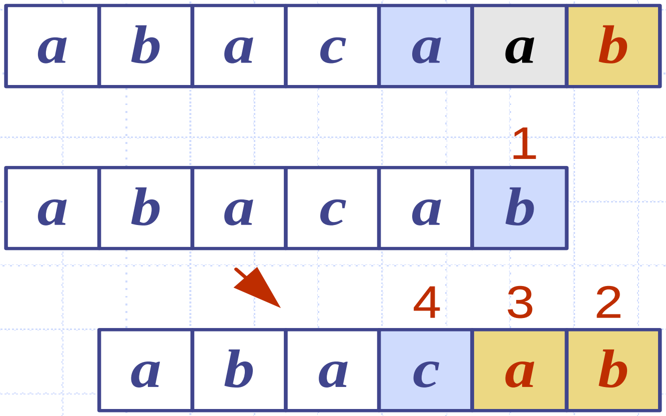
Pattern Matching



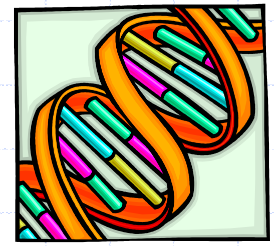
Outline and Reading

- Strings (§11.1)
- Pattern matching algorithms
  - Brute-force algorithm (§11.2.1)
  - Boyer-Moore algorithm (§11.2.2)
  - Knuth-Morris-Pratt algorithm (§11.2.3)

Strings

- A string is a sequence of characters
- Examples of strings:
  - C++ 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 (used by C and C++)
  - Unicode (used by Java)
  - {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 Algorithm



- 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 $i \leftarrow 0$ **to** $n - m$
        { test shift i of the pattern }
        $j \leftarrow 0$
        while $j < m \wedge T[i + j] = P[j]$
            $j \leftarrow j + 1$
        if $j = m$
            return i {match at i}
        {else mismatch at i}
    return -1 {no match anywhere}

Boyer-Moore Heuristics

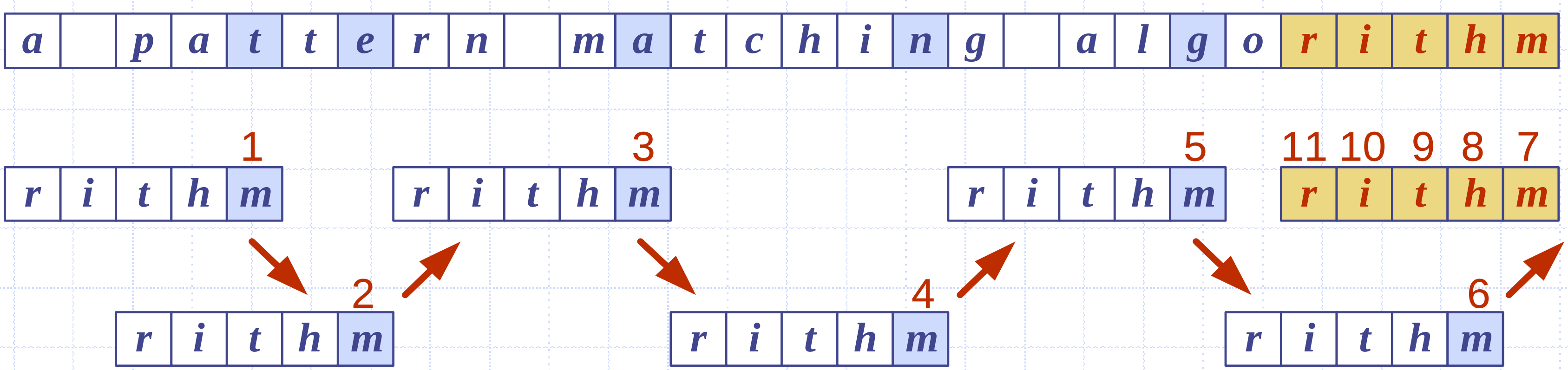
- The Boyer-Moore's pattern matching algorithm is based on two heuristics

  Looking-glass heuristic: Compare P with a subsequence of T moving backwards

  Character-jump heuristic: When a mismatch occurs at $T[i] = c$

  - If P contains c, shift P to align the last occurrence of c in P with $T[i]$
  - Else, shift P to align $P[0]$ with $T[i+1]$

- Example



Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function L mapping Σ to integers, where $L(c)$ is defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- Example:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

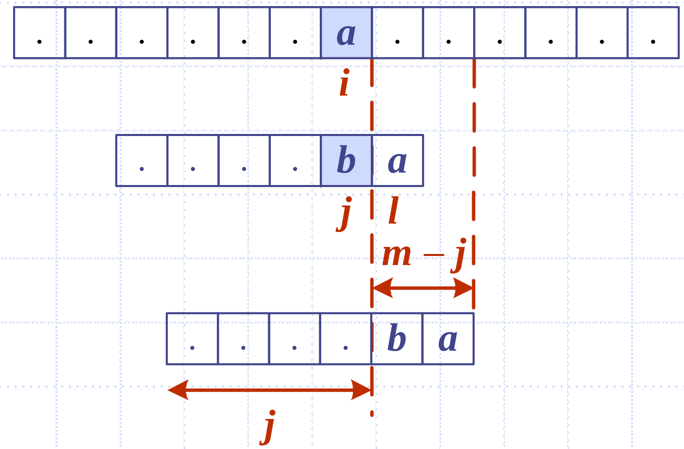
c	a	b	c	d
$L(c)$	4	5	3	-1

- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

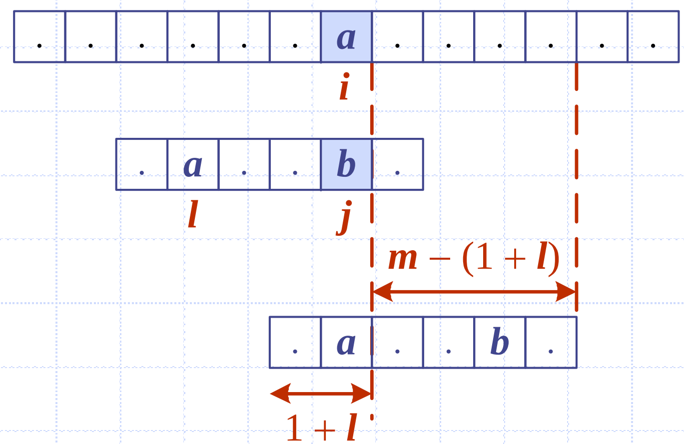
The Boyer-Moore Algorithm

```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ )
    i ← m − 1
    j ← m − 1
    repeat
        if T[i] = P[j]
            if j = 0
                return i  { match at i }
            else
                i ← i − 1
                j ← j − 1
        else
            { character-jump }
            l ← L[T[i]]
            i ← i + m – min(j, 1 + l)
            j ← m − 1
    until i > n − 1
    return −1 { no match }
```

