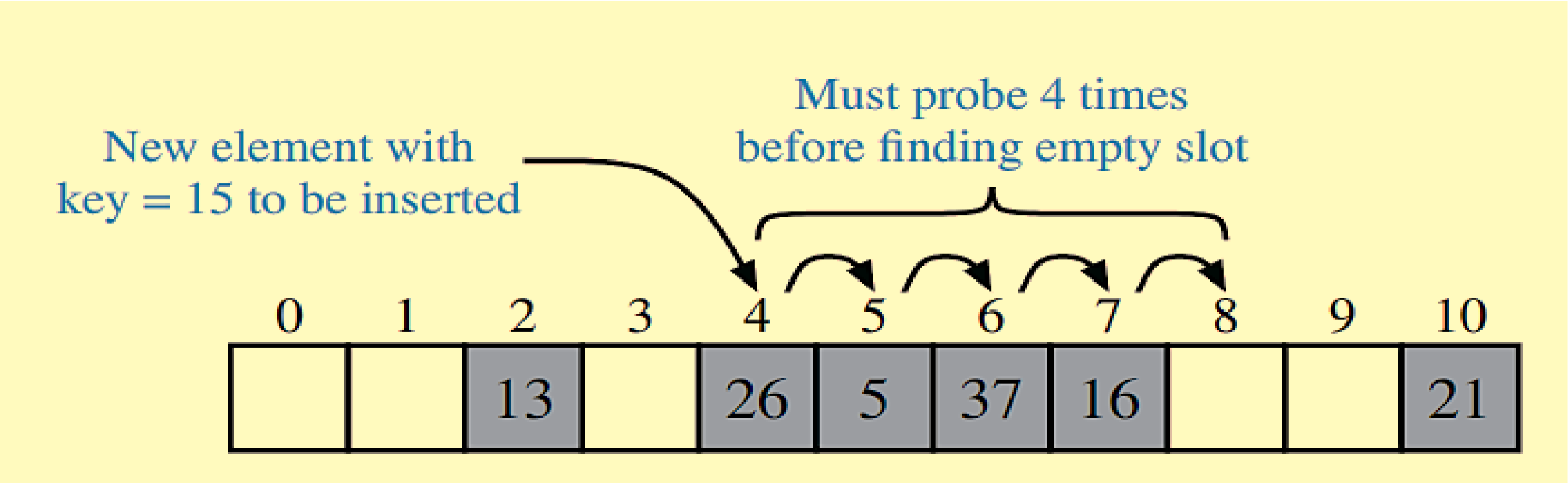
- the interval between successive probes is fixed (usually to 1)
- The probing sequence for linear probing will be:

index = index % S

index = (index + 1) % S

index = (index + 2) % S

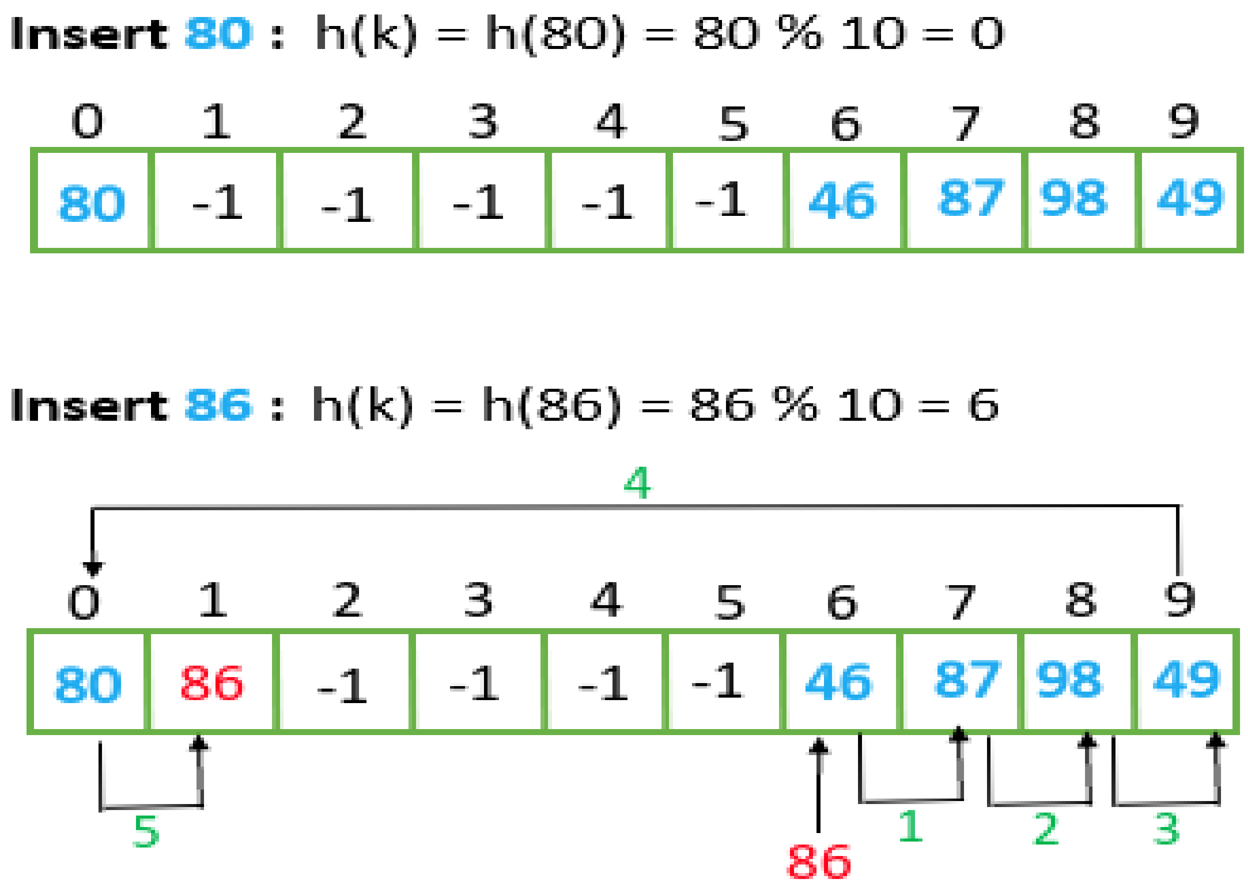
index = (index + 3) % S



Collision-Handling Schemes

Problems of linear probing:

- **Primary Clustering** - create long runs (cluster) of filled slots near the hash position of keys.



