Advanced Data Structures-DAY-3

By

HASHING

Hashing

- Mathematical concept
  - To define any number as set of numbers in given interval
  - To cut down part of number
  - Used in discreet maths, e.g graph theory, set theory
  - Used in Searching technique
  - Used in encryption methods

Applications of Hashing

- Compilers use hash tables to keep track of declared variables
- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different

Hash Functions and Hash Tables

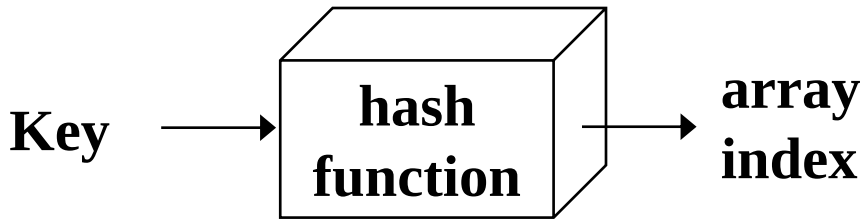
- Hashing has 2 major components
  - Hash function h
  - Hash Table Data Structure of size N
- A hash function h maps keys (a identifying element of record set) to hash value or hash key which refers to specific location in Hash table
- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

Hash Functions and Hash Tables

- A hash table data structure is an array or array type ADTof some fixed size, containing the keys.
- An array in which records are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*



- ***Hashed key***: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array
- Contains the main advantages of both Arrays and Trees
- Mainly the topic of hashing depends upon the two main factors / parts

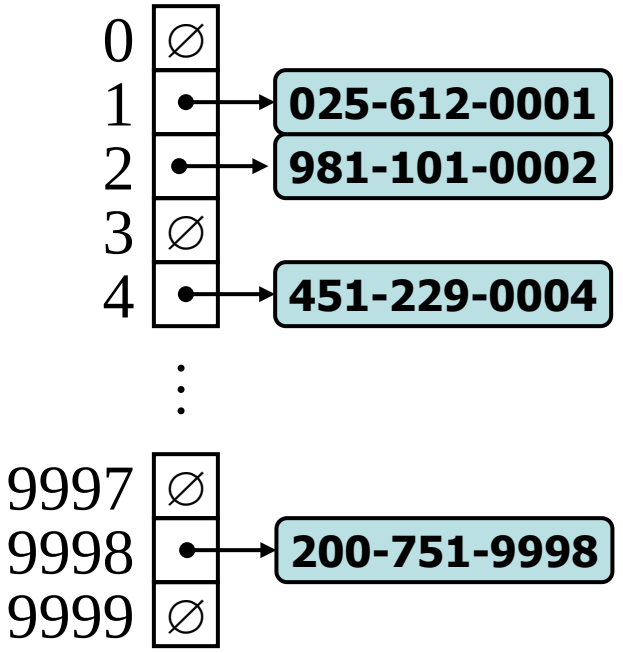
  (a) Hash Function (b) Collision Resolution
- Table Size is also an factor (miner) in Hashing, which is 0 to tablesize-1.

Table Size

- Hash table size
  - Should be appropriate for the hash function used
  - Too big will waste memory; too small will increase collisions and may eventually force *rehashing* (copying into a larger table)

Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- The actual data is not stored in hash table
- Pin points the location of actual data or set of data
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x) = \text{last four digits of } x$



Hash Function

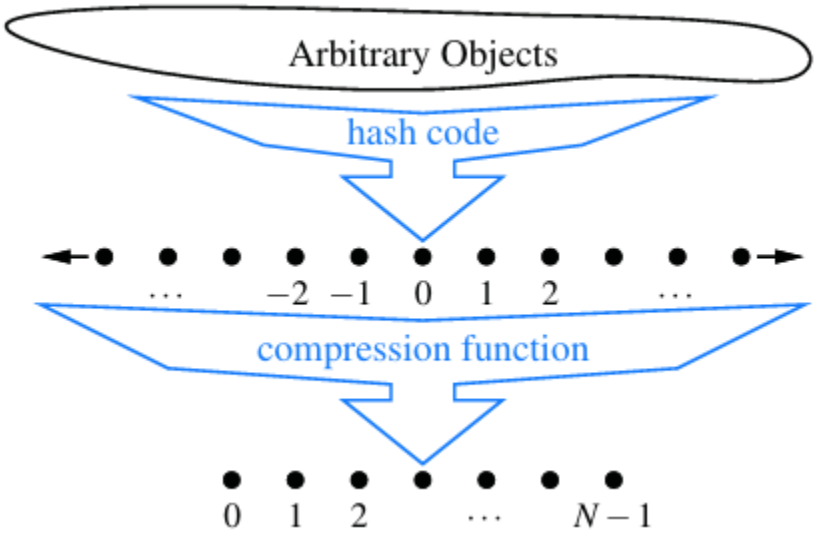
- The mapping of keys into the table is called ***Hash Function***

- A hash function,
  - Ideally, it should distribute keys and entries evenly throughout the table
  - It should be easy and quick to compute.
  - It should minimize *collisions*, where the position given by the hash function is already occupied
  - It should be applicable to all objects

- It is common to view the evaluation of a hash function, h(k), as consisting of two portions—

➢ a **hash code** that maps a key k to an integer,

➢ a **compression function** that maps the hash code to an integer within a range of indices, [0,N −1], for a bucket array.