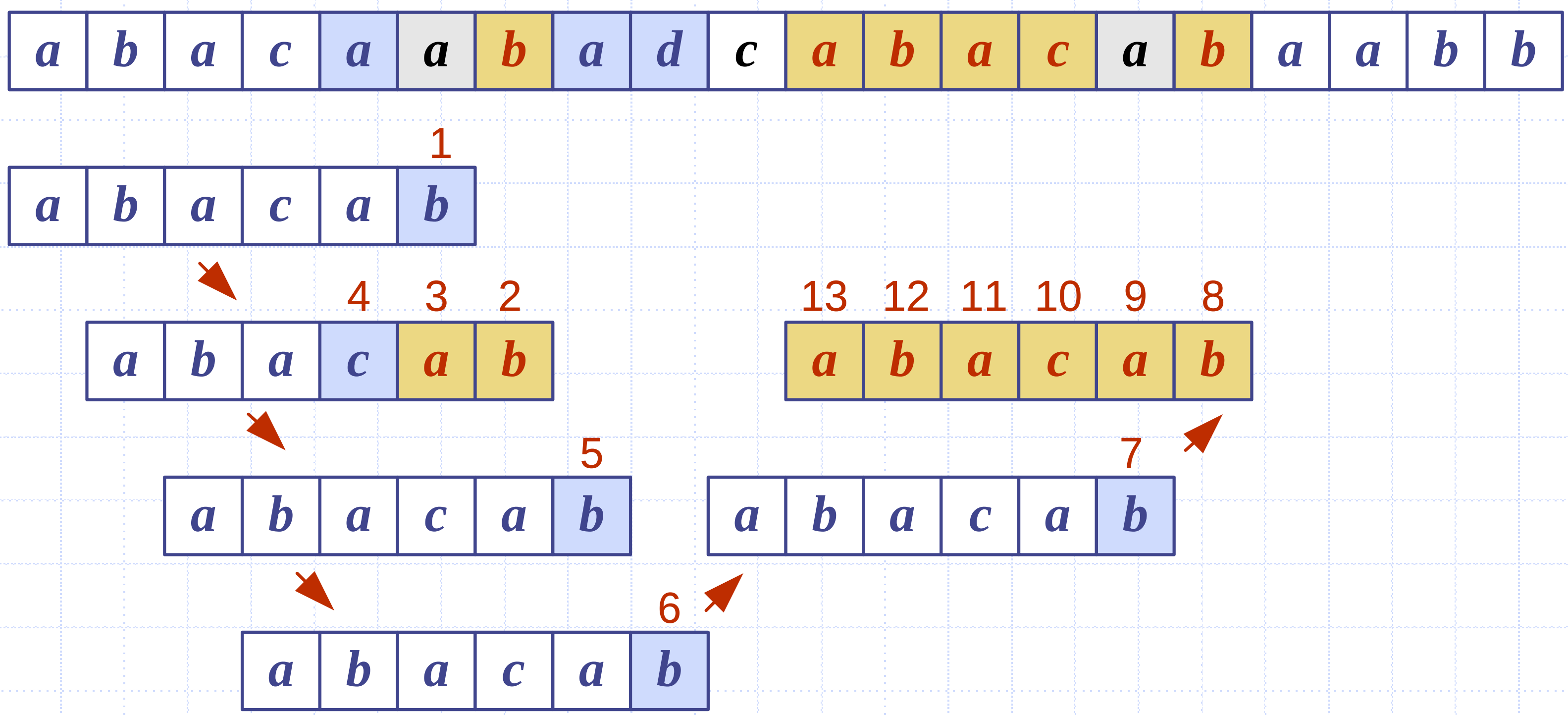
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

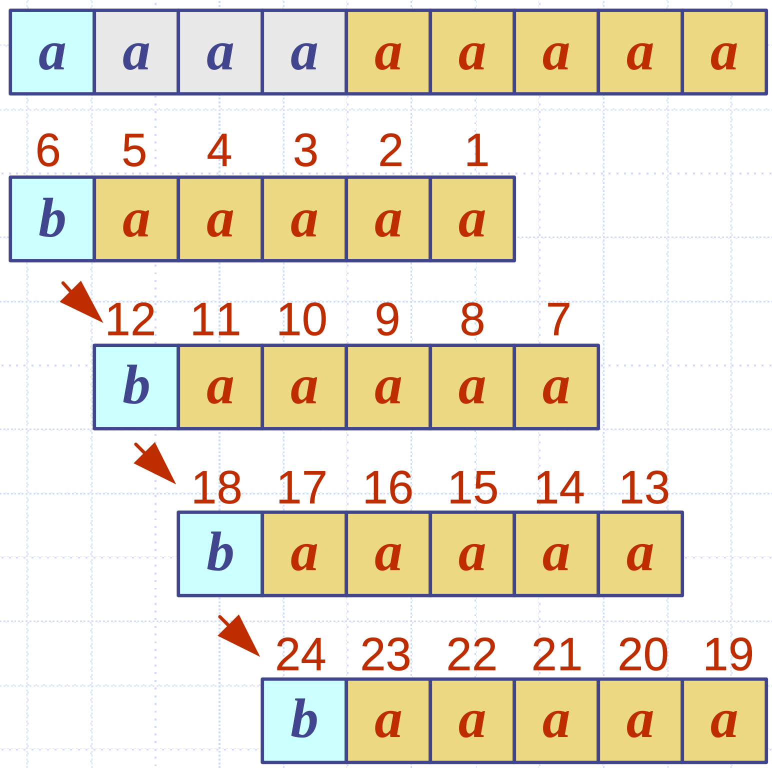


Example


a b a c a a b a d c a b a c a b a a b b
1
a b a c a b
4 3 2
a b a c a b
13 12 11 10 9 8
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