Collision-Handling Schemes

Quadratic probing:

- When the slot is already occupied, start traversing until an unoccupied slot is found.
- The interval between slots is computed by adding the successive value of an arbitrary polynomial to the original hashed index.
- The probing sequence for quadratic probing will be:

$$\textbf{index} = \textbf{index} \,\%\, \textbf{S}$$

$$\textbf{index} = (\textbf{index} + 1^2) \,\%\, \textbf{S}$$

$$\textbf{index} = (\textbf{index} + 2^2) \,\%\, \textbf{S}$$

$$\textbf{index} = (\textbf{index} + 3^2) \,\%\, \textbf{S}$$

Collision-Handling Schemes

Quadratic probing:

Let h(k) = k % S   where S = 10 (table size)

Insert 45 : h(k) = h(45) = 45 % 10 = 5

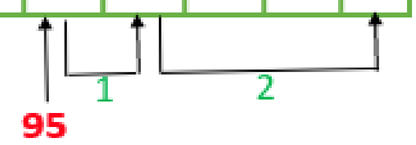
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	45	-1	-1	-1	-1

Insert 86 : h(k) = h(86) = 86 % 10 = 6

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	45	86	-1	-1	-1

Insert 95 : h(k) = h(95) = 95 % 10 = 5   If collision occurred $h(k)=(index + i^2)$ % 10 where i=1,2,3............n

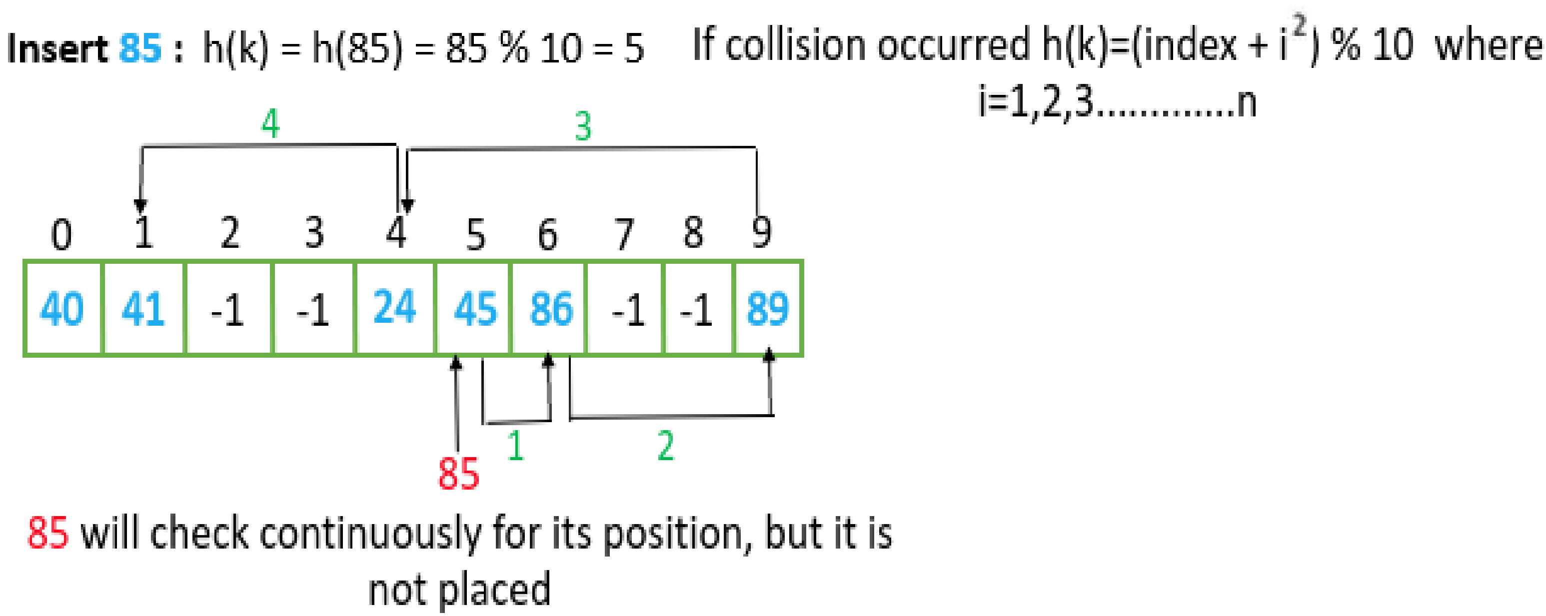
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	45	86	-1	-1	95



Collision-Handling Schemes

Problems of quadratic probing:

- **Secondary Clustering -** tendency to create long runs of filled slots away from the hash position of keys.
- Secondary clustering is less severe than primary clustering.



Collision-Handling Schemes

Double hashing:

- The interval between probes is computed by using a second hash function.
- The probing sequence will be:

index = (index + 1 * Hash2(k)) % S

index = (index + 2 * Hash2(k)) % S

index = (index + 3 * Hash2(k)) % S

Choosing second hash function:

A popular second hash function is :

Hash2(key) = PRIME – (key % PRIME)

where PRIME is a prime smaller than and close to the hash table size S.

