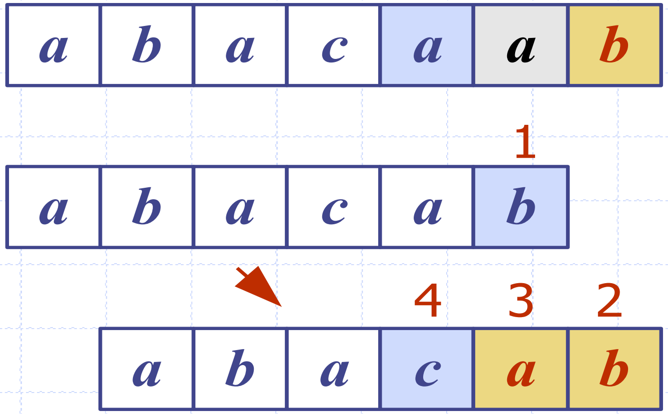
Pattern Matching



Strings

- A string is a sequence of characters
- Examples of strings:
  - Python program
  - HTML document
  - DNA sequence
  - Digitized image
- 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

Brute-Force Pattern Matching

- The 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}
```

Right to Left Matching

- Matching the pattern from right to left
- For a pattern abc:

T: `bbacdcbaabcddcdaddaaabcbcb`

P: `abc`

- Worst case is still O(n m)

Bad Character Rule (BCR) (1)

- On a mismatch between the pattern and the text, we can shift the pattern by more than one place.

```
ddbbacdcbaabcddcdaddaaabcbcb
acabc
```

Bad Character Rule (BCR) (2)

- Preprocessing –
  - A table, for each position in the pattern and a character, last occurrence of the mismatched character in P preceding the mismatch . O(n |Σ|) space. O(1) access time.

a c a b c

1 2 3 4 5

	1	2	3	4	5
a	1	1	3	3	3
b				4	4
c		2	2	2	5

Bad Character Rule (BCR) (3)

- On a mismatch, shift the pattern to the right until the first occurrence of the mismatched character in P.

- Still O(n m) worst case running time:

```
T: aaaaaaaaaaaaaaaaaaaaaaaaa
P: abaaaa
```

Good Suffix Rule (GSR) (1)

- We want to use the knowledge of the matched characters in the pattern’s suffix.
- If we matched S characters in T, what is (if exists) the smallest shift in P that will align a sub-string of P of the same S characters ?

Good Suffix Rule (GSR) (2)

- **Example 1 – how much to move:**

```
T:  bbacdcbaabcddcdaddaaabcbcb
P:  cabbabdbab
          cabbabdbab
```

Good Suffix Rule (GSR) (3)

- **Example 2 – what if there is no alignment:**

↓

```
T:  bbacdcbaabcbbabdbabcaabcbcb
P:  bcbbabdbabc
                 bcbbabdbabc
```

Good Suffix Rule (GSR) (4)

T: a b c d e b a

P: d a c

t

- We mark the matched sub-string in T with *t* and the mismatched char with *x*
  1. In case of a mismatch: shift right until the first occurrence of *t* in P such that the next char *y* in P holds $y \neq x$
  2. Otherwise, shift right to the largest prefix of P that aligns with a suffix of *t*.

Boyer Moore Algorithm

- Preprocess(P)
- k := n

while (k ≤ m) do

- Match P and T from right to left starting at k
- If a mismatch occurs: shift P right (advance k) by max(good suffix rule, bad char rule).
- else, print the occurrence and shift P right (advance k) by the good suffix rule.

Boyer-Moore Algorithm Demo

GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG

Bad Character Rule – shift by 7
Good Suffix Rule – shift by 0

Boyer-Moore Algorithm Demo

GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG *t*

Bad Character Rule – shift by 1

Good Suffix Rule – shift by 3 (9-6)

Pattern P – GTAGCGGCG ***t*** **– GCG**

Prefixes of P

G

GT

GTA

GTAG

GTAGC

GTAGCG

Find the longest prefix of P which has *t* as the suffix.

Boyer-Moore Algorithm Demo

GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG *t*

Pattern P – GTAGCGGCG

t – GCGGCG

Prefixes of P

G
GT
GTA
GTAG
GTAGC
GTAGCG
GTAGCGG
GTAGCGGC
GTAGCGGCG

Find the longest prefix of P which has t as the suffix.

Boyer-Moore Algorithm Demo

GTTATAGCTGATCGCGGCGTAGCGGCGAA

GTAGCGGCG *t*

Bad Character Rule – shift by 3
Good Suffix Rule – shift by 8 (9-1)

Pattern P – GTAGCGGCG *t* – GCGGCG

suffixes of *t*	Prefixes of P
G	G
CG	GT
GCG	GTA
GCGG	GTAG
GCGGC	GTAGC
GCGGCG	GTAGCG

Find the longest prefix of P which is a suffix of *t*.

Boyer-Moore Algorithm Demo