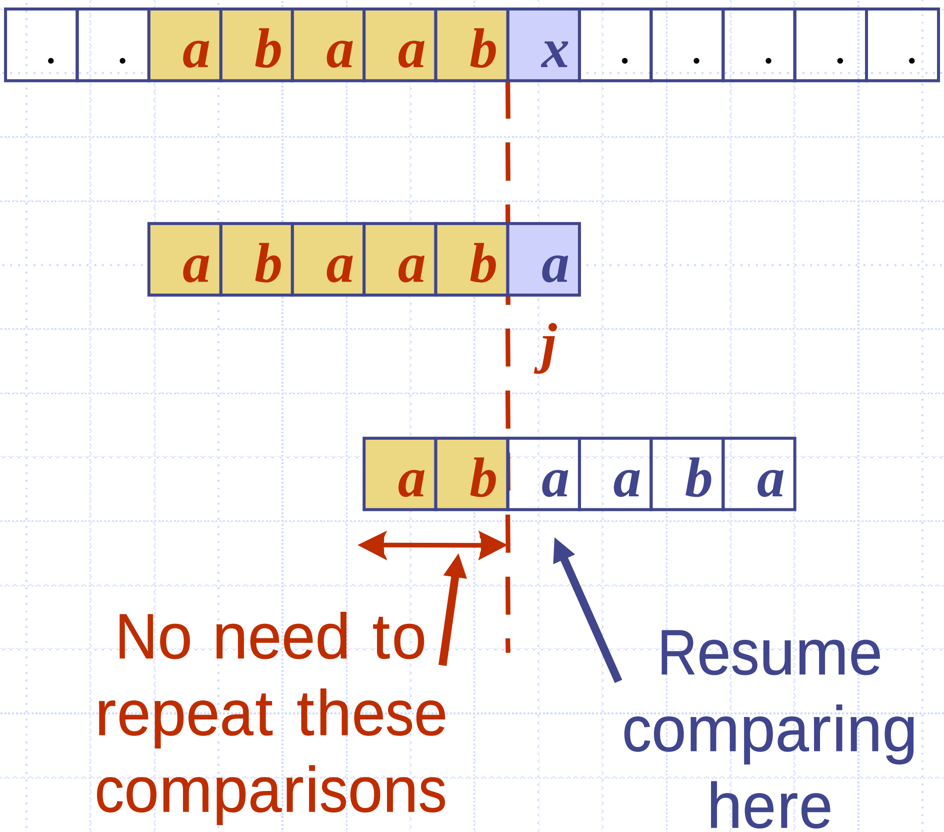
Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text



The KMP Algorithm - Motivation

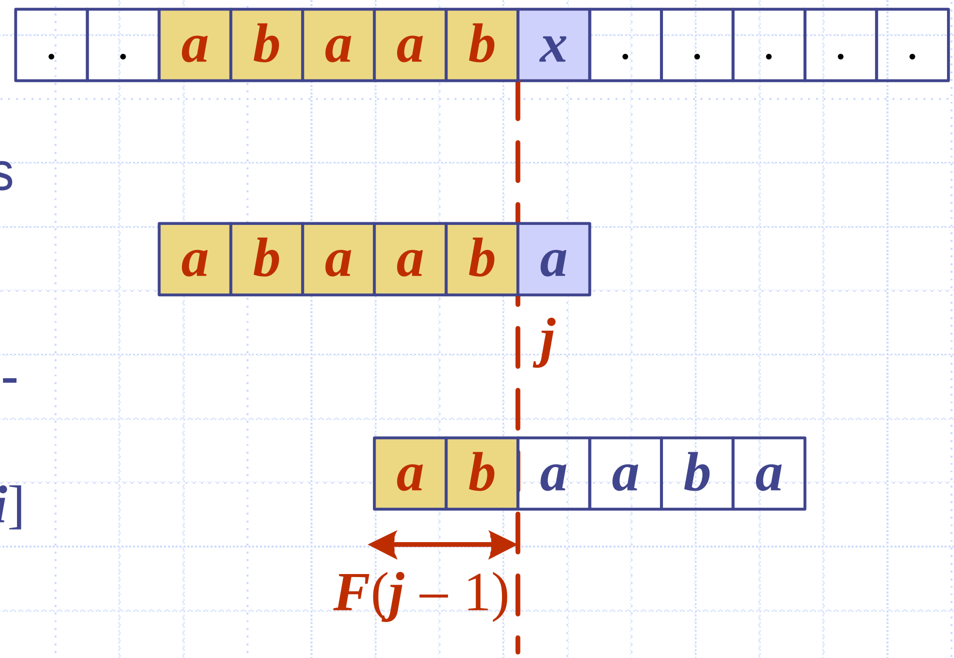
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix 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	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

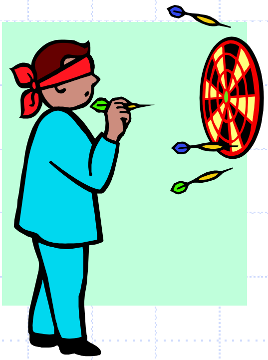


The KMP Algorithm

- 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*)
$F \leftarrow failureFunction(P)$
$i \leftarrow 0$
$j \leftarrow 0$
while $i < n$
  if $T[i] = P[j]$
    if $j = m - 1$
      return $i - j$ { match }
    else
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
  else
    if $j > 0$
      $j \leftarrow F[j - 1]$
    else
      $i \leftarrow i + 1$
return -1 { no match }