Collision-Handling Schemes

Double hashing:

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)

Hash1(19) = 19 % 13 = 6

Hash1(27) = 27 % 13 = 1

Hash1(36) = 36 % 13 = 10

Hash1(10) = 10 % 13 = 10

Hash2(10) = 7 – (10%7) = 4

(Hash1(10) + 1*Hash2(10))%13= 1

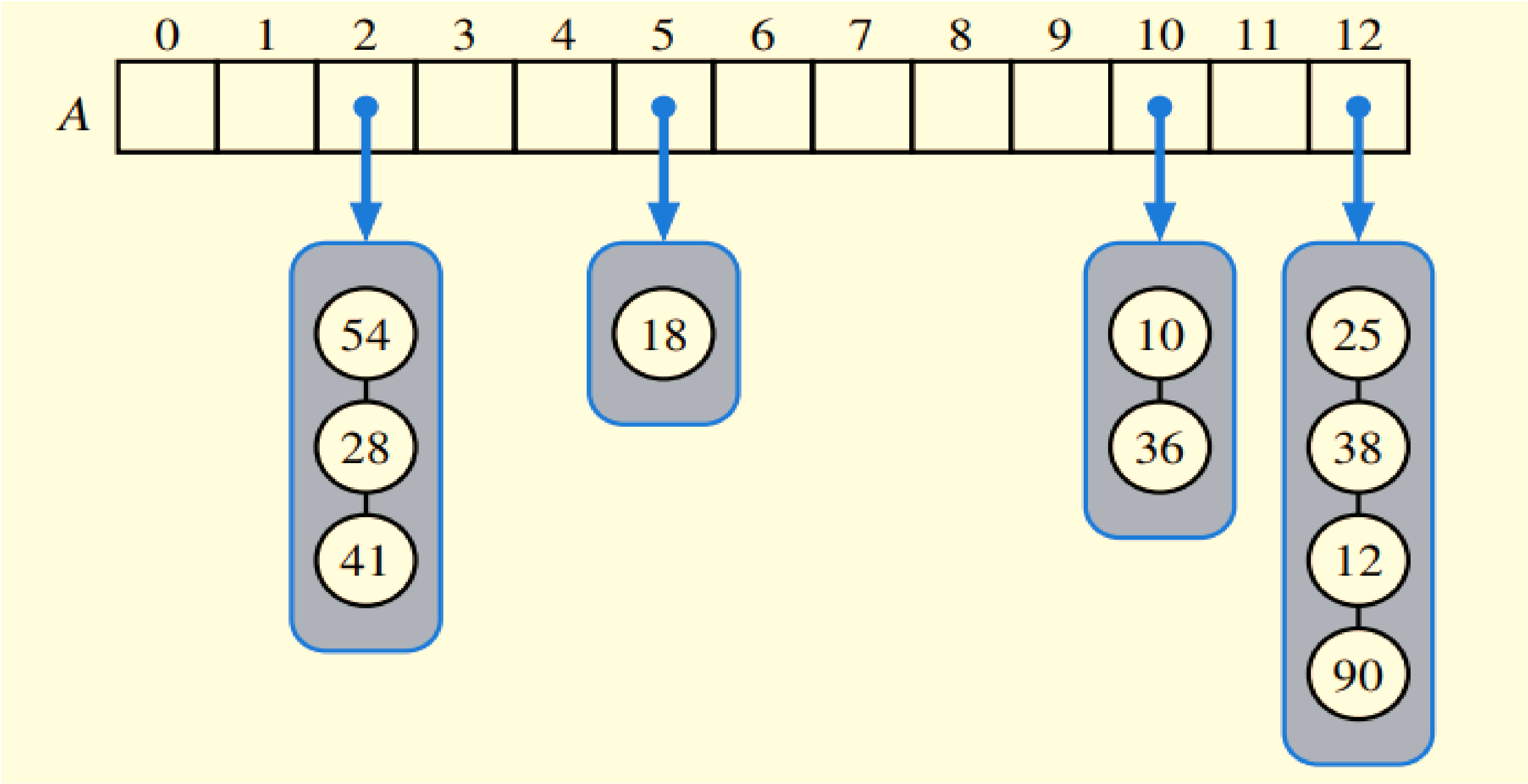
(Hash1(10) + 2*Hash2(10))%13= 5

Collision

Collision-Handling Schemes

Separate chaining:

- Each cell of hash table point to a linked list of records that have same hash function value.



Collision-Handling Schemes

Advantages of separate chaining:

- Easier to implement.
- Hash table never full up.
- Less sensitive to the hash function or load factors.

Disadvantages of separate chaining:

- Cache performance of chaining is not good as keys are stored using linked list.
- Wastage of Space.
- If the chain becomes long, then search time can become O(n) in worst case.

Rehashing

- When the load factor increases to more than its predefined value, the complexity increases.
- The size of the hash table increased and all the values are hashed again and stored in the new double-sized table to maintain a low load factor and low complexity.
- The time complexity for the rehashing is O(n) and the cost is shared by preceding n/2 insertions.