```
GTTATAGCTGATCGCGGCGTAGCGGCGAA
                  GTAGCGGCG
```

Good Suffix Rule (GSR) (5)

- L(i) – The biggest index j, such that j < n and prefix P[1..j] contains suffix P[i..n] as a suffix but not suffix P[i-1..n]

i	1	2	3	4	5	6	7	8	9
P	G	T	A	G	C	G	G	C	G
L(i)	**0**	**0**	**0**	**0**	**0**	**0**	**6**	**0**	**7**

Good Suffix Rule (GSR) (6)

- l(i) – The length of the longest suffix of P[i..n] that is also a prefix of P

i	1	2	3	4	5	6	7	8	9
P	G	T	A	G	C	G	G	C	G
l(i)	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Good Suffix Rule (GSR) (7)

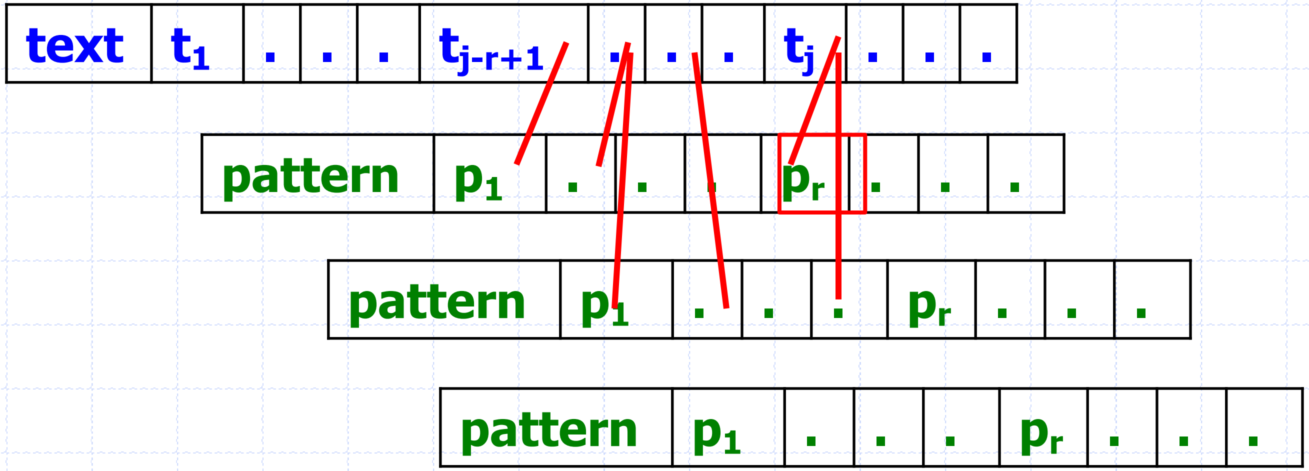
- Putting it together
  - If mismatch occurs at position n, shift P by 1
  - If a mismatch occurs at position i-1 in P:
    - If L(i) > 0, shift P by n – L(i)
    - else shift P by n – l(i)
  - If P was found, shift P by n – l(2)

Boyer-Moore Algorithm Analysis

- Worst case
  - O(n+m) if the pattern does not occur in the text
  - O(nm) if the pattern does occur in the text
  - For patterns of small size, the algorithm might not be efficient.

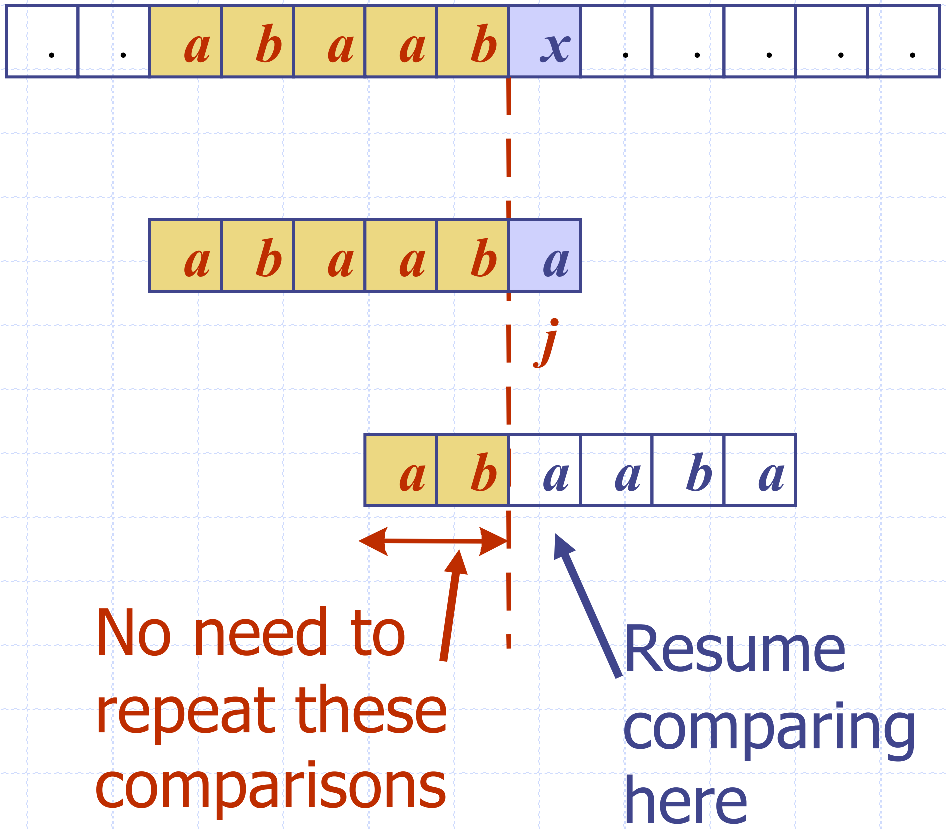
Knuth-Morris-Pratt - Algorithm

- KMP Algorithm
  - compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.



The KMP Algorithm

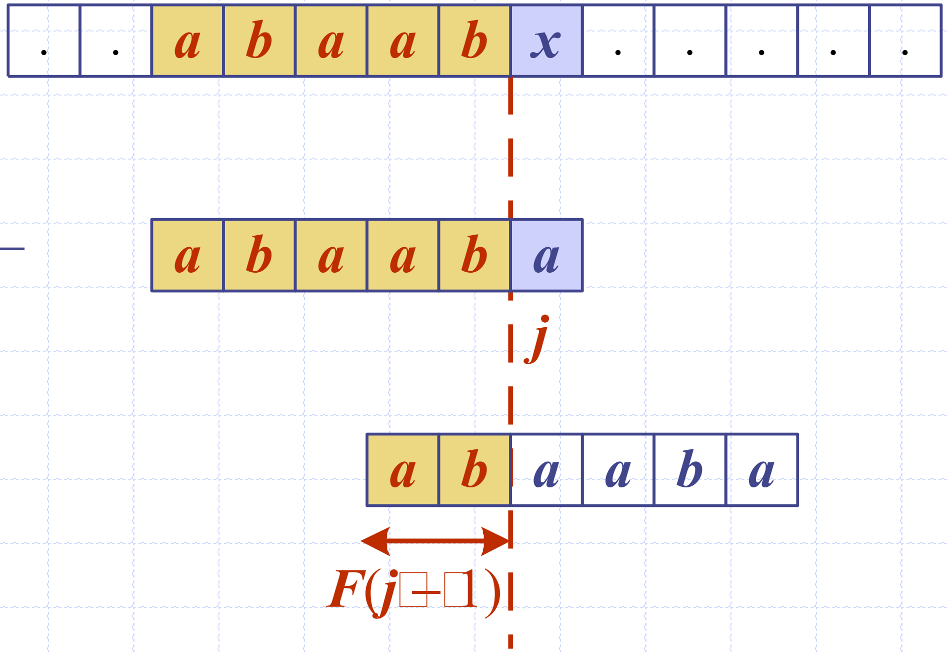
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$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



KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The failure function $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$

j	0	1	2	3	4	□
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	□



Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

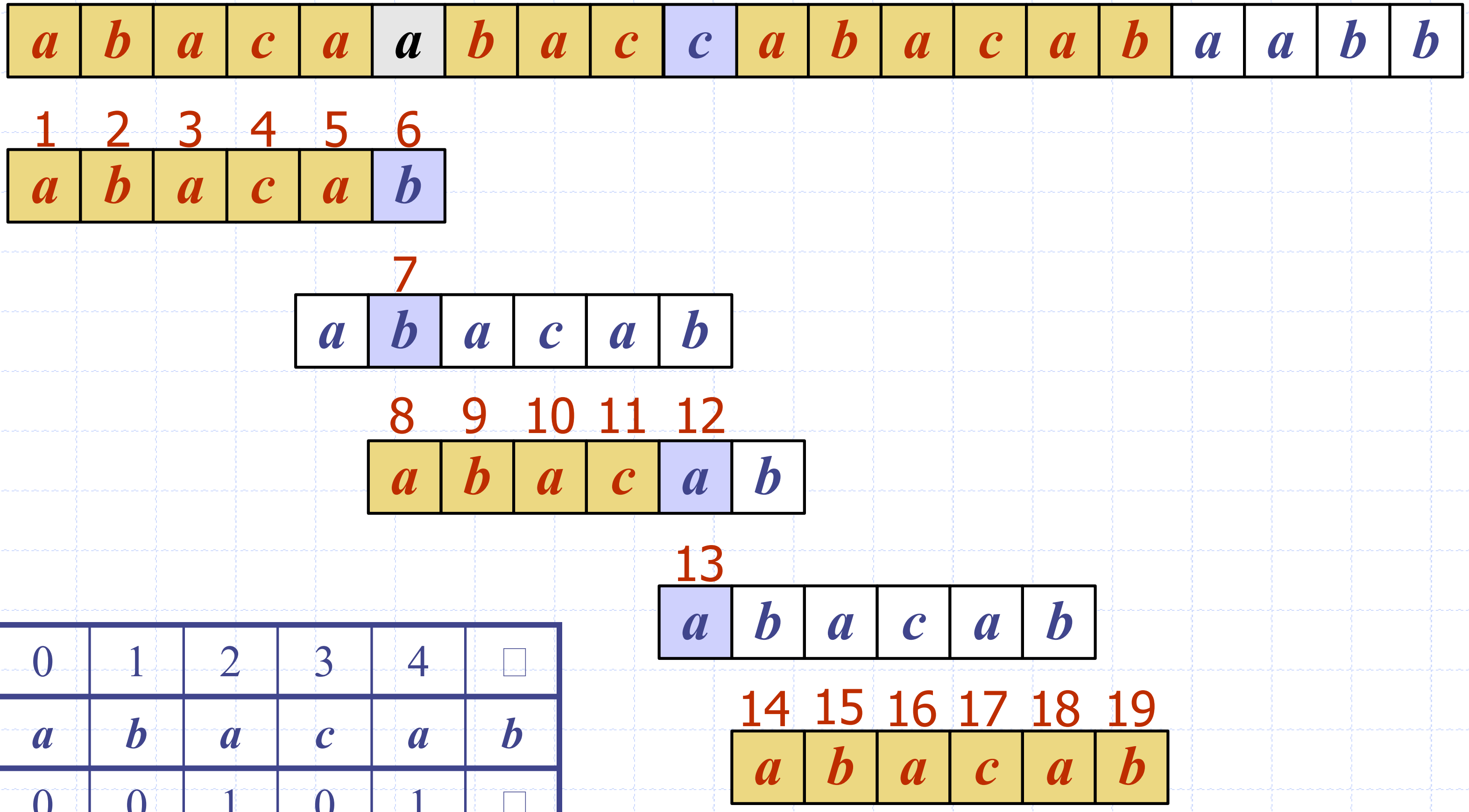
```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i □ m
        if P[i] □ P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i □ 1
            j ← j □ 1
        else if j □ 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i □ 1
```

The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

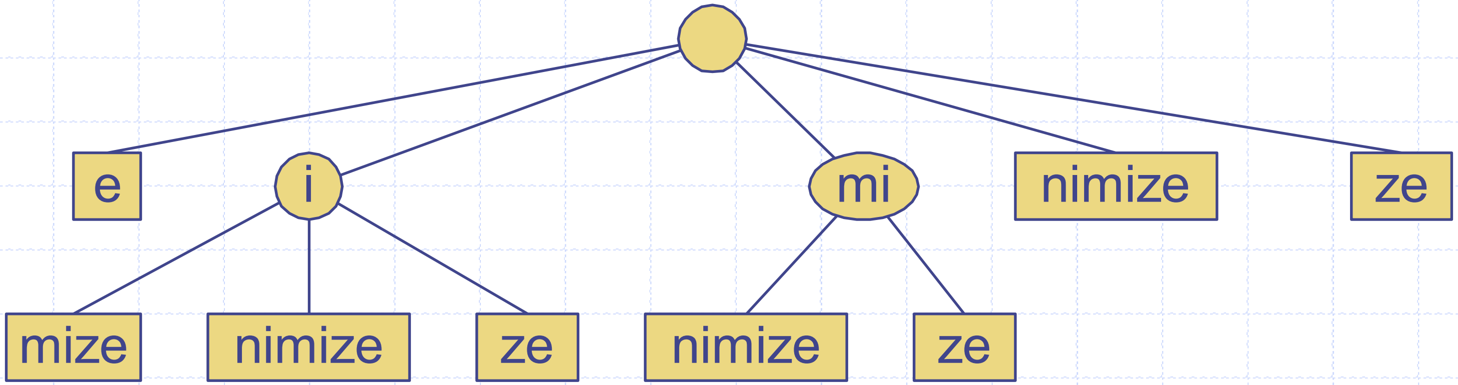
```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ←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