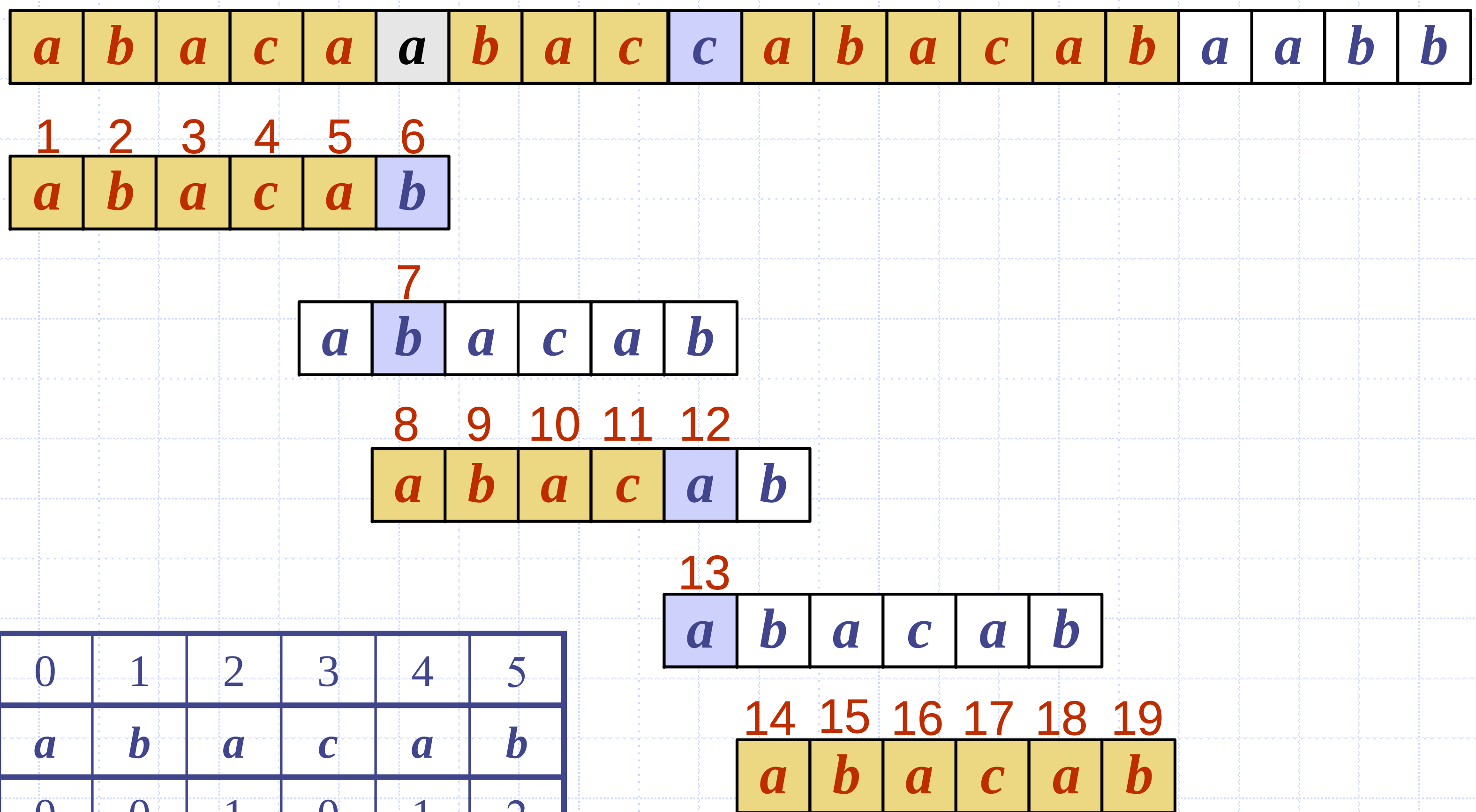
Computing the Failure Function



- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- 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

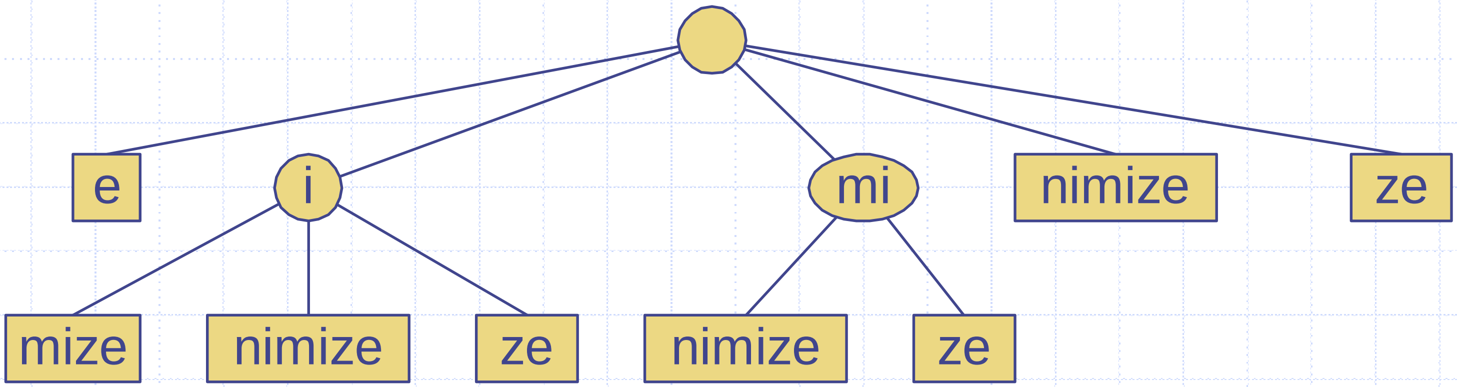
Algorithm *failureFunction*(P)
$F[0] \leftarrow 0$
$i \leftarrow 1$
$j \leftarrow 0$
while $i < m$
  if $P[i] = P[j]$
    {we have matched $j + 1$ chars}
    $F[i] \leftarrow j + 1$
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  else if $j > 0$ **then**
    {use failure function to shift P}
    $j \leftarrow F[j - 1]$
  else
    $F[i] \leftarrow 0$ { no match }
    $i \leftarrow i + 1$

Example



j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
F(*j*)	0	0	1	0	1	2

Tries



Outline and Reading

- Standard tries (§11.3.1)
- Compressed tries (§11.3.2)
- Suffix tries (§11.3.3)
- Huffman encoding tries (§11.4.1)