- **HASH CODES :**

The first action that a hash function performs is to take an arbitrary key k in our map and compute an integer that is called the hash code for k; this integer need not be in the range [0,N −1], and may even be negative.

Hash codes are two types :

- Polynomial hash codes
- Cyclic-shift hash codes

Polynomial hash codes

- **Polynomial hash codes**, also known as **Polynomial Rolling Hash Function**, are a type of hash function that uses only multiplications and additions. They are often used in string hashing and are particularly useful in solving problems related to string comparison
- The polynomial hash function is defined as follows:
- **hash(s)=s[0]+s[1]∗p+s[2]∗p^2+...+s[n−1]∗p^n−1 mod m**
- where:
- s is the input string of length n.
- p and m are some positive integers.
- p is usually a prime number roughly equal to the number of characters in the input alphabet.
- m is a large prime number to reduce the probability of hash collisions.

Cyclic-shift hash codes

- **Cyclic Shift Hash Codes**, also known as **Cyclic Permutation Hash Codes**, are a variant of hash functions that use cyclic shifts instead of multiplication.
- They are often used in string hashing and are particularly useful in solving problems related to string comparison.
- The basic idea of Cyclic Shift Hash Codes is to generate a binary representation of the data (typically stored in a 32-bit integer) with a series of 32 0s and 1s, then to cyclically shift the digits of this binary number.
- If the data are the same, they should result in the same binary number, which will be shifted into the same resulting number when the hash code is computed.

compression function

- Compression function is one that minimizes the number of collisions for a given set of distinct hash codes.
- Compression function is of two types:
    1. Division method
    2. MAD method

1. Division Method

- Choose a number *m* larger than the number *n* of keys in *k*.
- The number *m* is usually chosen to be a prime no.
- The hash function *H* is defined as,

  H(k) = k(mod m) or H(k) = k(mod m) + 1
- Denotes the remainder, when *k* is divided by *m*
- 2nd formula is used when range is from 1 to m.

- Example:

  Elements are: 3205, 7148, 2345

  Table size: 0 – 99 (prime)

  m = 97 (prime)

  H(3205)= 4, H(7148)=67, H(2345)=17

- **For 2nd formula *add* 1 into the remainders**.

2.MAD Method

- A more sophisticated compression function, which helps eliminate repeated pat terns in a set of integer keys, is the **Multiply-Add-and-Divide (or "MAD")** method.
- This method maps an integer i to

$$[(ai+b) \bmod p] \bmod N$$

- where N is the size of the bucket array,
- p is a prime number larger than N, and
- a and b are integers chosen at random from the interval $[0,p-1]$, with $a > 0$.

Collision Resolution Strategies

- If two keys map on the same hash table index then we have a collision.
- As the number of elements in the table increases, the likelihood of a *collision* increases - so make the table **as large as practical**
- Collisions may still happen, so we need a *collision resolution strategy*

- Two approaches are used to resolve collisions.

  (a) ***Separate chaining***: chain together several keys/entries in each position.

  (b) ***Open addressing***: store the key/entry in a different position.

- **Probing**: If the table position given by the hashed key is already occupied, increase the position by some amount, until an empty position is found

Open Addressing

- Types of open addressing are

    1. Linear Probing
    2. Quadratic Probing
    3. Double Hashing.

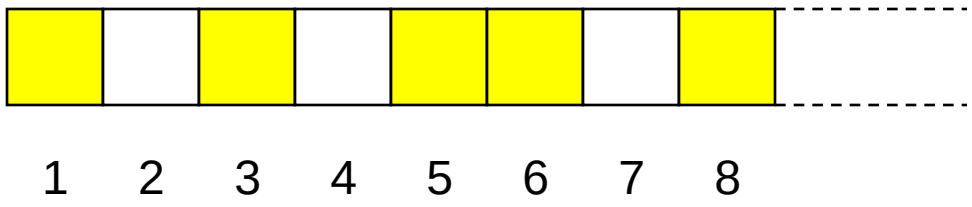
1. Linear Probing

- Locations are checked from the hash location **k** to the end of the table and the element is placed in the first empty slot
- If the bottom of the table is reached, checking “wraps around” to the start of the table. Modulus is used for this purpose
- Thus, if linear probing is used, these routines must continue down the table until a match or empty location is found

- Linear probing is guaranteed to find a slot for the insertion if there still an empty slot in the table.
- Even though the hash table size is a prime number is probably not an appropriate size; the size should be at least 30% larger than the maximum number of elements ever to be stored in the table.
- If the load factor is greater than 50% - 70% then the time to search or to add a record will increase.

H(k)=h, h+1, h+2, h+3,……, h+I

- However, linear probing also tends to promote clustering within the table.