Example



j	0	1	2	3	4	□
$P[j]$	a	b	a	c	a	b
$F(j)$	0	0	1	0	1	□

Tries

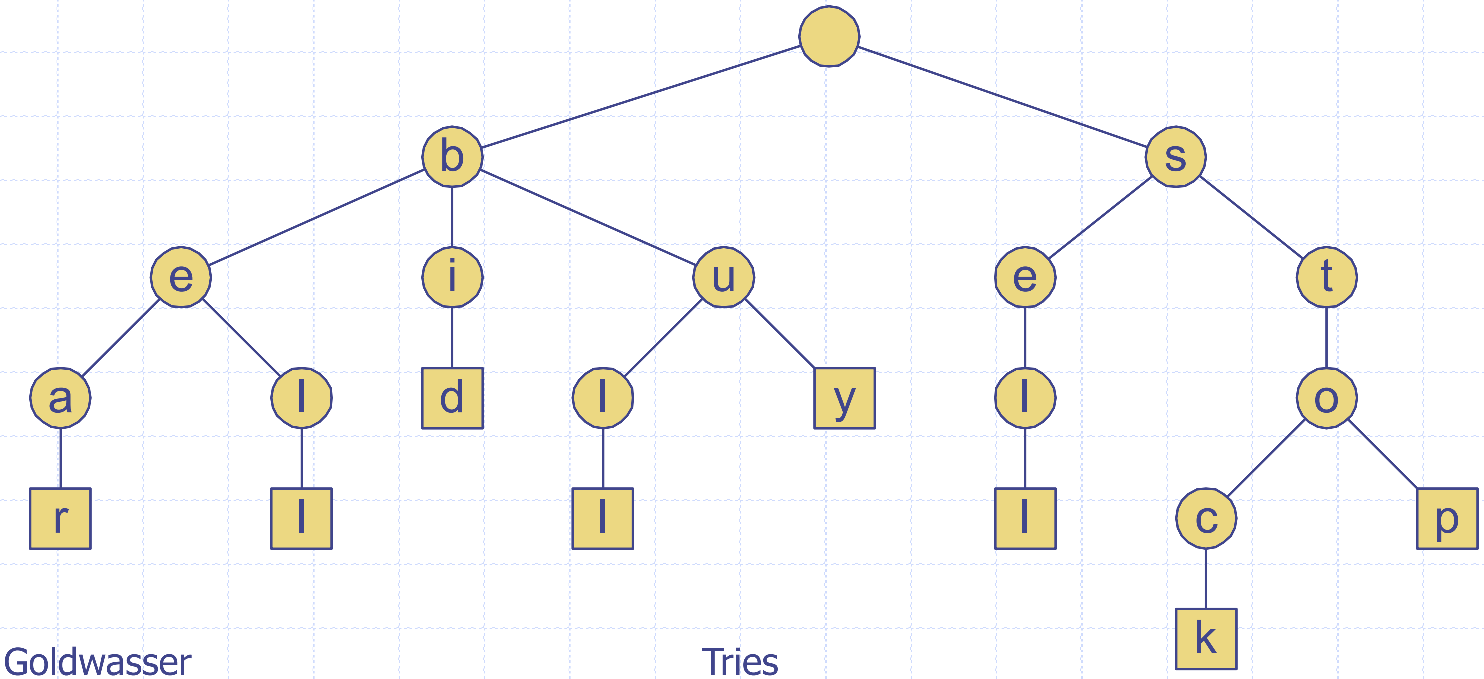


Preprocessing Strings

- 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 trie supports pattern matching queries in time proportional to the pattern size

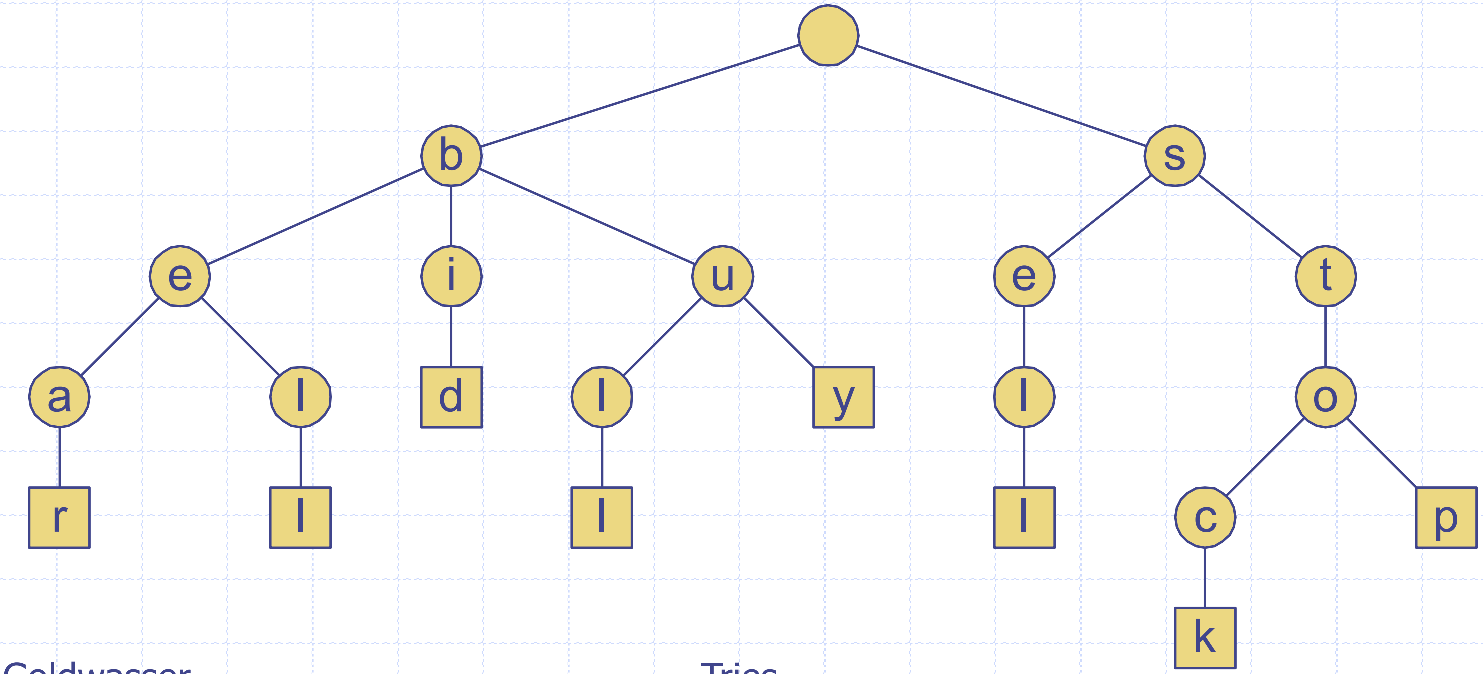
Standard Tries

- 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



Application of Tries

- A standard trie supports the following operations on a processed text in O(m) time, where m is the length of the string.
  - word matching: find the first occurrence of word X in the text.
  - prefix matching: find the first occurrence of the longest prefix of word X in the text.