String Matching

- Pattern matching is defined as follows:

Given two strings:

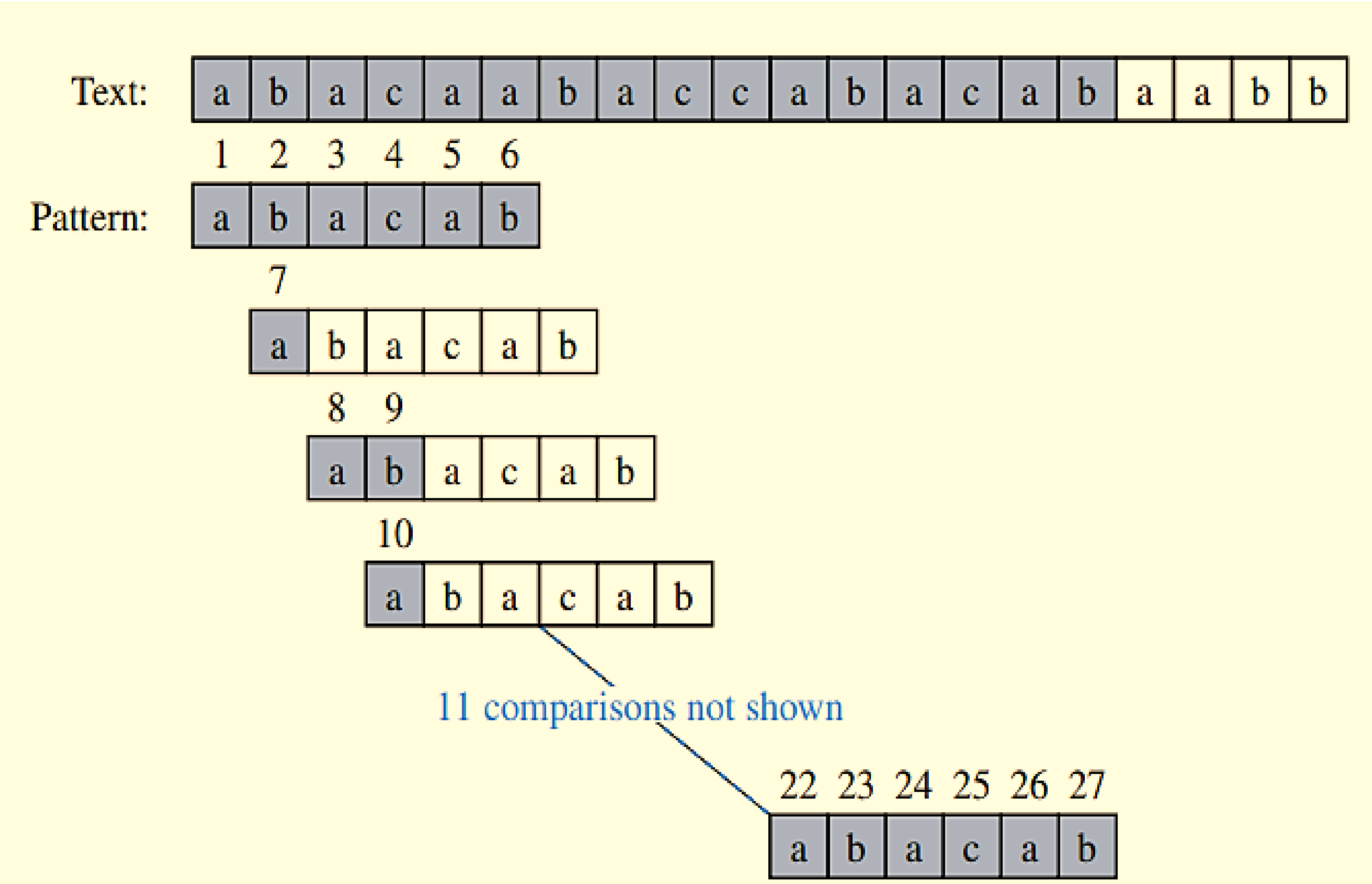
1. text and
2. pattern,

print all the occurrences of pattern in the text.

The naive string-matching algorithm (Brute Force):

- straight forward.
- For every position in the text, considering it as the starting position of the pattern, print the position if a match occurs.
- Easy to understand but it can be too slow in some cases.

Naïve String Matching



String Matching

```
void naive_approach(text[], pattern[]){
        for (int i = 0; i < n-m; i++) {
                int j = 0;
                while (j < m && t[i+j] == p[j])
                {       j++;    }
                if (j == m)     return i;       }
        System.out.println(“No match found”);
        return -1;
```

If the length of the **text is n** and the length of the **pattern m**, in the worst case the naive approach may take as much as **(n * m)** iterations to complete the task.

String Matching

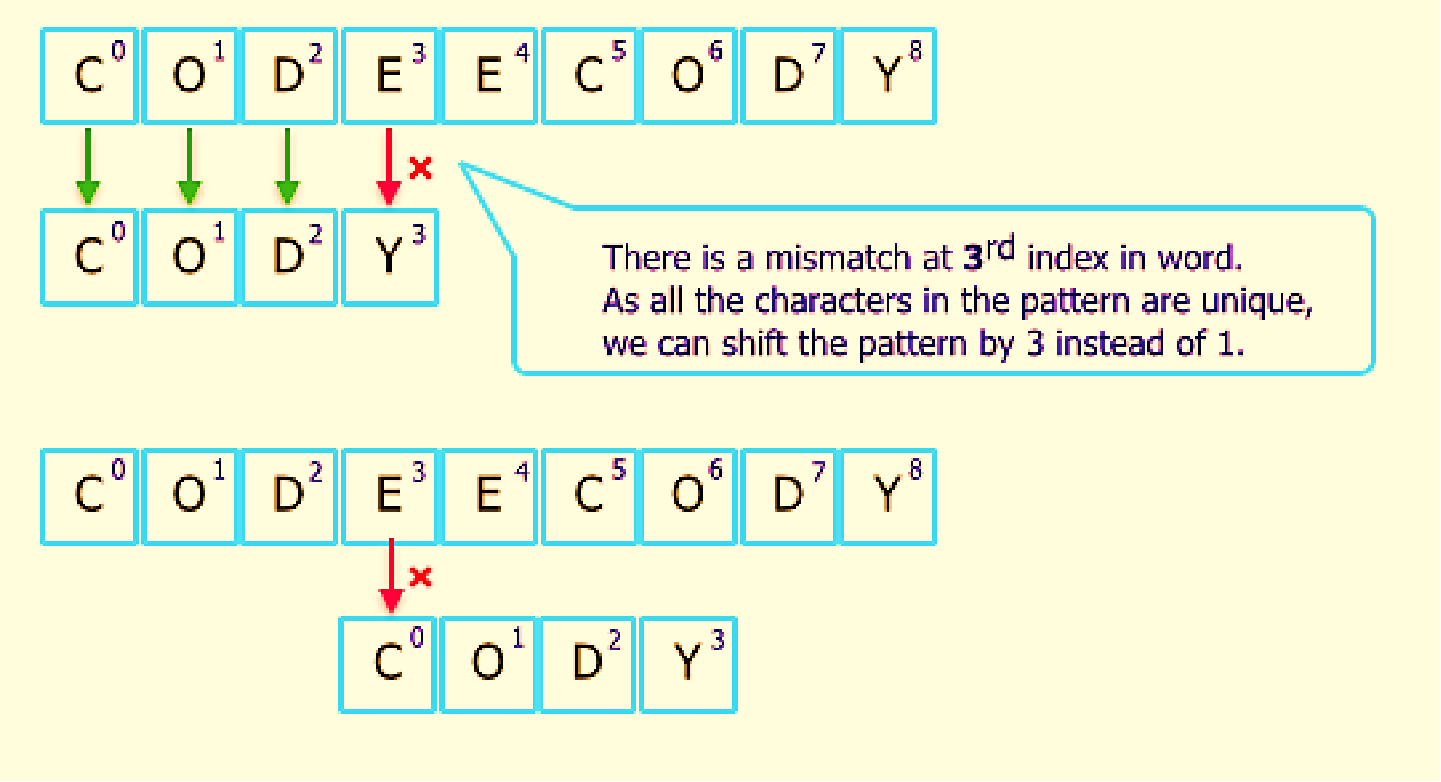
Knuth-Morris-Pratt Algorithm (KMP Algorithm):