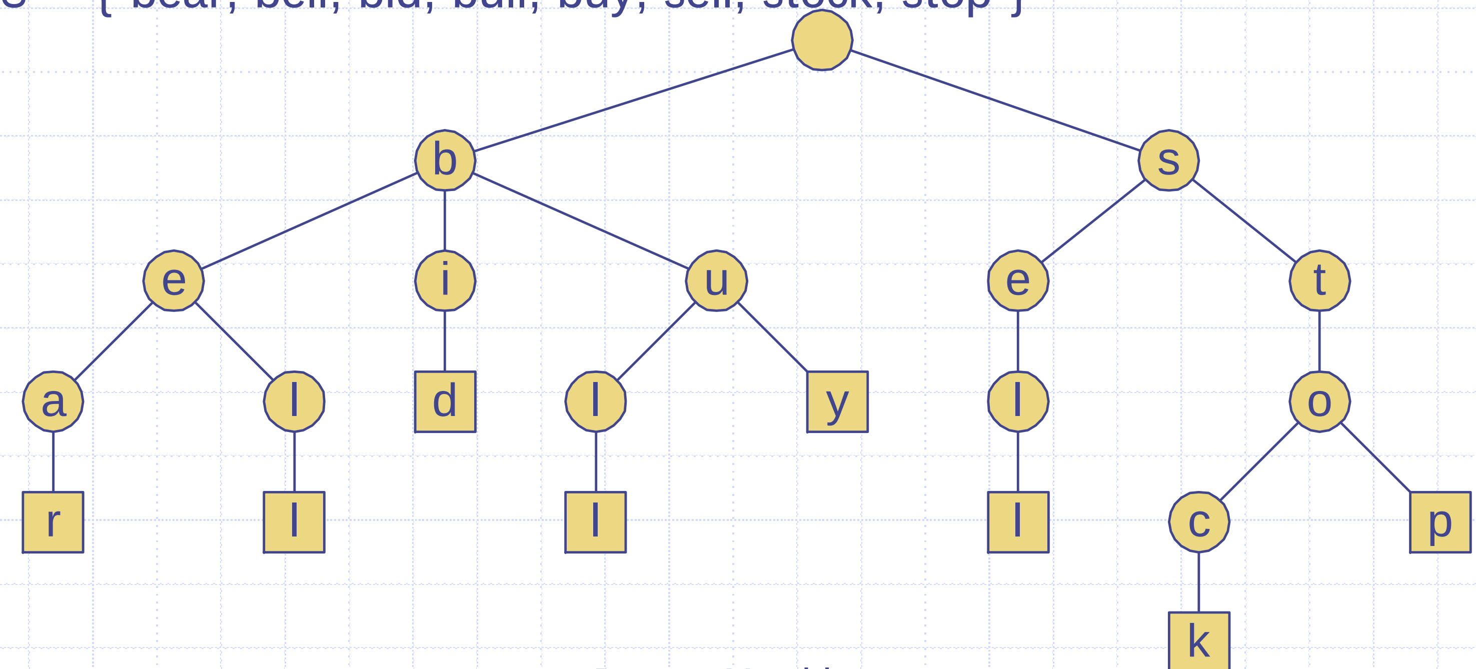
Preprocessing Strings

- 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 Trie (1)

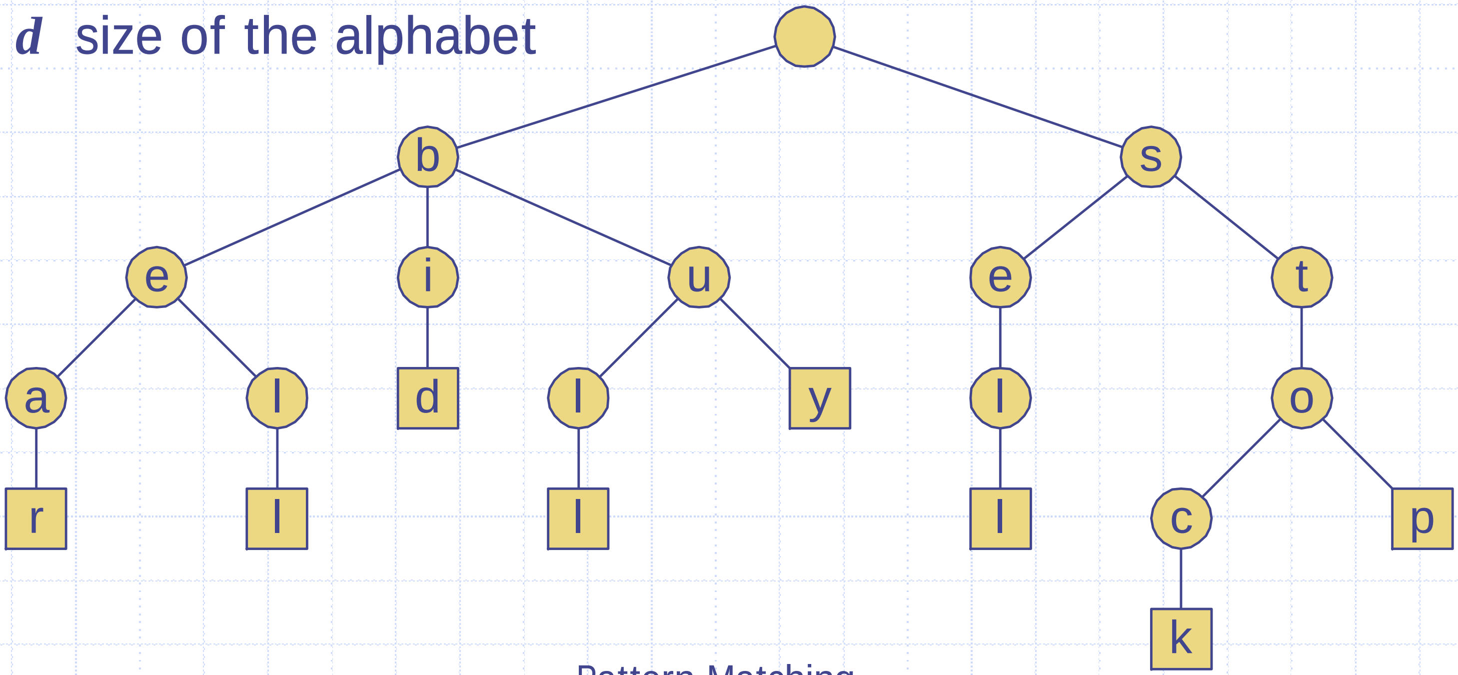
- The standard trie for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings

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



Standard Trie (2)

- 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

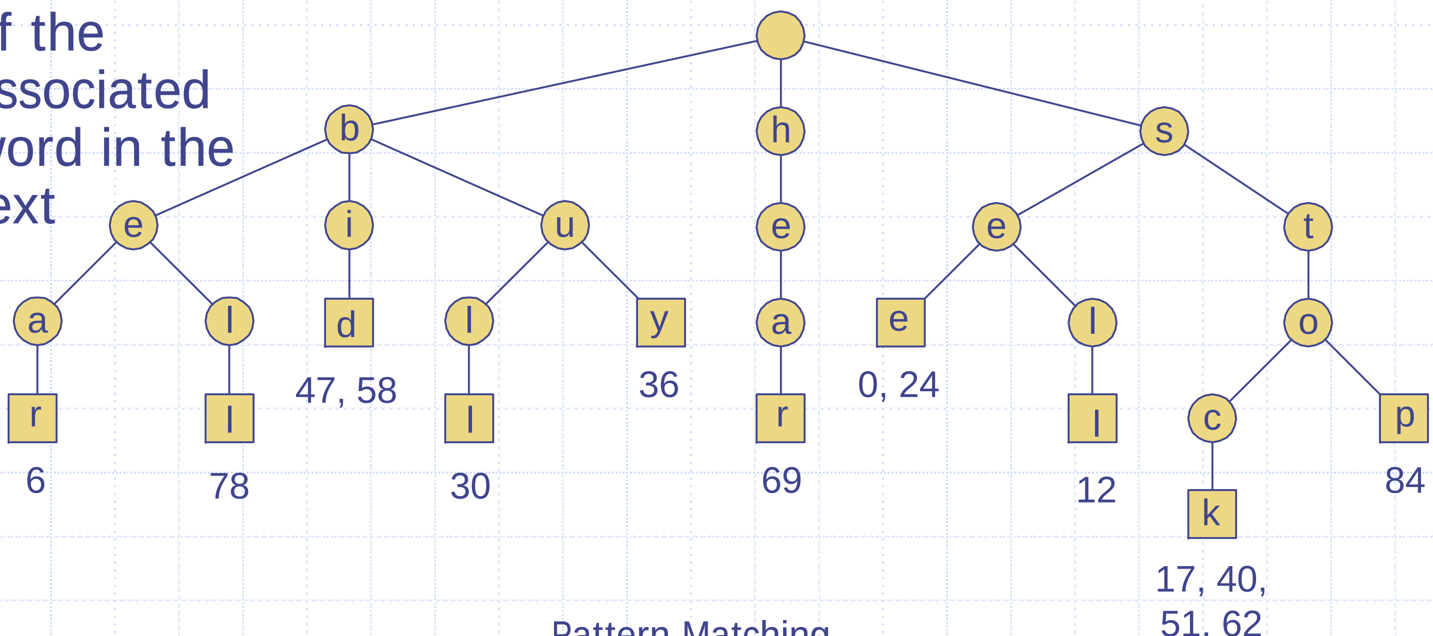
- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text