Word Matching with a Trie

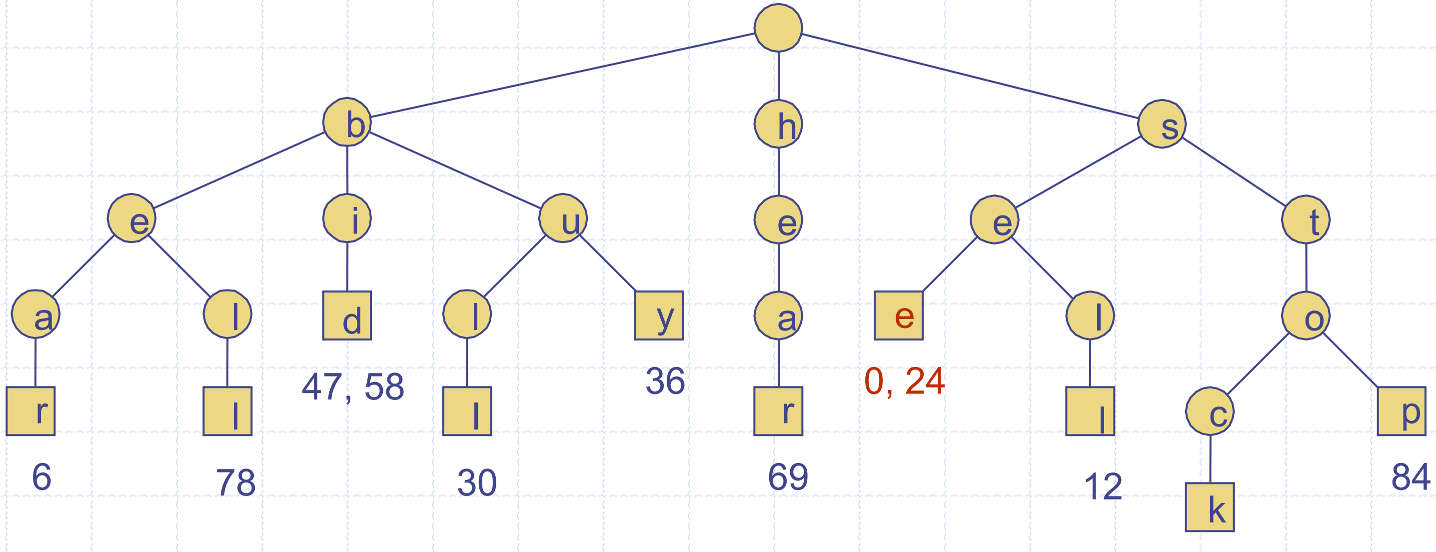
- insert the words of the text into trie
- Each leaf is associated w/ one particular word
- leaf stores indices where associated word begins ("see" starts at index 0 & 24, leaf for "see" stores those indices)

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

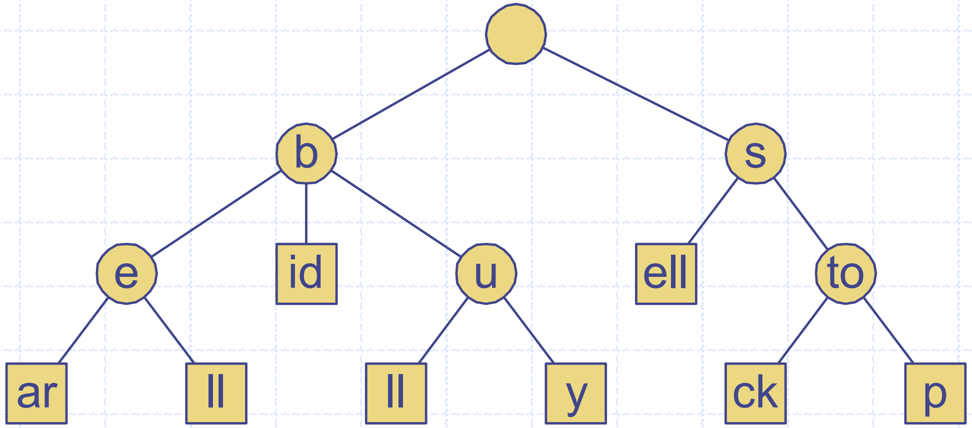
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

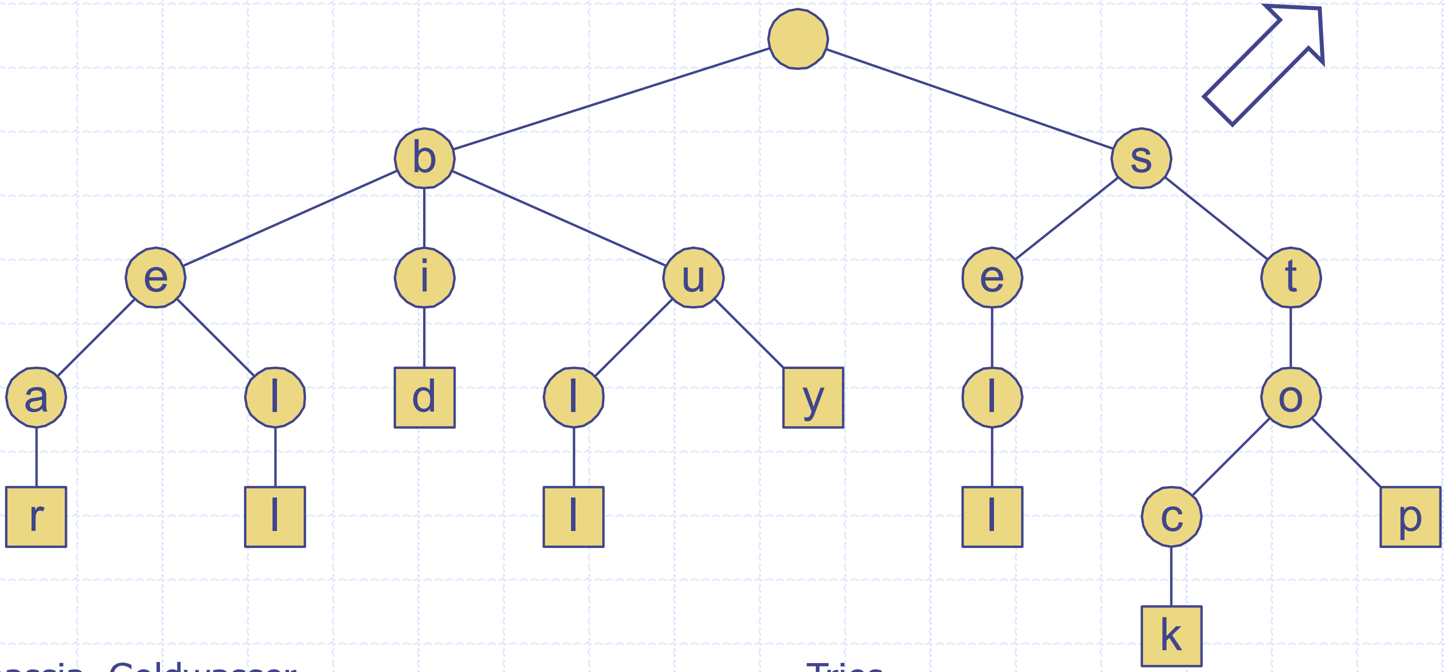
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88



Compressed Tries

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes
- ex. the “i” and “d” in “bid” are “redundant” because they signify the same word



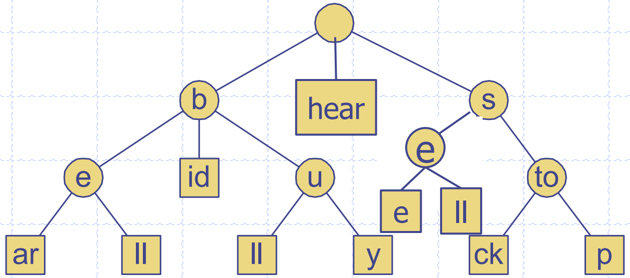


Why Compressed Tries?