- Solving the pattern matching problem in linear time was a challenge.
- KMP algorithm was the first linear time complexity algorithm for string matching.
- The KMP matching algorithm uses degenerating property of the pattern and improves the worst case complexity to O(N).
- when we detect a mismatch after several matches, we are already looking up some of the characters of the next window.
- This information can be used to avoid matching the characters that we know will anyway match.

String Matching

Case-1 (Trivial): When all the pattern to be matched has all unique characters.

Text = "CODEECODY“ and **Pattern = "CODY"**



String Matching

Case-2: When all the pattern or parts of pattern have common suffix and prefix

For a given string, a proper prefix is prefix with whole string not allowed.

For example for a string **"ABC"** :

Prefixes are : "", "A", "AB", "ABC".

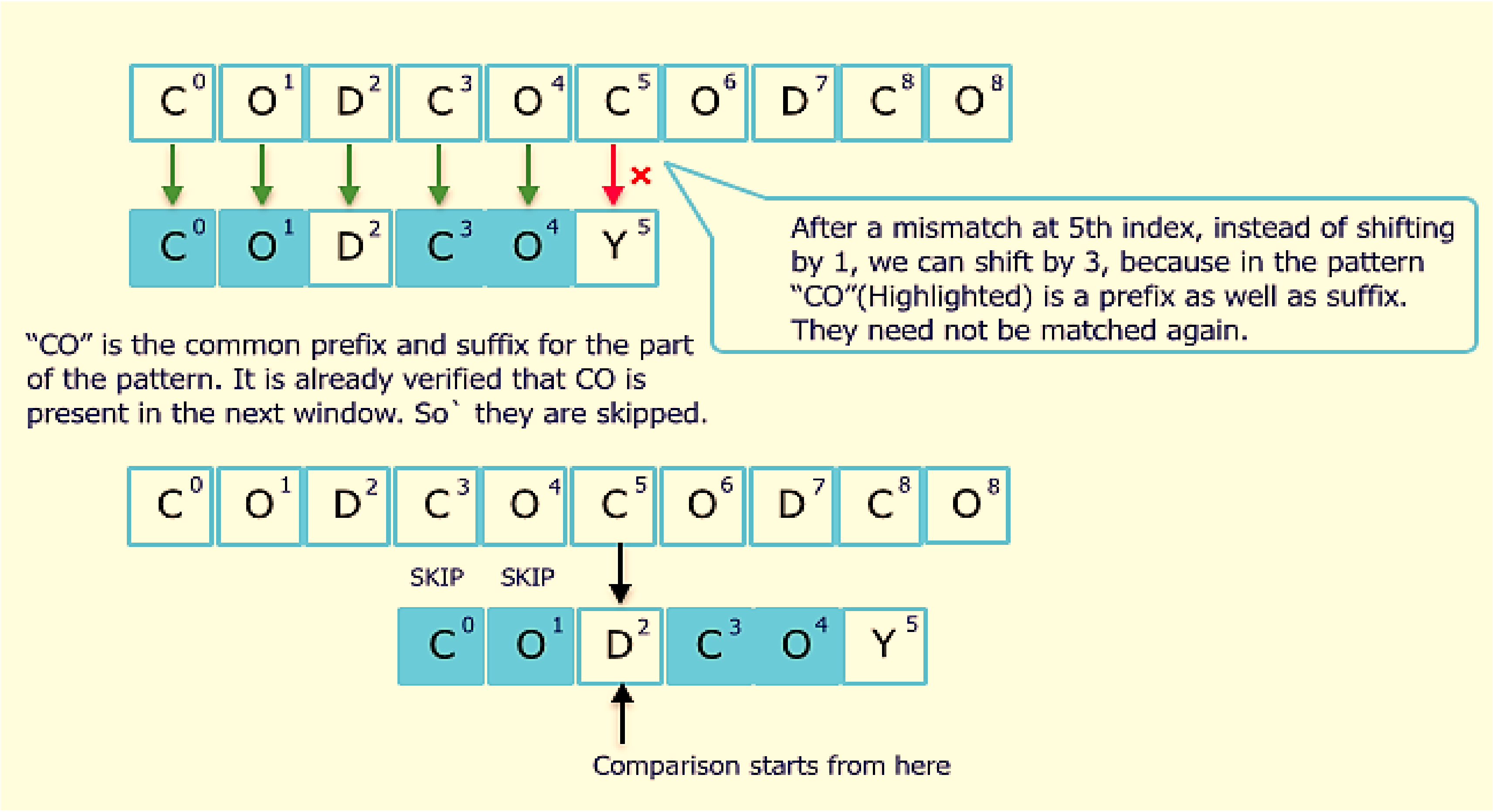
Proper prefixes are : "", "A", "AB".