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

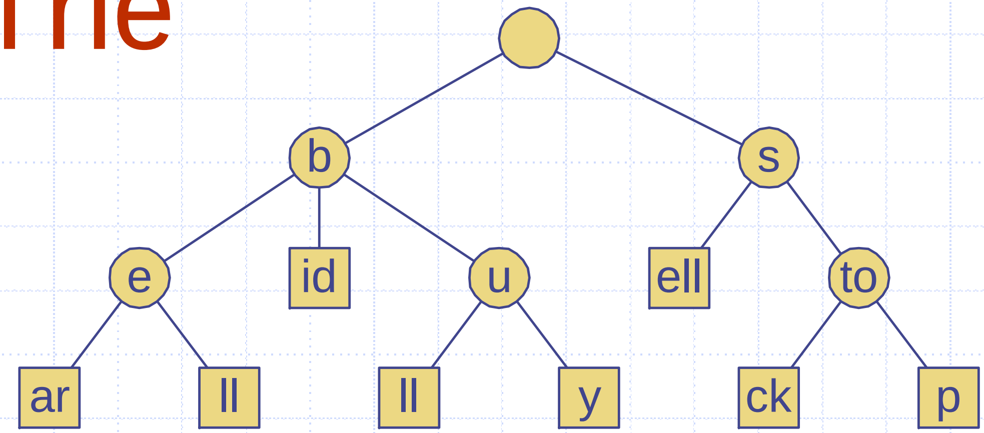
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

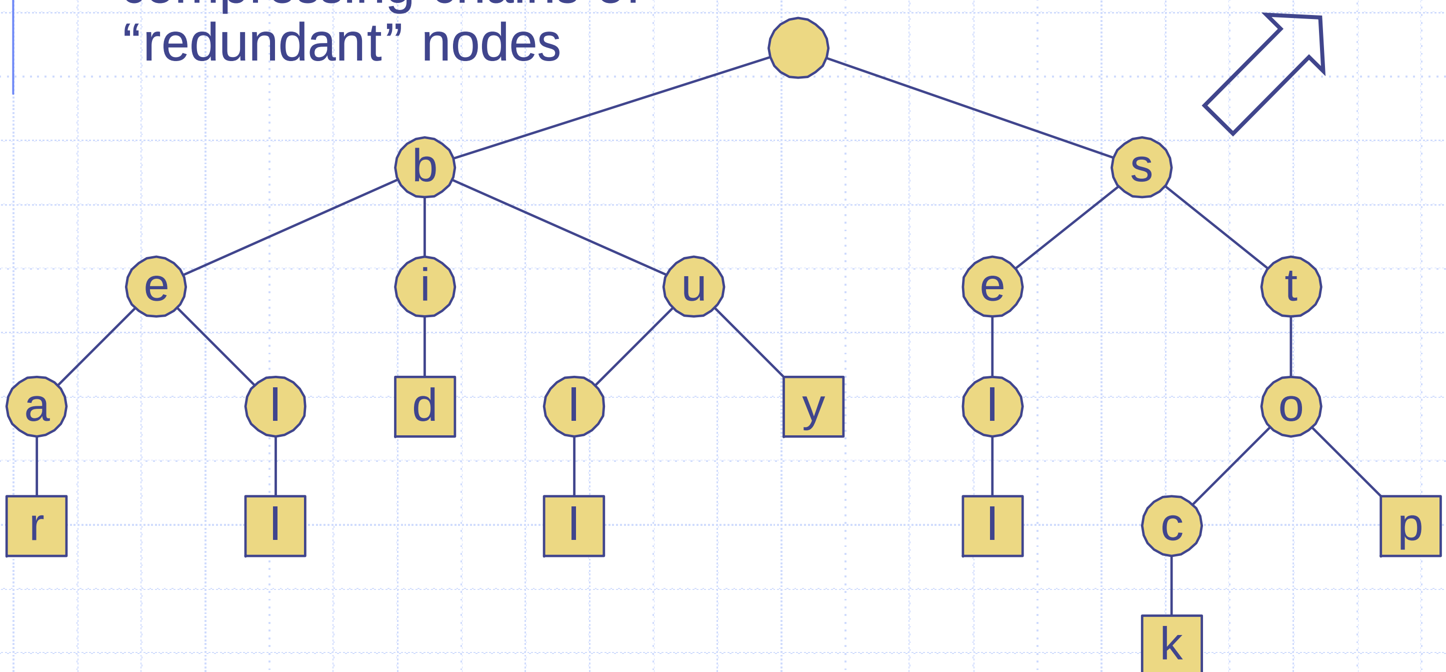
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88