2. Quadratic Probing

- Quadratic probing is a solution to the clustering problem
  - *Linear probing* adds 1, 2, 3, etc. to the original hashed key
  - *Quadratic probing* adds 1^2, 2^2, 3^2 etc. to the original hashed key
- However, whereas linear probing guarantees that all empty positions will be examined if necessary, quadratic probing does not

- If the table size is prime, this will try approximately half the table slots.
- More generally, with quadratic probing, insertion may be impossible if the table is more than half-full!

$$H(k) = h, h+1, h+4, h+5, h+6, \ldots\ldots, h+i^2$$

3. Double Hashing

- 2nd hash function H' is used to resolve the collision.
- Here H'(k) = h' ≠ m
- Therefore we can search the locations with addresses,

  H'(k) = h, h+h', h+2h', h+3h',…….
- If m is prime, then this sequence access all the locations.

Double Hashing

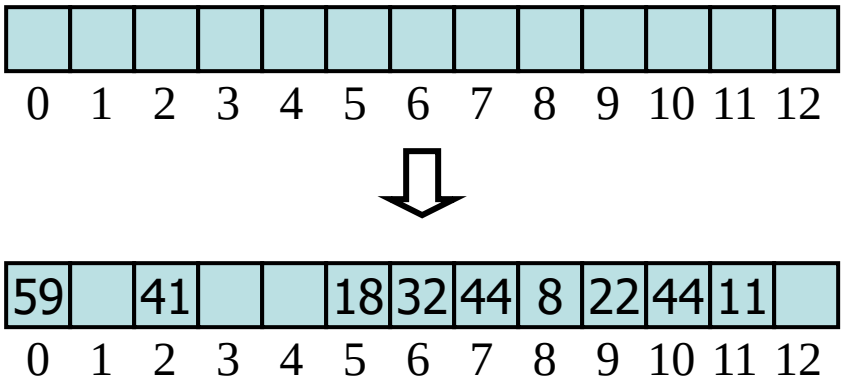
- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(h + jd(k)) \bmod N$$
for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression map for the secondary hash function:
$$d_2(k) = k \bmod q$$
where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
$$1, 2, \ldots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	9	5		
41	2	8	2		
22	9	10	9		
44	5	5	5	7	
59	7	10	7	10	0
32	6	4	6		
31	5	8	5	8	
73	8	11	8	11	

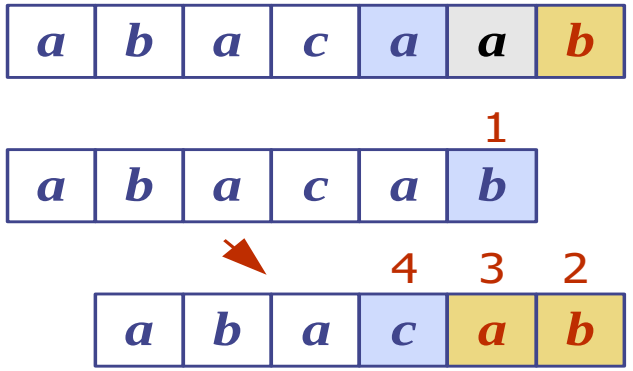


REHASHING

- Rehashing means **hashing again**.
- Basically, when the load factor increases to more than its predefined value, the complexity increases.
- So to overcome this, 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

- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - ordinary text
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}



- Let ***P*** be a string of size ***m***
  - A substring ***P***[***i*** **..** ***j***] of ***P*** is the subsequence of ***P*** consisting of the characters with ranks between ***i*** and ***j***
  - A prefix of ***P*** is a substring of the type ***P***[0 **..** ***i***]
  - A suffix of ***P*** is a substring of the type ***P***[***i*** **..*m*** **-** 1]
- Given strings ***T*** (text) and ***P*** (pattern), the pattern matching problem consists of finding a substring of ***T*** equal to ***P***
- Applications:
  - text editors
  - search engines
  - biological research

NAÏVE (BRUTE-FORCE) STRING

- The Naïve(brute-force) pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences
  - unlikely in English text

Algorithm ***BruteForceMatch***(***T***, ***P***)

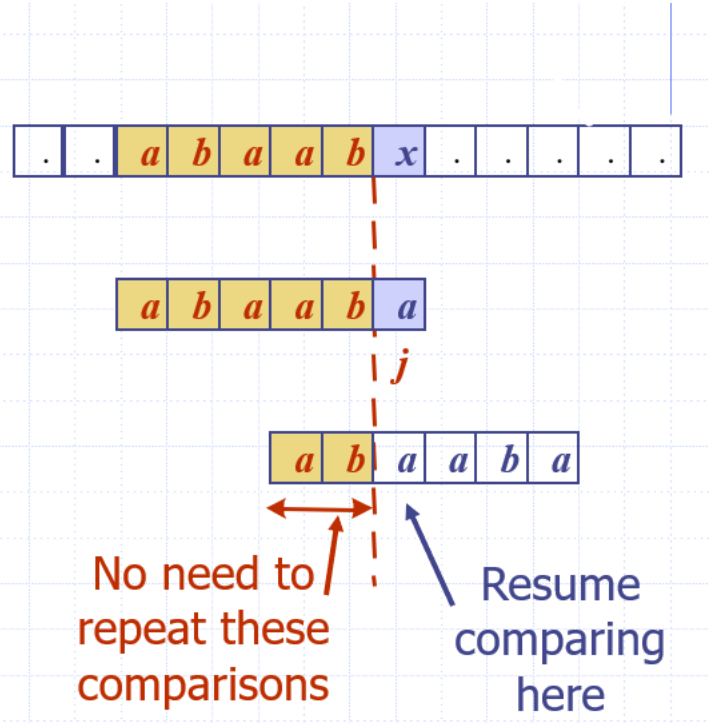
Input text ***T*** of size ***n*** and pattern ***P*** of size ***m***