- A compressed trie storing a collection S of s strings from an alphabet of size d has the following properties
  - Every internal node of T has at least two children and at most d children
  - T has s leaf nodes
  - The number of internal nodes of T is O(s)

Compact Representation

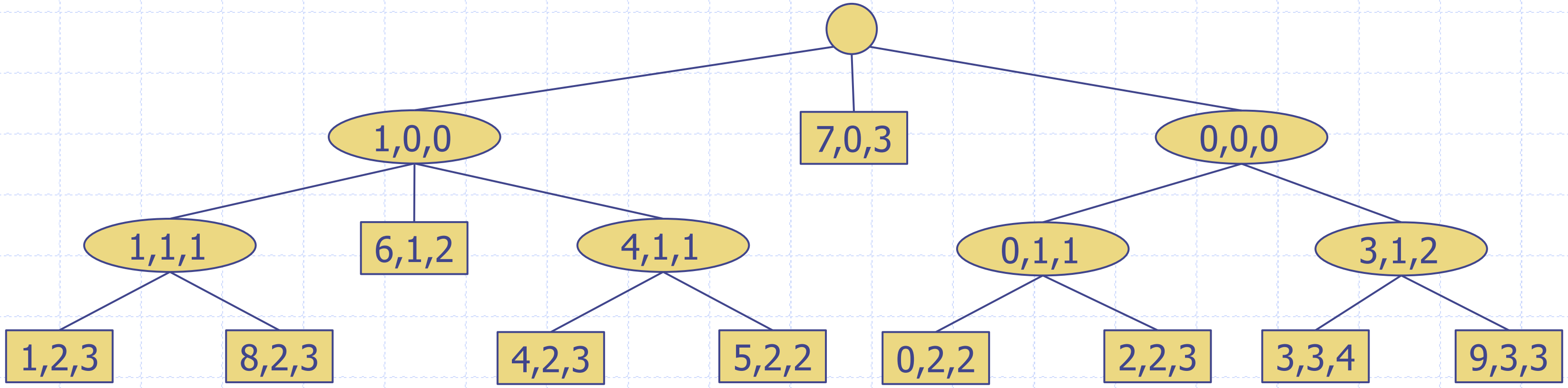
- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses $O(s)$ space, where s is the number of strings in the array
  - Serves as an auxiliary index structure



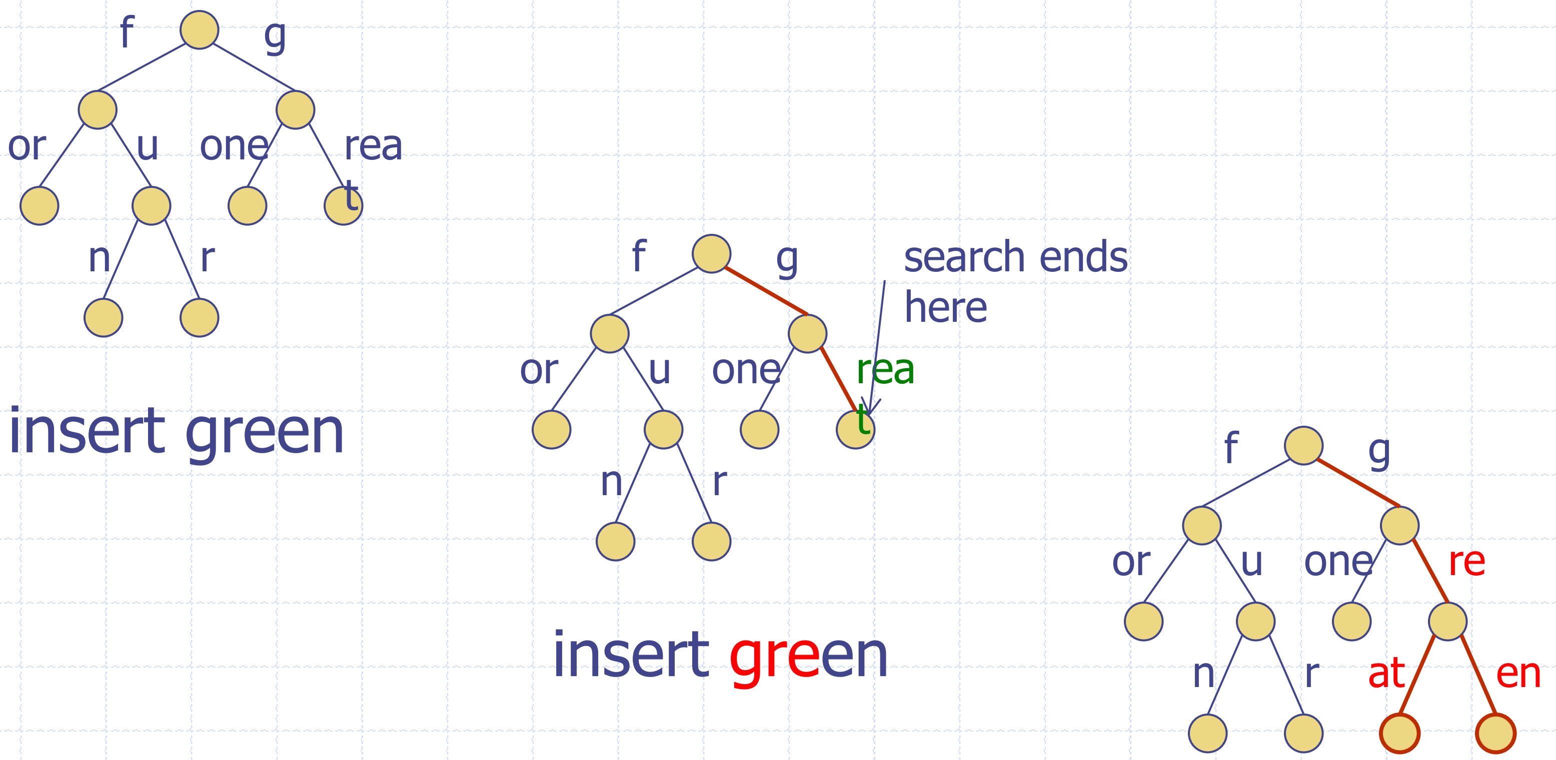
	0	1	2	3	4
S[0] =	s	e	e		
S[1] =	b	e	a	r	
S[2] =	s	e	l	l	
S[3] =	s	t	o	c	k

	0	1	2	3
S[4] =	b	u	l	l
S[5] =	b	u	y	
S[6] =	b	i	d	

	0	1	2	3
S[7] =	h	e	a	r
S[8] =	b	e	l	l
S[9] =	s	t	o	p

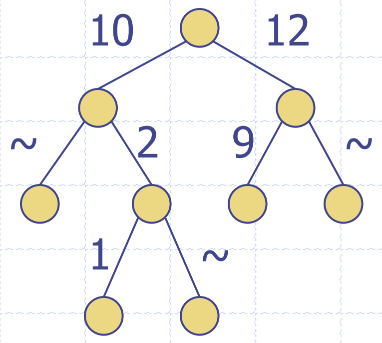


Insertion/Deletion in Compressed Trie



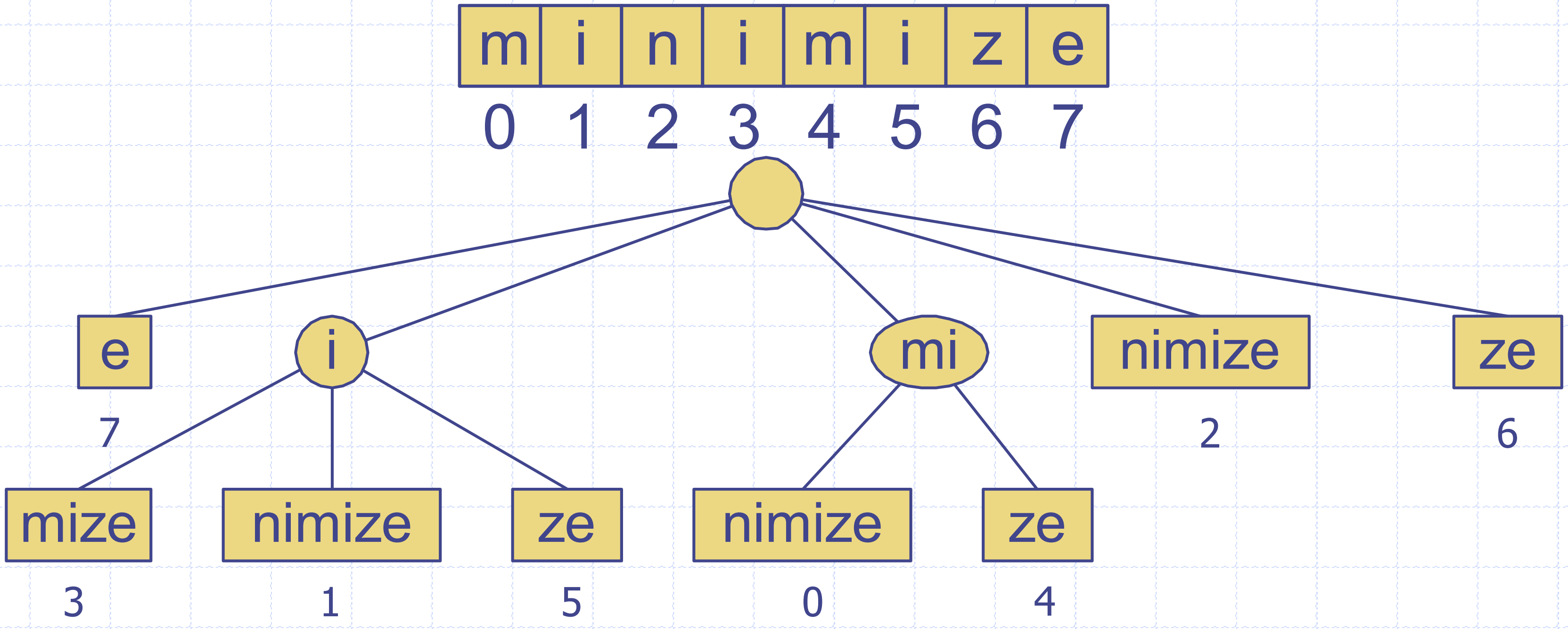
Application - Routing through Tries

- Internet Routers maintain a Trie (table)
- It is not a lookup
  - forwards packets to its neighbors using IP prefix matching rules