Compressed Trie

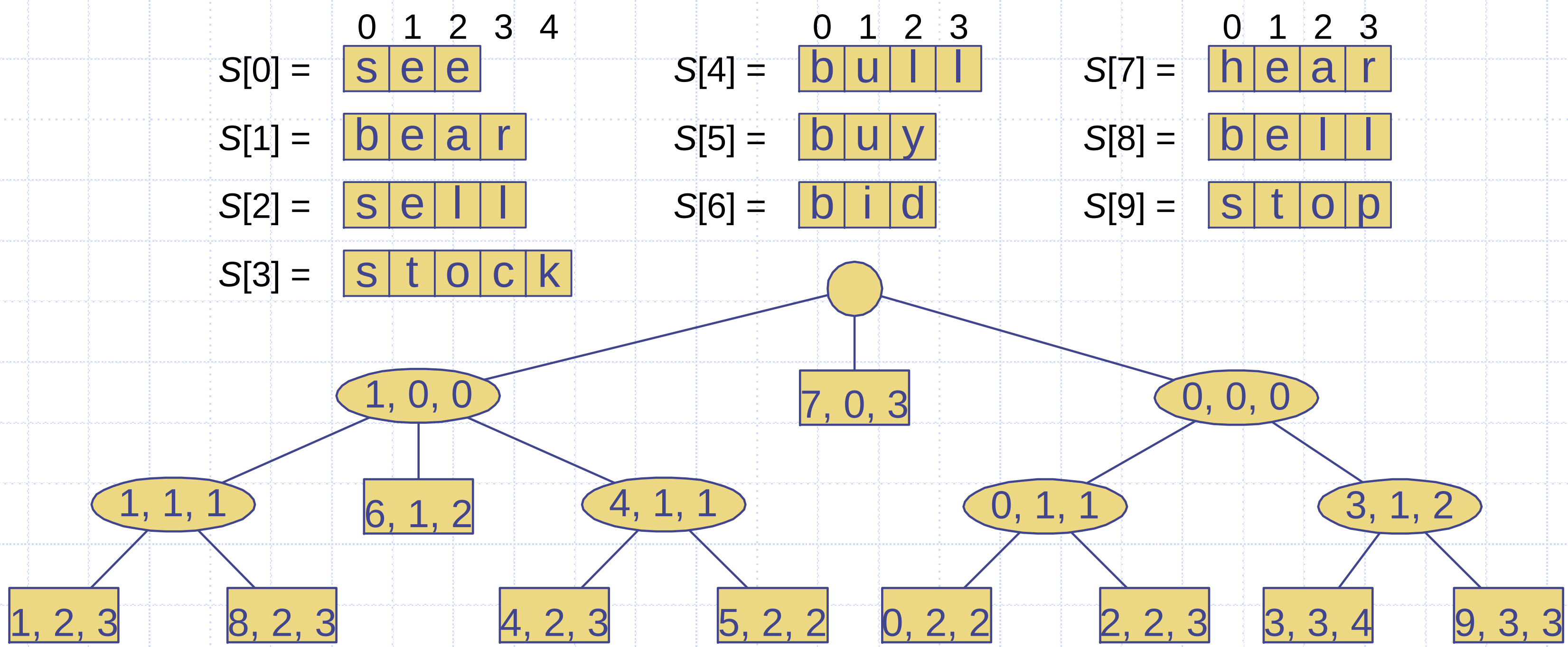
- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes





Compact Representation

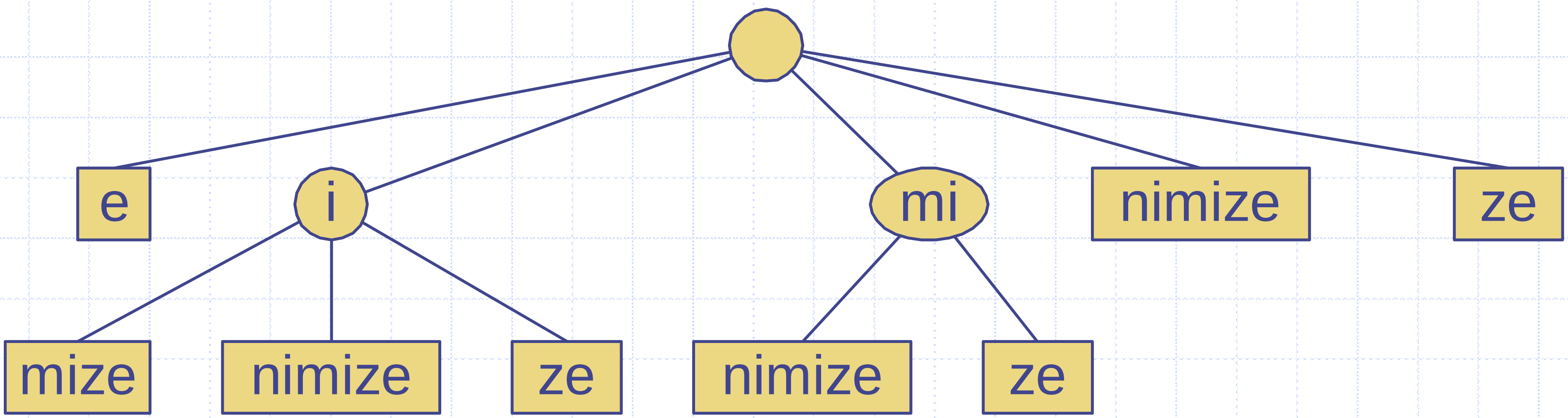
- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses $O(s)$ space, where s is the number of strings in the array
  - Serves as an auxiliary index structure



Suffix Trie (1)