Output starting index of a substring of ***T*** equal to ***P*** or −1 if no such substring exists

```
for i ← 0 to n − m
    { test shift i of the pattern }
    j ← 0
    while j < m ∧ T[i + j] = P[j]
      j ← j + 1
    if j = m
      return i {match at i}
    else
      break while loop {mismatch}
return -1 {no match anywhere}
```

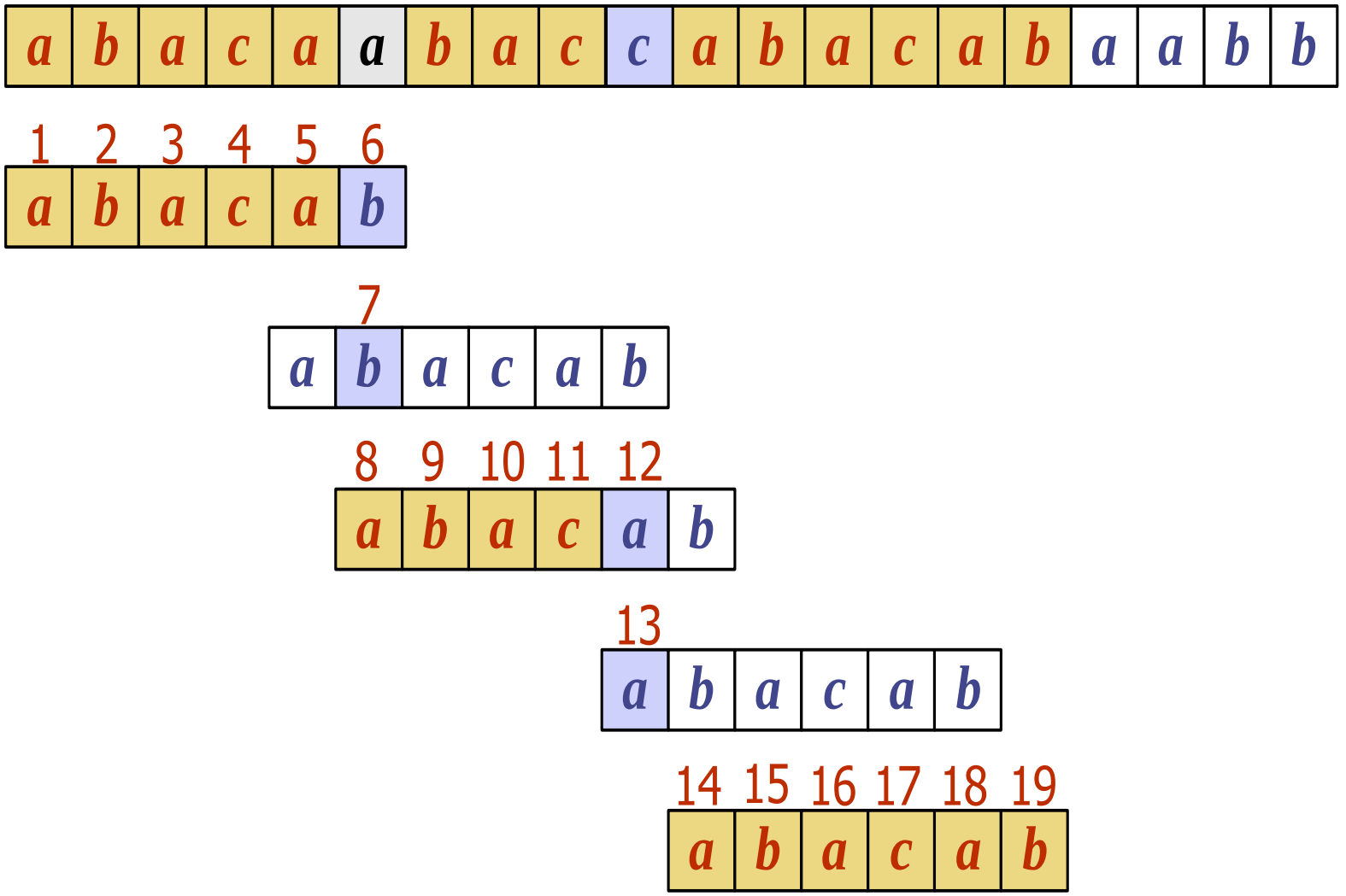
KMP Algorithm

- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$
- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$