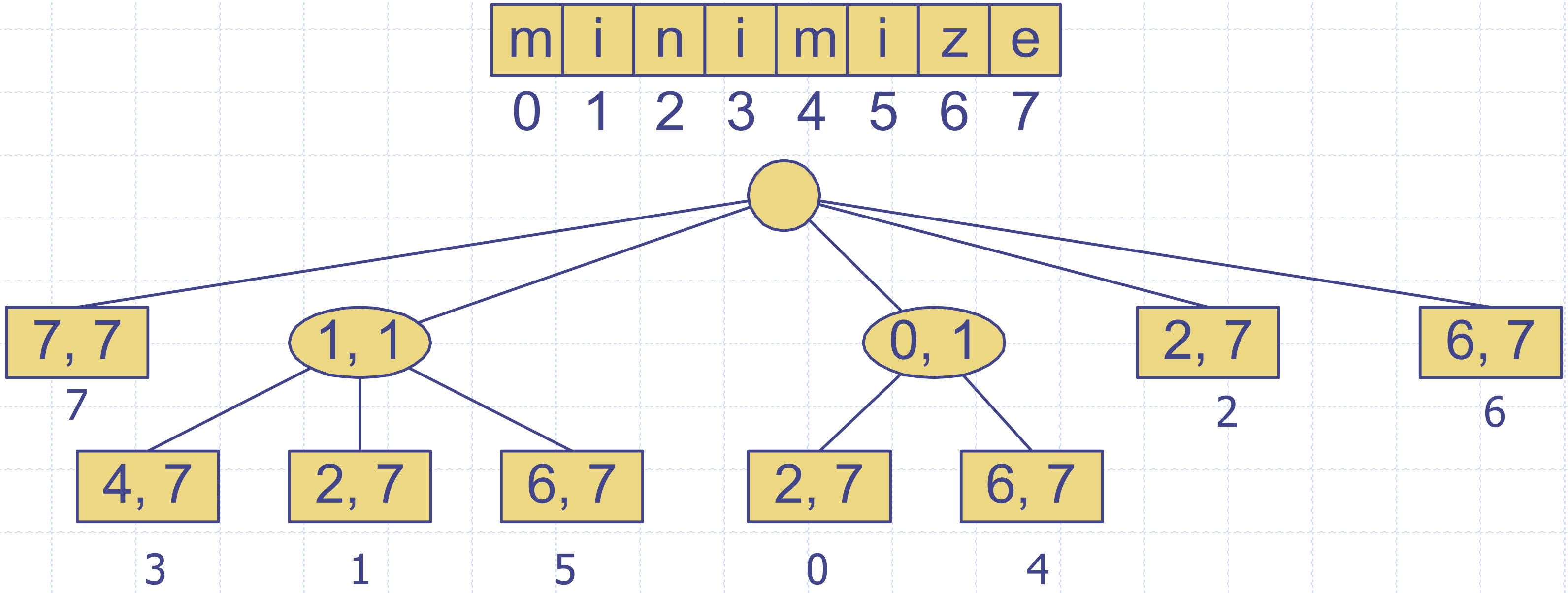
Suffix Trie

- The suffix trie of a string ***X*** is the compressed trie of all the suffixes of ***X***
- Each leaf corresponds to a suffix of ***X***



Analysis of Suffix Tries

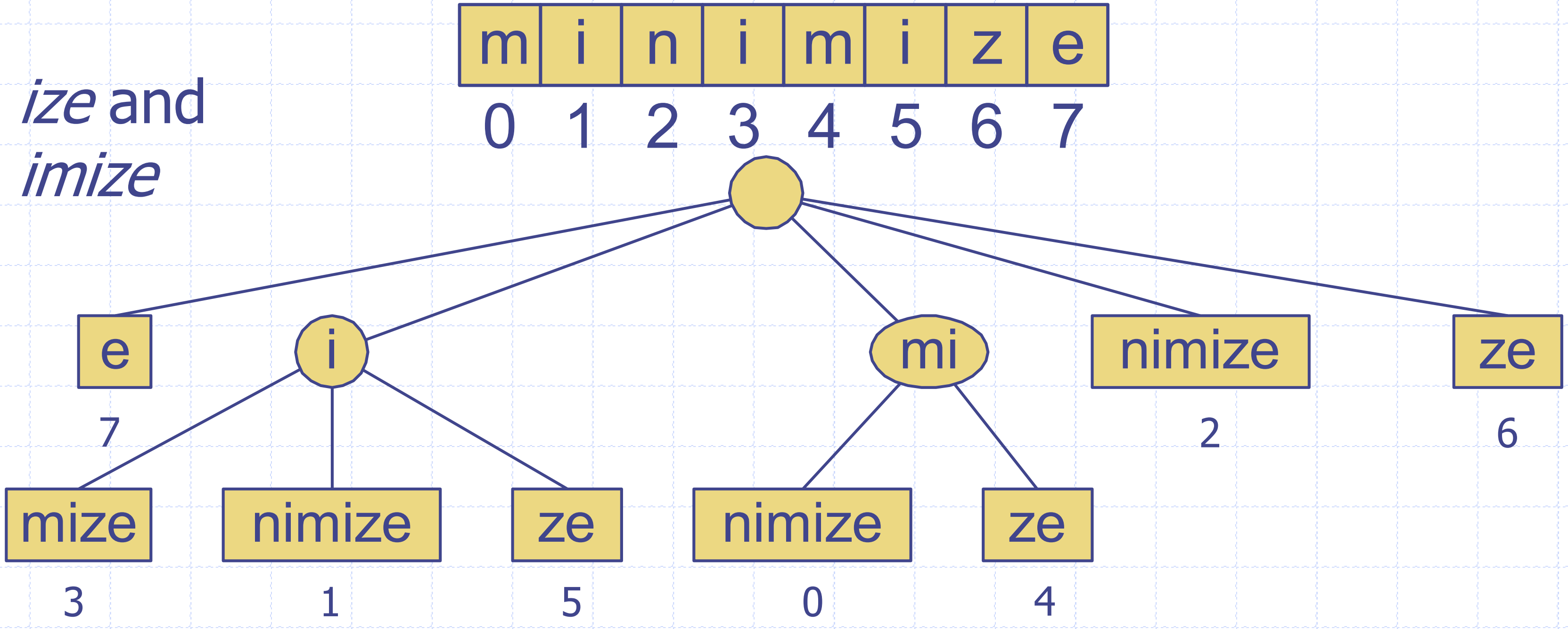
- Compact representation of the suffix trie for a string X of size n from an alphabet of size d
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in X in $O(dm)$ time, where m is the size of the pattern
  - Can be constructed in $O(n)$ time



Prefix Matching using Suffix Trie

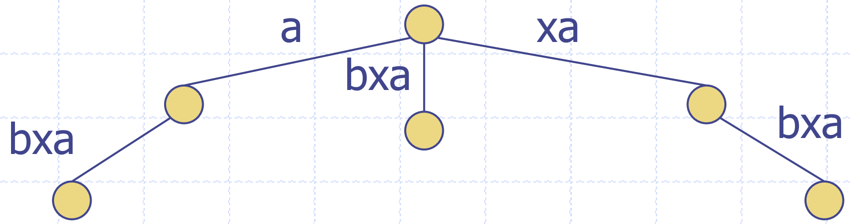
- If two suffixes have a same prefix, then their corresponding paths are the same at their beginning, and the concatenation of the edge labels of the mutual part is the prefix.