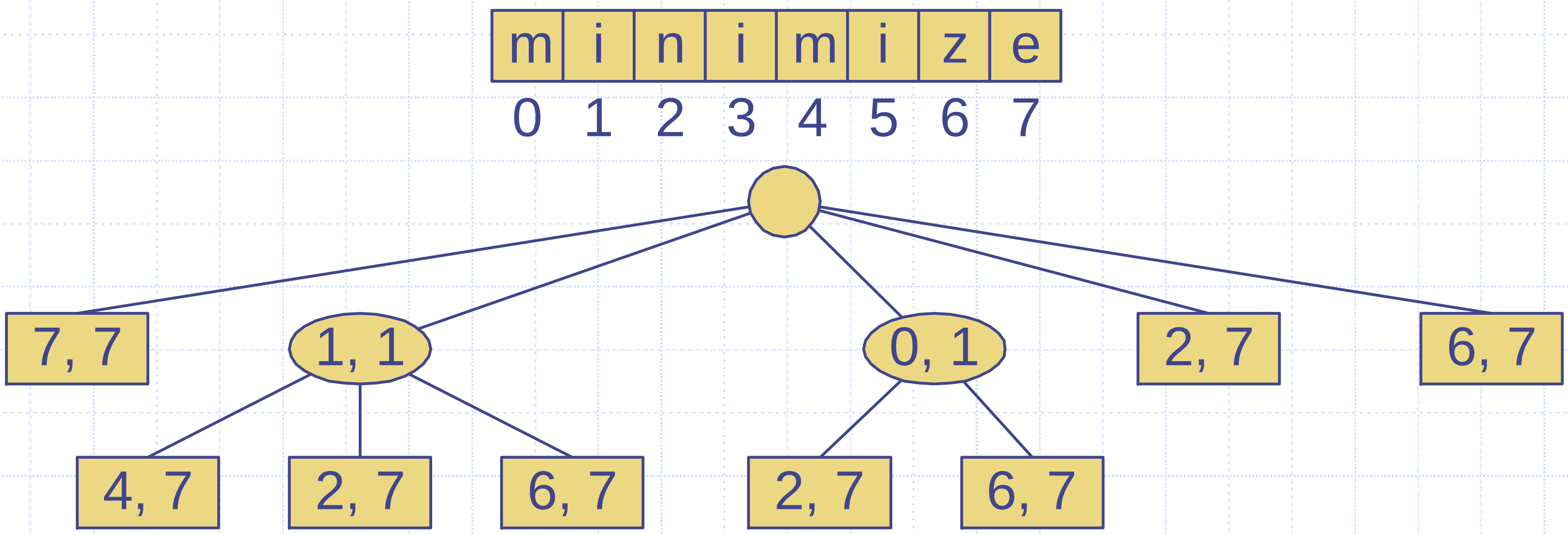
- The suffix trie of a string X is the compressed trie of all the suffixes of X

m	i	n	i	m	i	z	e
0	1	2	3	4	5	6	7



Suffix Trie (2)

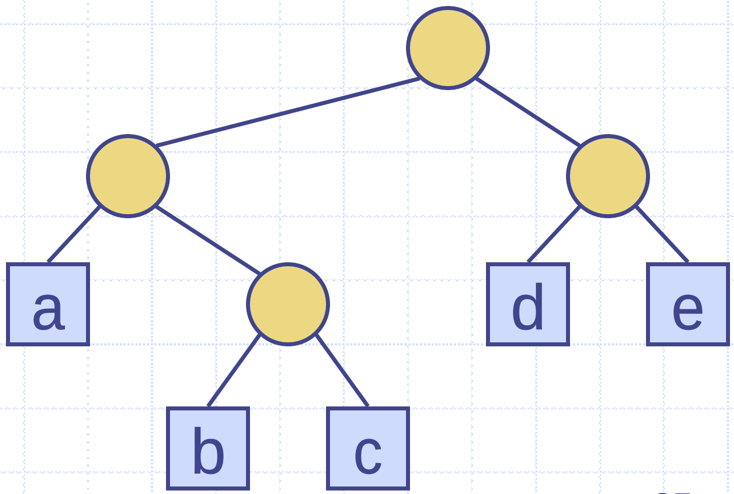
- Compact representation of the suffix trie for a string X of size n from an alphabet of size d
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in X in $O(dm)$ time, where m is the size of the pattern



Encoding Trie (1)

- 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 trie represents a prefix code
  - Each leaf stores a character
  - The code word of a character is given by the path from the root to the leaf storing the character (0 for a left child and 1 for a right child

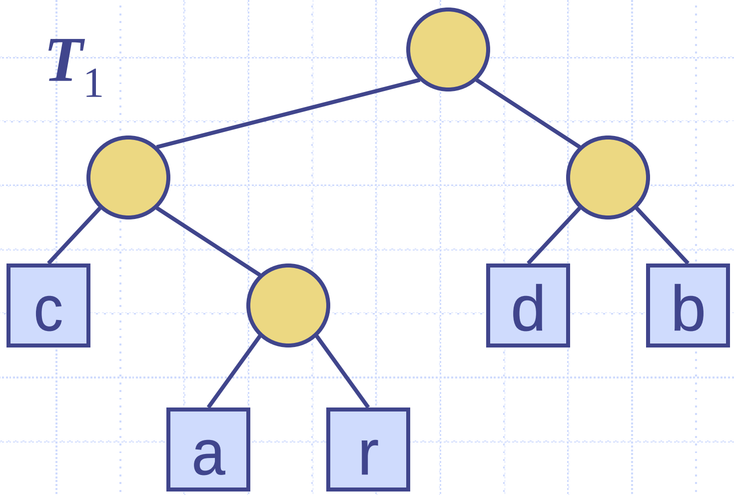
00	010	011	10	11
a	b	c	d	e



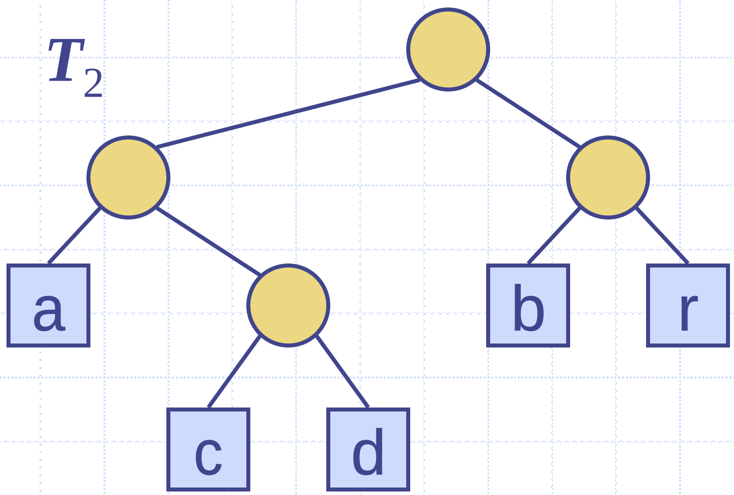
Encoding Trie (2)

- 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

T_1



T_2



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

Algorithm *HuffmanEncoding*(X)
    Input string X of size n
    Output optimal encoding trie for X
    $C \leftarrow$ *distinctCharacters*(X)
    computeFrequencies(C, X)
    $Q \leftarrow$ new empty heap
    for all $c \in C$
        $T \leftarrow$ new single-node tree storing c
        Q.*insert*(*getFrequency*(c), T)
    while Q.*size*() > 1
        $f_1 \leftarrow Q$.*minKey*()
        $T_1 \leftarrow Q$.*removeMin*()
        $f_2 \leftarrow Q$.*minKey*()
        $T_2 \leftarrow Q$.*removeMin*()
        $T \leftarrow$ *join*(T_1, T_2)
        Q.*insert*($f_1 + f_2$, T)
    return Q.*removeMin*()

Example

X = abracadabra

Frequencies

a	b	c	d	r
5	2	1	1	2