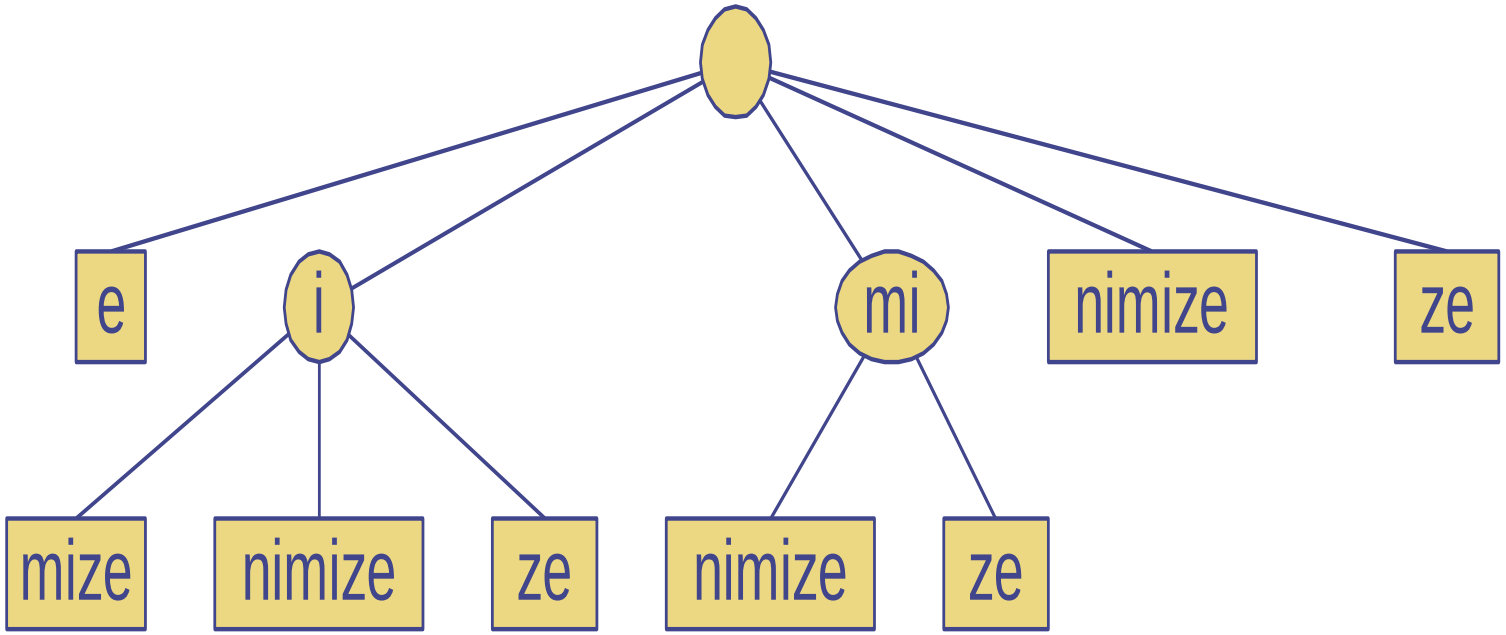


```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```

EXAMPLE


a b a c a a b a c c a b a c a b a a b b
1 2 3 4 5 6
a b a c a b
7
a b a c a b
8 9 10 11 12
a b a c a b
13
a b a c a b
14 15 16 17 18 19
a b a c a b

TRIES



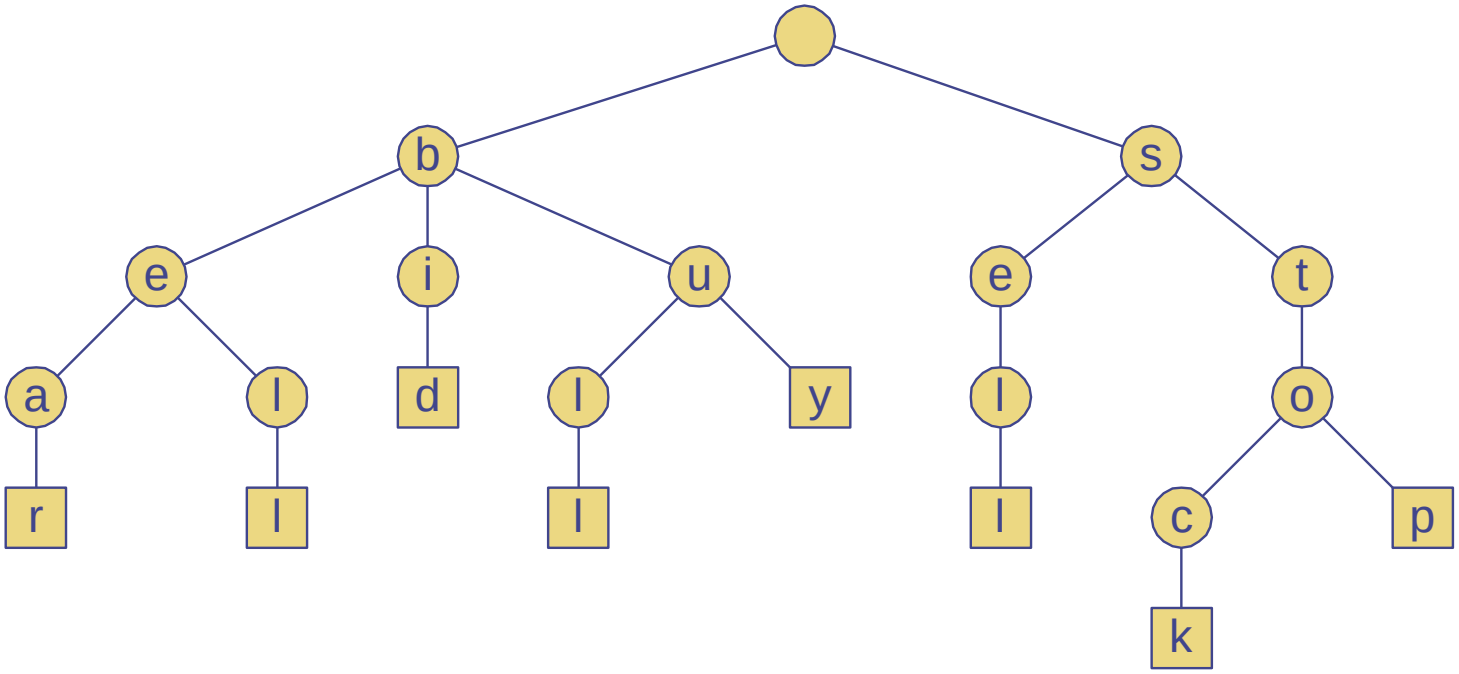
Preprocessing Strings

- Preprocessing the pattern speeds up pattern matching queries
  - After preprocessing the pattern, KMP's algorithm performs pattern matching in time proportional to the text size
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - A tries supports pattern matching queries in time proportional to the pattern size