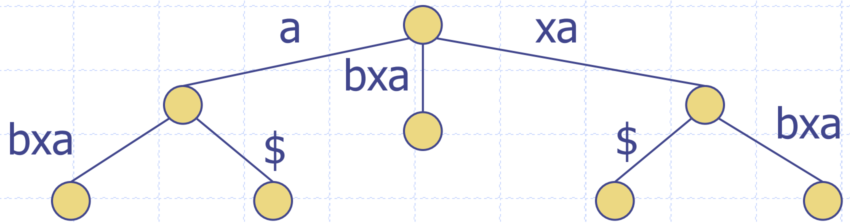
ize and
imize



Suffix Trie

- Not all strings are guaranteed to have corresponding suffix trie.
- For example: xabxa

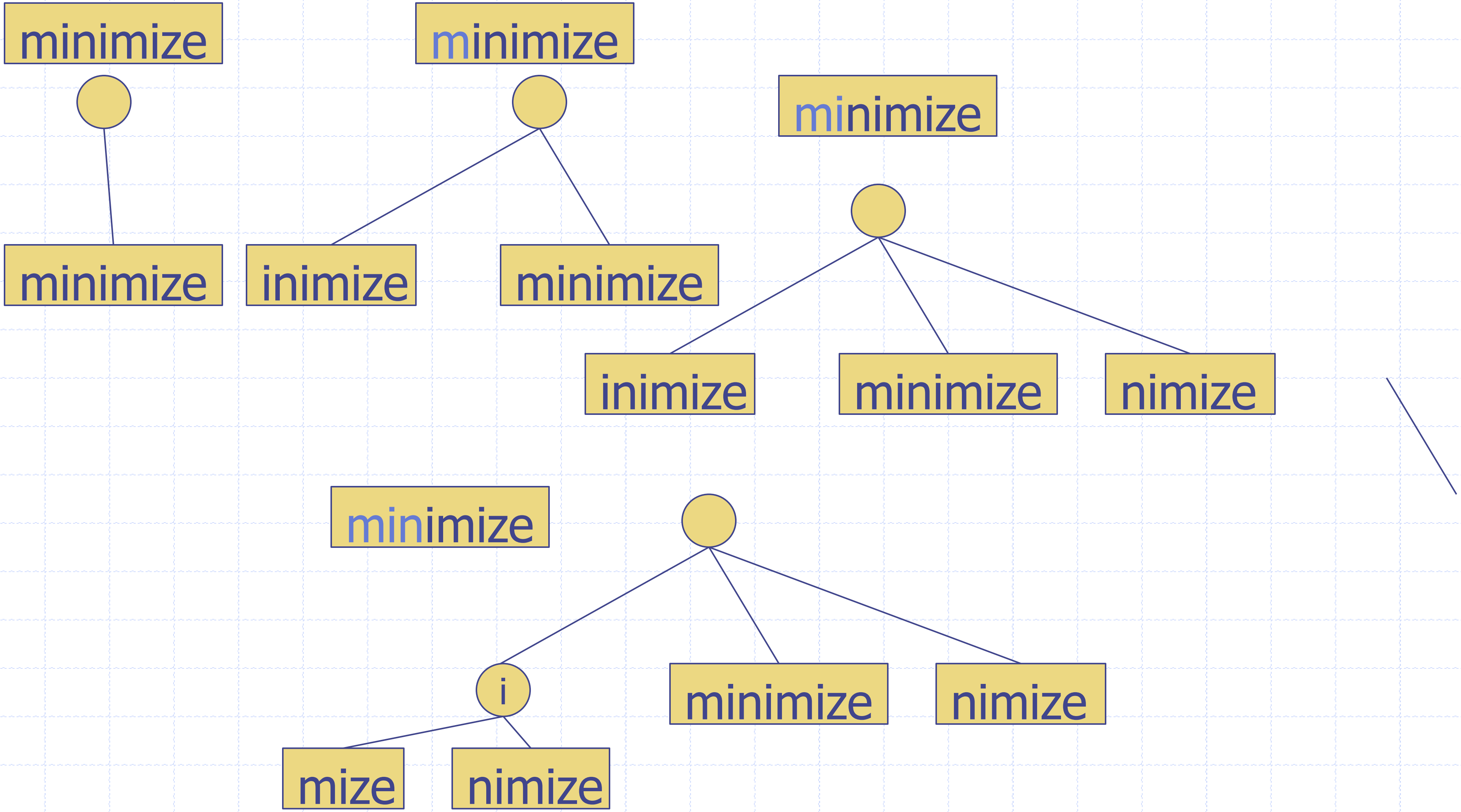




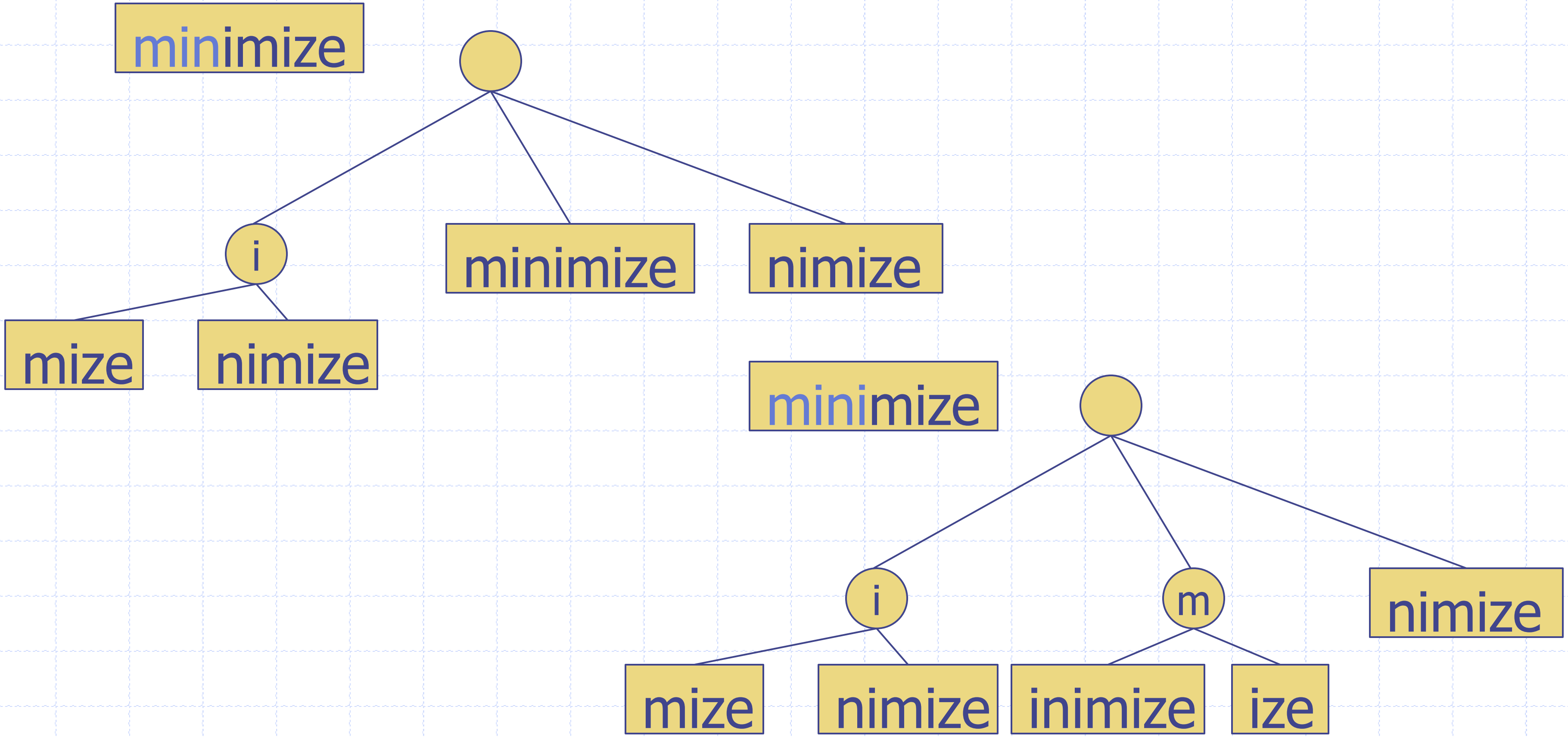
Constructing a Suffix Trie

- S[1..n] is the string
- start with a single edge for S
- enter the edges for the suffix S[i..n] where i goes from 2 to n
  - Starting at the root node find the longest part from the root whose label matches a prefix of S[i..n]. At some point, no further matches are possible
    - If the point is at a node, then denote this node by w
    - If it is in the middle of an edge, then insert a new node called w, at this point
    - create a new edge running from w to a new leaf labeled S[i..n]