Suffixes are : "", "C", "BC", "ABC".

String Matching

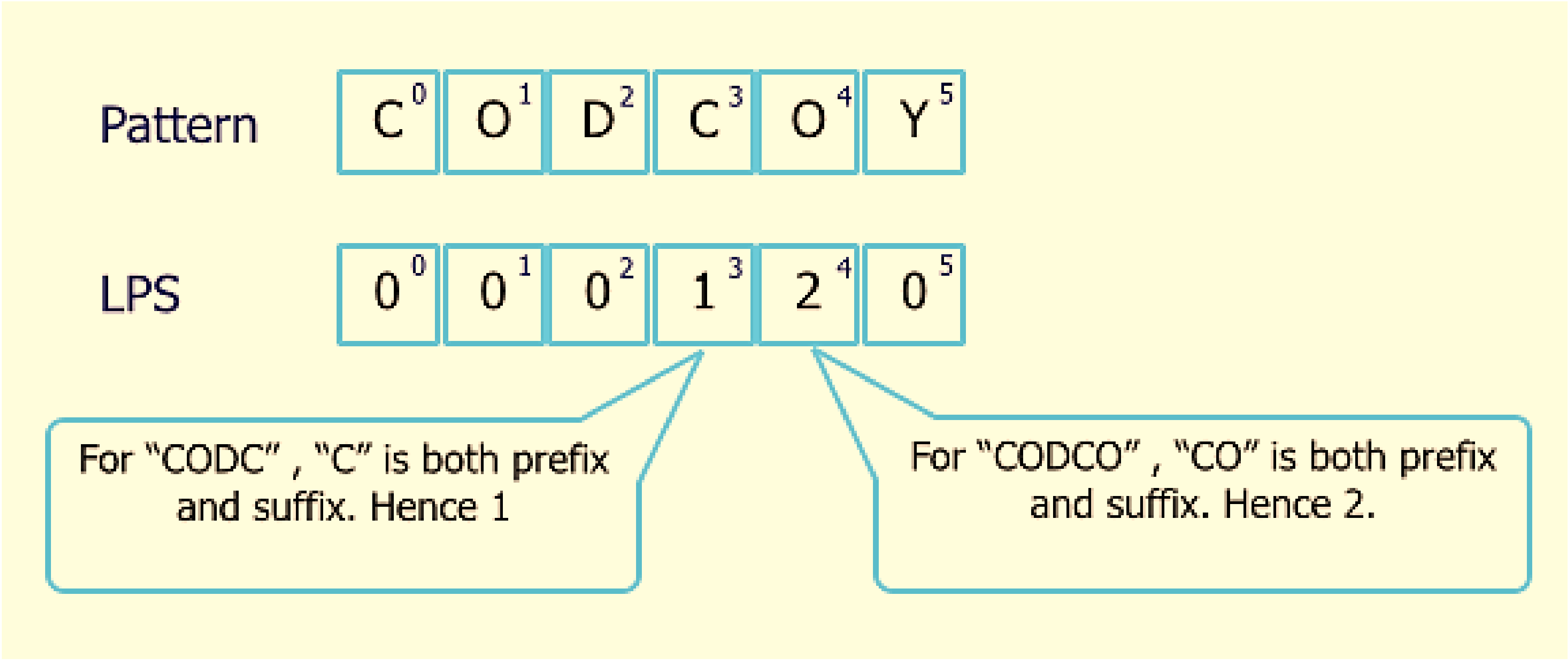
Case-2: When all the pattern or parts of pattern have common suffix and prefix

Text = "CODCOCODCO" and **Pattern = "CODCOY"**



String Matching

- Finding the common prefix and suffix for each and every substring of the pattern will help to skip the unnecessary matches and increase the efficiency of the algorithm.
- Find LPS(Longest Proper Prefix) for the given pattern.



String Matching

```
public static int findKMP(char[ ] text, char[ ] pattern) {
    int n = text.length;
    int m = pattern.length;
    if (m == 0) return 0;
    int[ ] fail = computeFailKMP(pattern);
    int i = 0; int j = 0;
    while (i < n) {
    if (text[i] == pattern[j]) {
    if (j == m − 1) return i − m + 1;
    i++; j++;
    } else if (j > 0)
    j = fail[j−1];
    else
    i++;   }
    return −1; }
```

String Matching

```
private static int[ ] computeLPS(char[ ] pattern) {
    int m = pattern.length;
    int[ ] Lps = new int[m];
    int j = 1;
    int k = 0;
    while (j < m) {
    if (pattern[j] == pattern[k]) {
    Lps[j] = k + 1;
    j++;    k++; }
    else if (k > 0)
    k = Lps[k−1];
    else                          // no match found starting at j
    j++;}
    return fail;    }
```

Tries

- A tree-based data structure for storing strings in order to support fast pattern matching.
- The main application for tries is in information retrieval.
- The primary query operations that tries support are pattern matching and
- prefix matching.
- The latter operation involves being given a string X, and looking for all the strings in S that being with X.

Binary Tries

- Binary tries are binary trees
- Unlike a BST in a binary trie, only leaf nodes hold keys.
- Search in a binary trie is guided by comparing keys one bit at a time.
- It has two kinds of nodes: branch nodes and element nodes.
- **A branch node** has 2 data members - LeftChild and RightChild.
- **An element node** has the single data member data.
- Keys are represented as a sequence of bits.
- Key must not be prefix of another key in terms of bit representation.

Binary Tries

The basic search algorithm to find key in a binary trie:

Step-1: Set currentNode = root and i = 0.

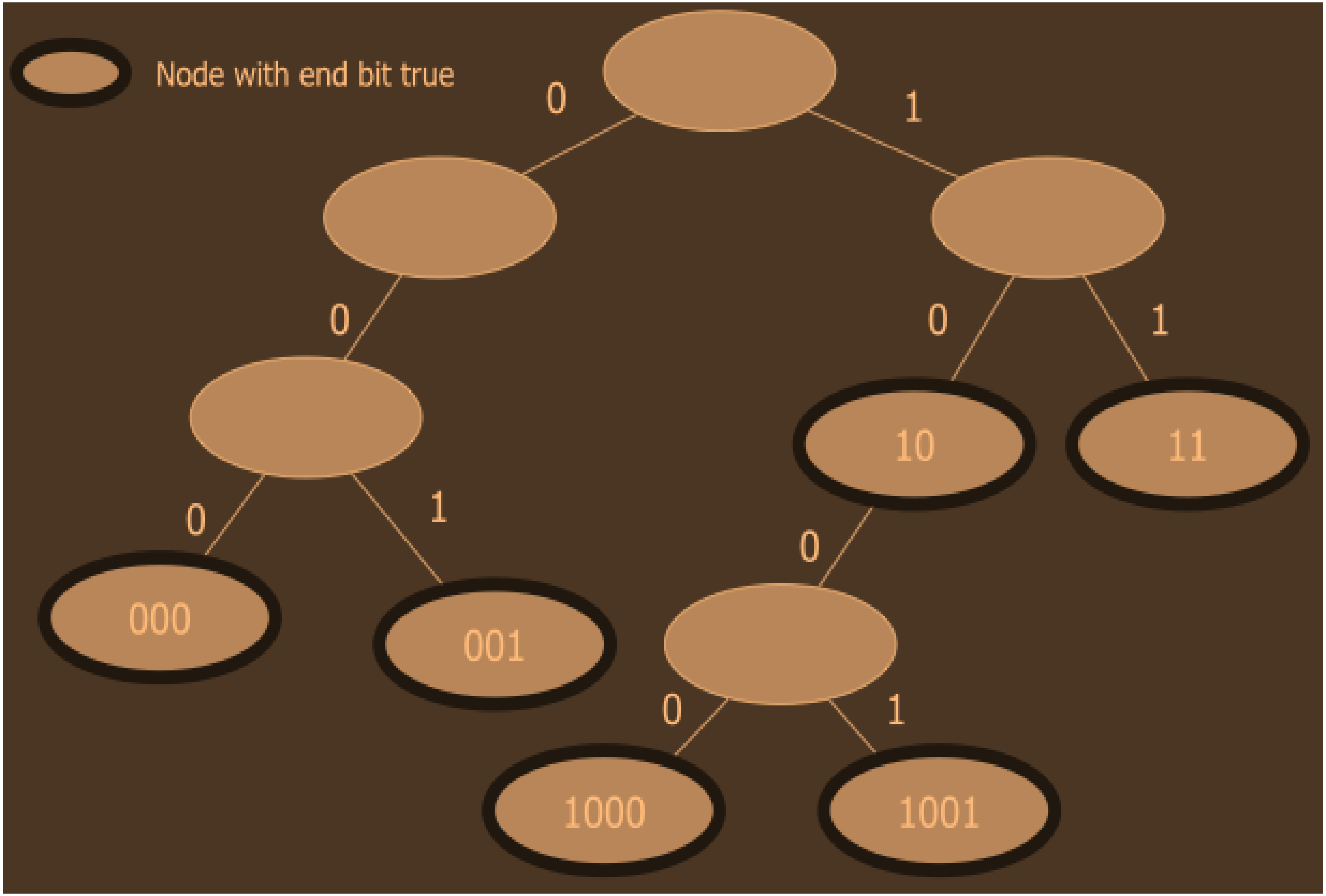
Step-2: If currentNode is NULL or i > number of bits in key then return "not found".

Step-3: If currentNode is a leaf, and i == number of bits in key, return "found".

Step-4: Look at the value of the ith bit in key. If 0, set currentNode = currentNode.left; else set currentNode = currentNode.right

Step-5: Set i = i+1 and go to Step-2.

Binary Tries



Binary Tries

Example:

Consider the following 6 keys with values as shown,

10

11

000

001

1000

1001

We will insert them in that order into an initially empty binary trie.

Binary Tries

Example:

Consider the following 8 keys with values as shown

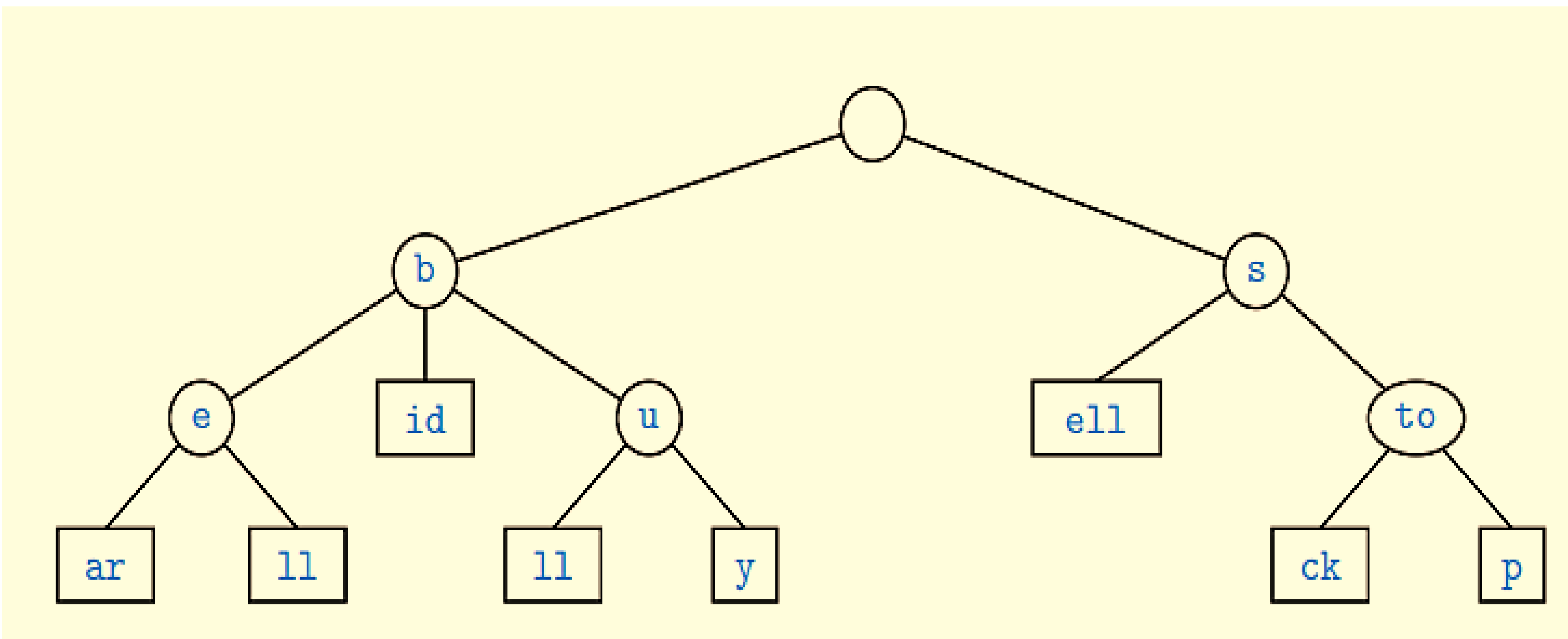
A	**00001**
S	**10011**
E	**00101**
R	**10010**
C	**0001**
H	**101**
I	**001**
N	**10**
G	**000**

Compressed Tries

- We can transform T into a compressed trie by replacing each redundant chain $(v0,v1)\cdots(vk-1,vk)$ of $k \geq 2$ edges into a single edge $(v0,vk)$

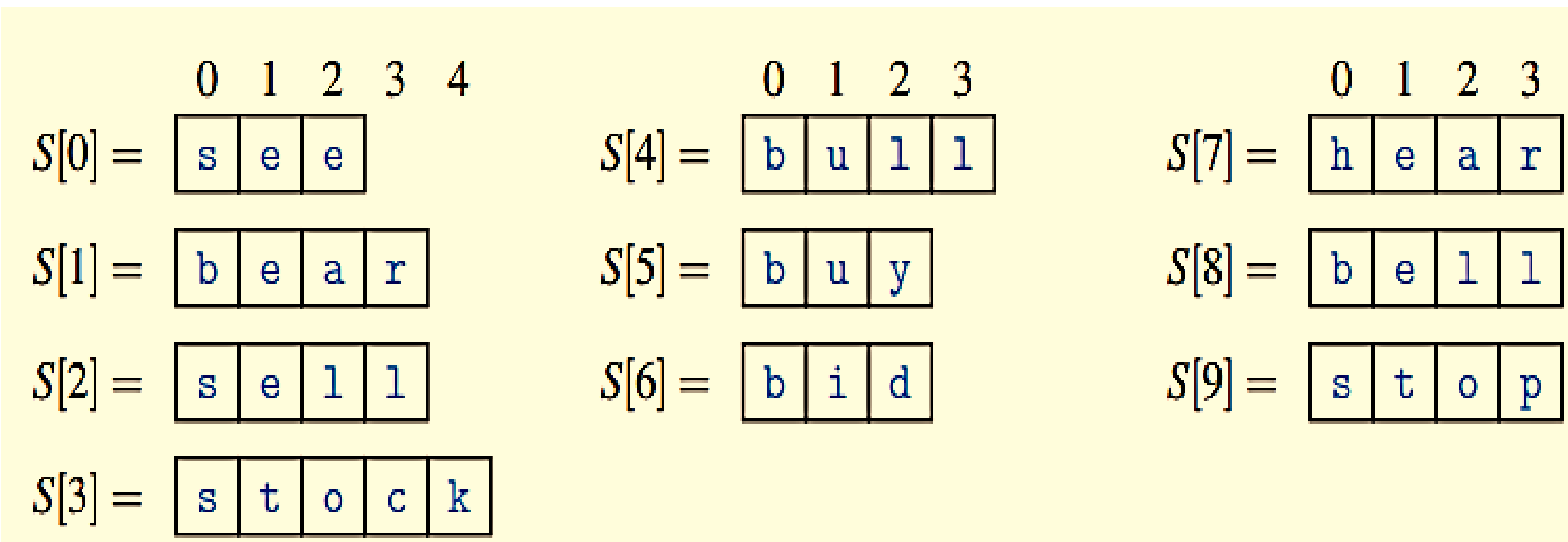
Example:

Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}



Compressed Tries

- a compressed trie is truly advantageous only when it is used as an auxiliary index structure
- It is not required to actually store all the characters of the strings in the collection.



Compressed Tries

- This additional compression scheme allows us to reduce the total space for the trie itself from O(n) for the standard trie to O(s) for the compressed trie, where n is the total length of the strings in S and s is the number of strings in S.