Standard Tries

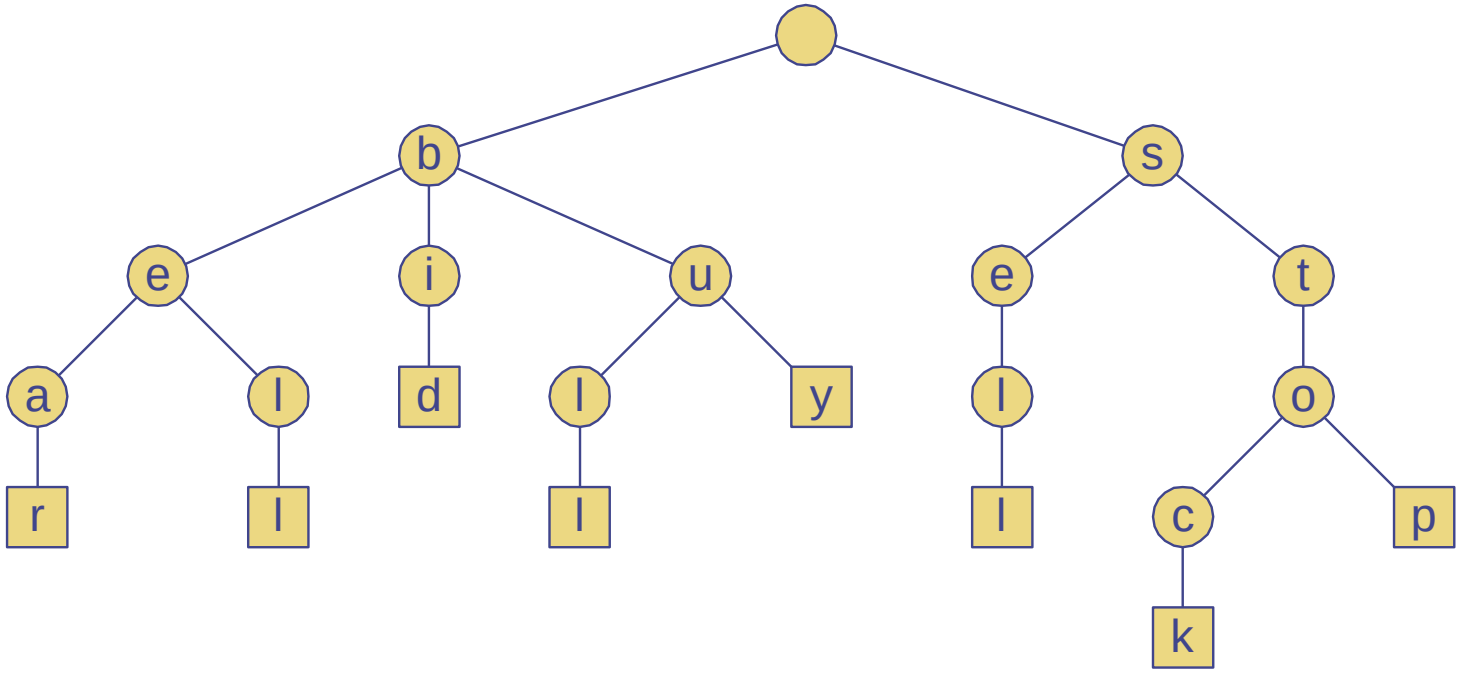
- The standard trie for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings

  S = { bear, bell, bid, bull, buy, sell, stock, stop }



Analysis of Standard Tries

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
  - n total size of the strings in S
  - m size of the string parameter of the operation
  - d size of the alphabet



Word Matching with a Trie

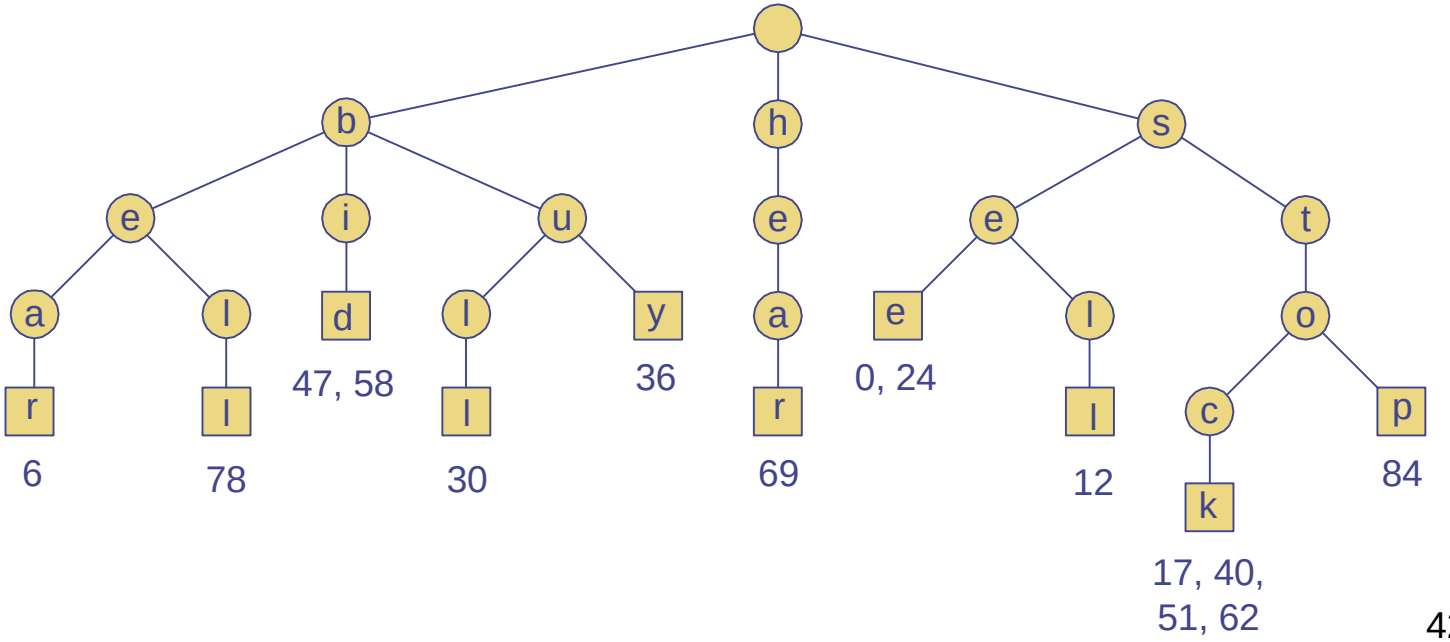
We insert the words of the text into a trie
Each leaf stores the occurrences of the associated word in the text