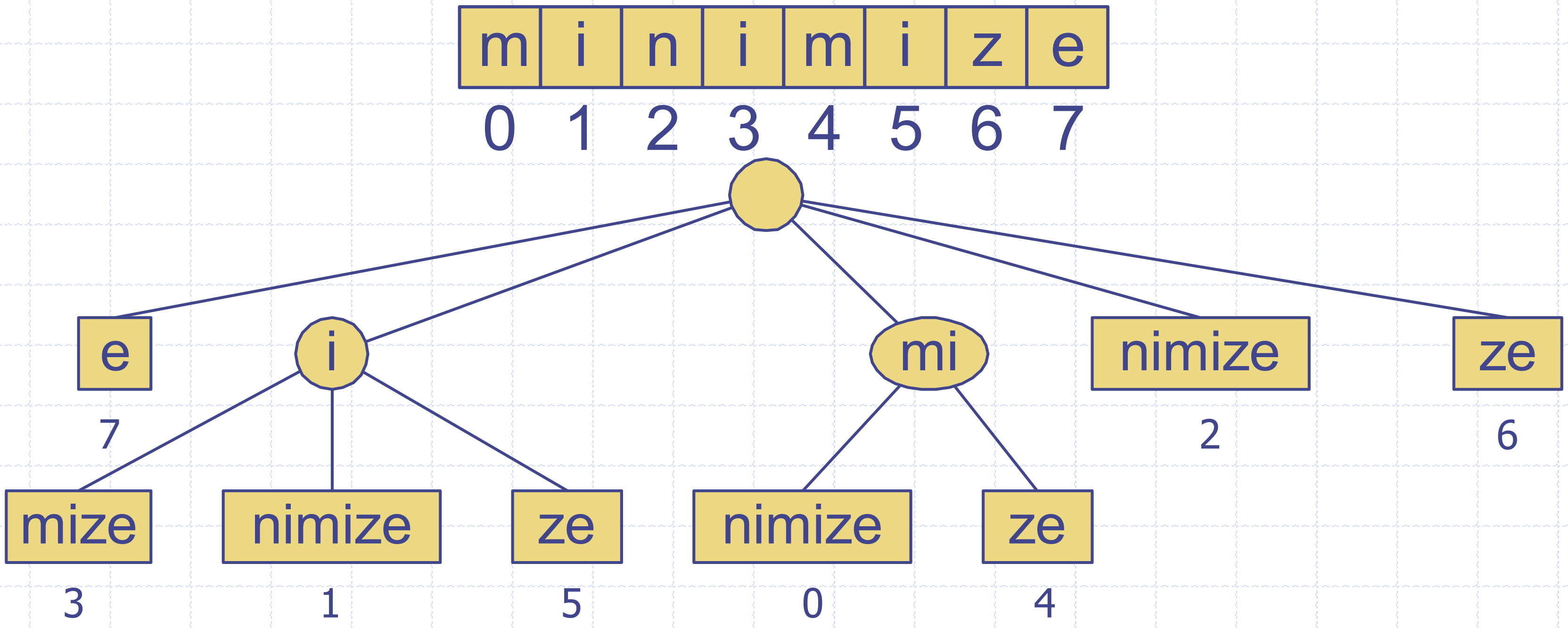
Example


minimize
minimize
minimize
inimize
minimize
minimize
inimize
minimize
nimize
minimize
i
mize
nimize
minimize
nimize

Example (2)


minimize
i
minimize
nimize
mize
nimize
minimize
i
m
nimize
mize
nimize
inimize
ize

Example (3)



m i n i m i z e
0 1 2 3 4 5 6 7
e
7
i
mi
nimize
2
ze
6
mize
3
nimize
1
ze
5
nimize
0
ze
4

Constructing a Suffix Trie

Complexity- $O(n^2)$

- S[1..n] is the string
- start with a single edge for S
- enter the edges for the suffix S[i..n] where i goes from 2 to n
  - Starting at the root node find the longest part from the root whose label matches a prefix of S[i..n]. At some point, no further matches are possible
    - If the point is at a node, then denote this node by w
    - If it is in the middle of an edge, then insert a new node called w, at this point
    - create a new edge running from w to a new leaf labeled S[i..n]

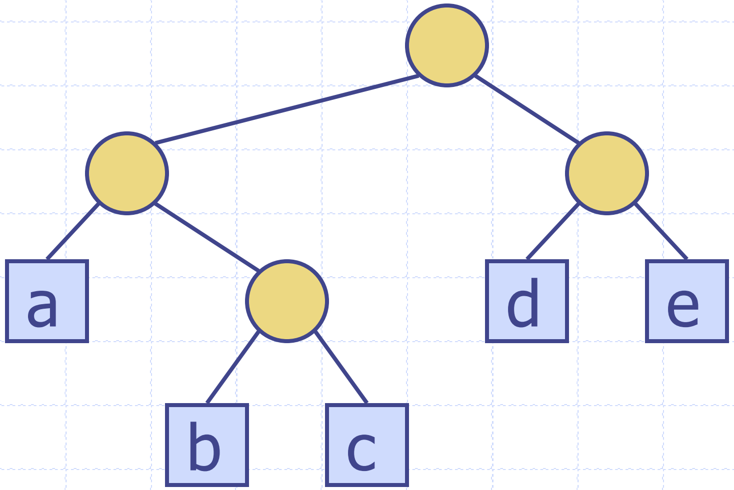
Text Compression

- Given a string X, efficiently encode X into a smaller string Y
  - Saves memory and/or bandwidth
- A good approach: **Huffman encoding**
  - Compute frequency f(c) for each character c.
  - Encode high-frequency characters with short code words
  - No code word is a prefix for another code
  - Use an optimal encoding tree to determine the code words

Encoding Tree Example

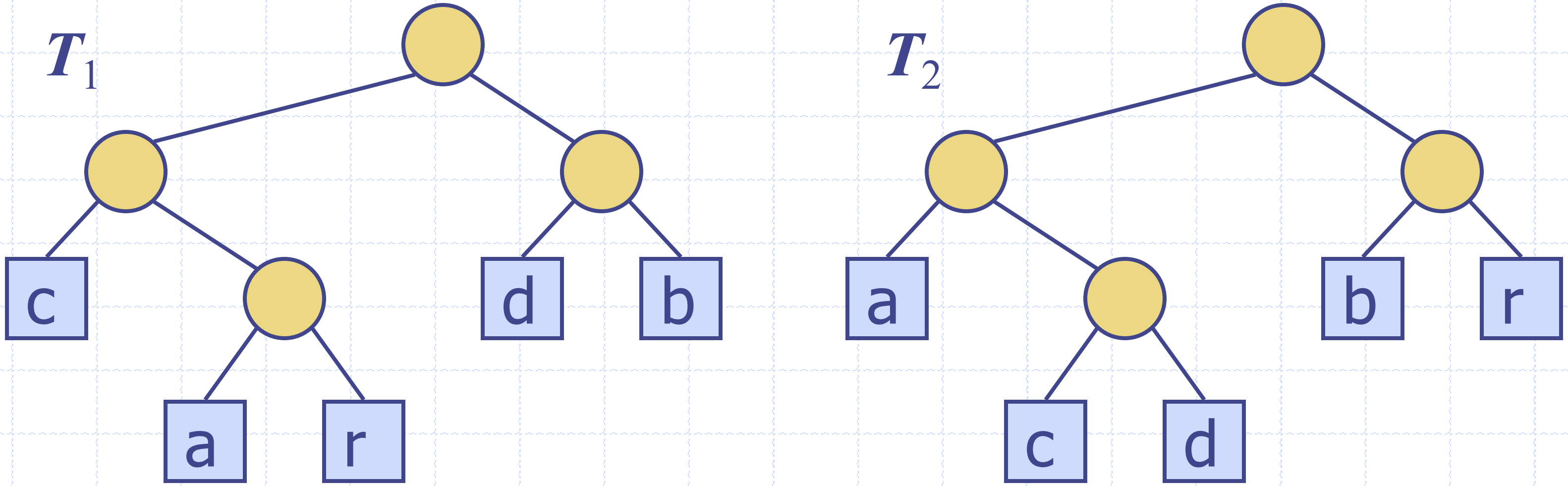
- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node stores a character
  - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



Encoding Tree Optimization

- Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X
  - Frequent characters should have long code-words
  - Rare characters should have short code-words
- Example
  - $X =$ abracadabra
  - T_1 encodes X into 29 bits
  - T_2 encodes X into 24 bits



Huffman's Algorithm

- Given a string X, Huffman's algorithm construct a prefix code the minimizes the size of the encoding of X
- It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

Huffman's Algorithm

Algorithm Huffman(X):
  Input: String X of length n with d distinct characters
  Output: Coding tree for X

Compute the frequency $f(c)$ of each character c of X.
Initialize a priority queue Q.
for each character c in X **do**
  Create a single-node binary tree T storing c.
  Insert T into Q with key $f(c)$.
while len(Q) > 1 **do**
  $(f_1, T_1) = Q$.remove_min()
  $(f_2, T_2) = Q$.remove_min()
  Create a new binary tree T with left subtree T_1 and right subtree T_2.
  Insert T into Q with key $f_1 + f_2$.
$(f, T) = Q$.remove_min()
return tree T

Example

X = abracadabra

Frequencies

a	b	c	d	r
5	2	1	1	2