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!	
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

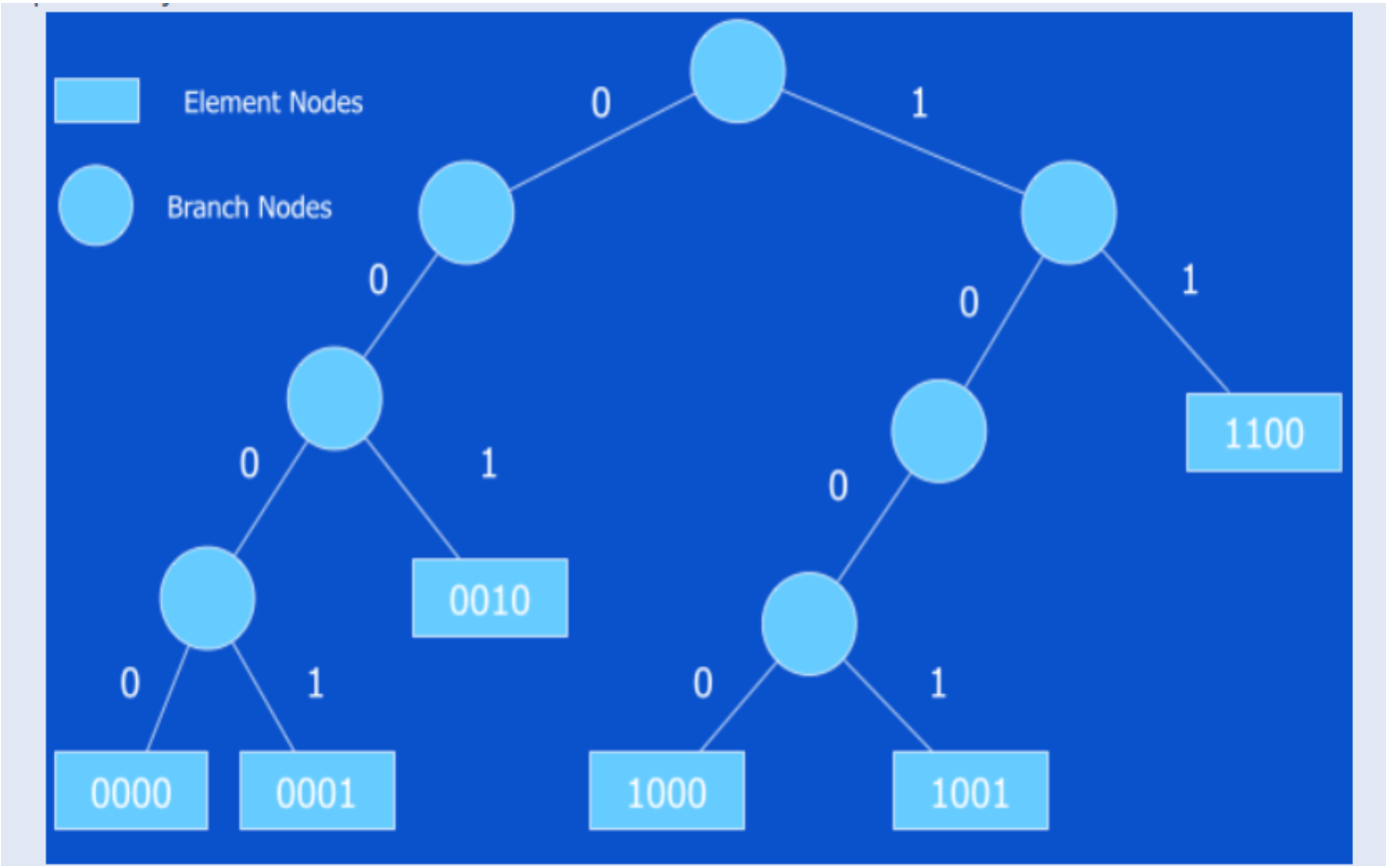
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88



BINARY TRIES

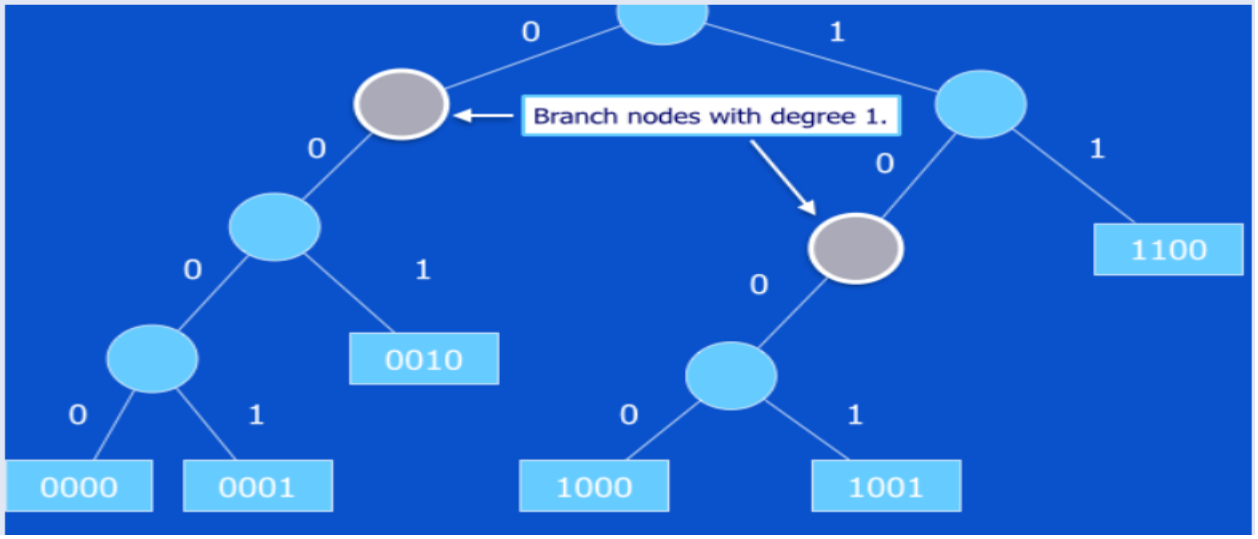
- A binary trie is a binary tree that has two kinds of nodes: **branch nodes** and **element nodes**.
- A branch node has the two data members LeftChild and RightChild.
- It has no data member.
- An element node has the single data member data.
- Branch nodes are used to build a binary tree search structure similar to that of a digital search tree. This leads to element nodes

Simple binary trie



COMPRESSED BINARY TRIES

- The binary trie contains several branch nodes whose degree is one.
- By adding another data member, BitNumber , to each branch node, we can eliminate all degree-one branch nodes from the trie.



- The BitNumber data member of a branch node gives the bit number of the key that is to be used at this node.
- The binary trie after removing the branch nodes with degree 1 is called as **"Compressed binary trie“**
- The binary trie after removing the branchNodes with degree 1 is as follows: