CH12. STRINGS AND DYNAMIC PROGRAMMING

CSED233 Data Structure

Prof. Hwanjo Yu
POSTECH

String Operations

- String and substring
  - *String* P = *"I am a boy" where the size of* P *is* m
  - *Substring* $P[i..j]$ = *"am a"*
  - *Prefix* $P[0..i]$ = *"I am a"*
  - *Suffix* $P[i..m-1]$ = *"a boy"*
  - *Alphabet:* Σ
  - *Size of alphabet:* $|\Sigma|$
- The STL String Class
  - *size()*
  - *empty()*
  - *operator[i]*
  - *at(i)*
  - *insert(i, Q)*
  - *append(Q)*
  - *erase(i,m)*
  - *substr(i,m)*
  - *find(Q)*
  - *c_str()*

Example 12.1: *Consider the following series of operations, which are performed on the string* S = "`abcdefghijklmnop`":

Operation	*Output*
S.size()	16
S.at(5)	'f'
S[5]	'f'
S + "qrs"	"abcdefghijklmnopqrs"
S == "abcdefghijklmnop"	**true**
S.find("ghi")	6
S.substr(4, 6)	"efghij"
S.erase(4, 6)	"abcdklmnop"
S.insert(1, "xxx")	"axxxbcdklmnop"
S += "xy"	"axxxbcdklmnopxy"
S.append("z")	"axxxbcdklmnopxyz"

Dynamic Programming

- Matrix Chain-Product problem
  - $A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1}$ *where* A_i *is a* $d_i \times d_{i+1}$ *matrix*
  - *Determine the parenthesization of expression which minimizes the total number of scalar multiplications*
  - *For example,* B *is* 2×10, C *is* 10×50, D *is* 50×20
    - $B \cdot (C \cdot D) = 2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10{,}400$ multiplcations
    - $(B \cdot C) \cdot D = 2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3{,}000$ multiplications
  - *Enumerate all the possible ways of parenthesizing: # of ways = # of binary trees that have n external nodes = exponential in n*
- **Subproblem optimality**
  - *Characterize the optimal solution to the problem in terms of optimal solutions to its subproblems.*
  - *The full parenthesization of* $A_i \cdots A_j$ *is in the form of* $(A_i \cdots A_k)(A_{k+1} \cdots A_j)$, *and* $(A_i \cdots A_k)$ *and* $(A_{k+1} \cdots A_j)$ *must be solved optimally for its parent solution to be optimal.*
  - *The optimal parent solution must be one of* $(A_0)(A_1 \cdots A_{n-1})$, $(A_0 \cdot A_1)(A_2 \cdots A_{n-1})$, *…, or* $(A_0 \cdots A_{n-2})(A_{n-1})$ *where each* $(\cdots)$ *is optimally solved.*
  - $N_{i,j}$*: optimal # of multiplications for* $A_i \cdots A_j$
  - $N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$ *where* $N_{i,i} = 0$
  - *int N(i, j)*
    - If i=j then return 0
    - minN = infinite
    - For k = i to j
      - *minN = min(minN, N(i,k)+N(k+1, j)+d[i]*d[k+1]*d[j+1])*
    - return minN
  - *How many N(i,j) will be called?*

Dynamic Programming

- **Overlapping subproblems** (memorization)
  - *Before calling N(i,j), check if it is already computed.*
  - *If already computed, use it.*
  - *How to modify the following codes to apply the overlapping subproblems?*
  - *int N(i, j)*
    - If i=j then return 0
    - minN = infinite
    - For k = i to j
      - *minN = min(minN, N(i,k)+N(k+1, j)+d[i]*d[k+1]*d[j+1])*
    - return minN
- How to implement dynamic programming?
  1. *Find the **recurrence relation** which characterizes the optimal solution to the problem in terms of optimal solutions to its subproblems.*
  2. *Implement it using **recursive function***
  3. *Apply the **memorization***
- **Top-down design but bottom-up executions!**
- What about bottom-up design?
  - *The designs in the textbook are bottom-up.*
  - *Hard to code, hard to understand*

Dynamic Programming: LCS problem

- LCS (longest common subsequence) problem
  - *Y is a subsequence of X if Y is a sequence of characters that are not necessarily contiguous but are taken in order from X*
  - *AAAG is a subsequence of CGATAATTGAGA*
  - *Given two strings X and Y over some alphabet (e.g., {A,C,G,T}), find a longest string S that is a subsequence of both X and Y.*
  - *How many subsequence of X where $|X| = n$?*
  - *Brute-force approach takes $O(m*2^n)$ where $|Y| = m$*
  - *Design a polynomial time algorithm using dynamic programming!*
- Subproblem optimality
  - *X and Y of length n and m*
  - $L[i,j]$*: length of longest subsequence of X[0..i] and Y[0..j]*
  - $L[i,j] = L[i-1,j-1]+1$ *if* $X[i] = Y[j]$
  - $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$ *if* $X[i] \neq Y[j]$
  - $L[-1,j] = L[i,-1] = 0$
  - *E.g., X="atta" and Y="attatt"*
  - *int L(i, j)*
    - If i = -1 or j = -1 then return 0
    - If X[i] = Y[i] then return L(i-1,j-1)+1
    - Else return max(L(i-1,j), L(i,j-1))
- Apply the memorization!

Dynamic Programming

- Dynamic Programming
  - *Algorithm-design technique for optimization problem*
  - *Optimization problem: find min or max F(x)*
  - *Similar to divide-and-conquer*
  - *Solve “hard-looking” problem in polynomial time*
  - *Typically need just a few lines of codes (using recursive function)*
  - *Design top-down (but executed bottom-up)*

Pattern Matching Algorithms

- Given a string T and pattern P, find whether P is a substring T. (find(P) in STL)
- Brute Force: $O(nm)$ where $n = |T|$ and $m = |P|$

Algorithm BruteForceMatch(T, P):
  Input: Strings T (text) with n characters and P (pattern) with m characters
  Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

  for $i \leftarrow 0$ **to** $n - m$ {for each candidate index in T} **do**
    $j \leftarrow 0$
    while $(j < m$ **and** $T[i + j] = P[j])$ **do**
      $j \leftarrow j + 1$
    if $j = m$ **then**
      return i
  return "There is no substring of T matching P."

Code Fragment 12.3: Brute-force pattern matching.

Example 12.4: *Suppose we are given the text string*

$$T = \texttt{"abacaabaccabacabaabb"}$$

and the pattern string

$$P = \texttt{"abacab"}.$$

In Figure 12.3, we illustrate the execution of the brute-force pattern matching algorithm on T and P.

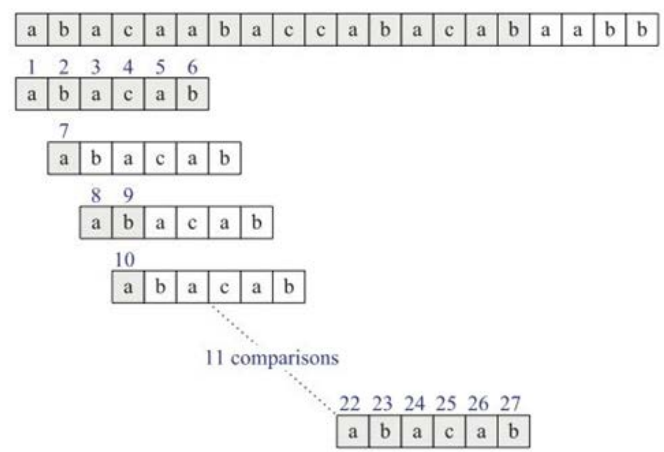


Figure 12.3: Example run of the brute-force pattern matching algorithm. The algorithm performs 27 character comparisons, indicated above with numerical labels.

The Boyer-Moore (BM) Algorithm

- Works better when alphabet is finite and text is relatively long, e.g., searching words in documents
- Two heuristics (no improvement in time complexity)
  - *Looking-Glass heuristic : comparing backward from the end of P*
  - *Character-Jump heuristic : when mismatch, jump multiple characters using last(c)*
    - last(c) : the index of the last (right-most) occurrence of c in P
    - If no c in P, last(c) = -1
    - Define last(c) for every character in alphabet Σ

```
Algorithm BMMatch(T, P):
  Input: Strings T (text) with n characters and P (pattern) with m characters
  Output: Starting index of the first substring of T matching P, or an indication
    that P is not a substring of T
  compute function last
  i ← m − 1
  j ← m − 1
  repeat
    if P[j] = T[i] then
      if j = 0 then
        return i          {a match!}
      else
        i ← i − 1
        j ← j − 1
    else
      i ← i + m − min(j, 1 + last(T[i]))          {jump step}
      j ← m − 1
  until i > n − 1
  return "There is no substring of T matching P."
```

Code Fragment 12.4: The Boyer-Moore pattern matching algorithm.

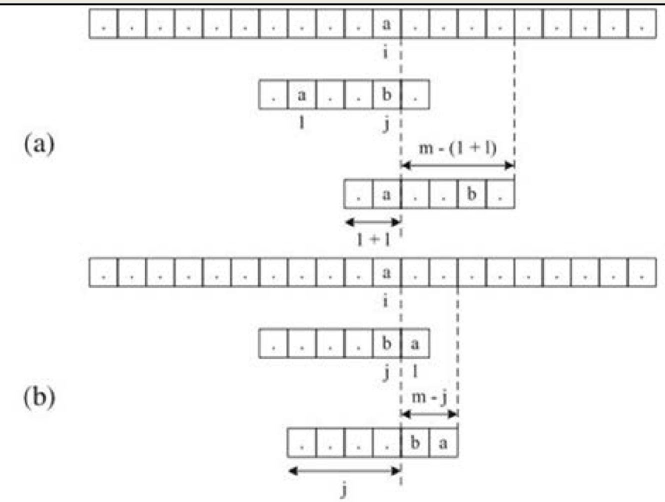
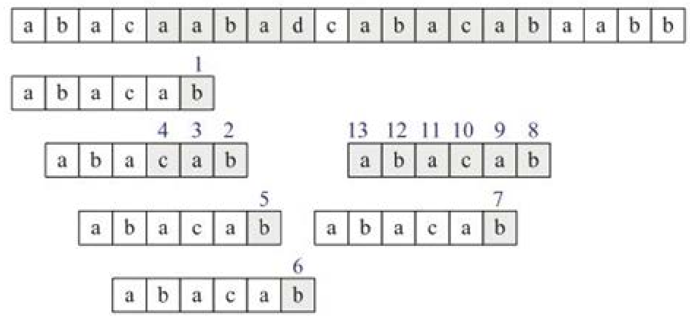


Figure 12.4: The jump step in the algorithm of Code Fragment 12.4, where we let $l = \text{last}(T[i])$. We distinguish two cases: (a) $1 + l \le j$, where we shift the pattern by $j - l$ units; (b) $j < 1 + l$, where we shift the pattern by one unit.

The Boyer-Moore (BM) Algorithm



The last(c) function:

c	a	b	c	d
last(c)	4	5	3	−1

Figure 12.5: The BM pattern matching algorithm. The algorithm performs 13 character comparisons, which are indicated with numerical labels.

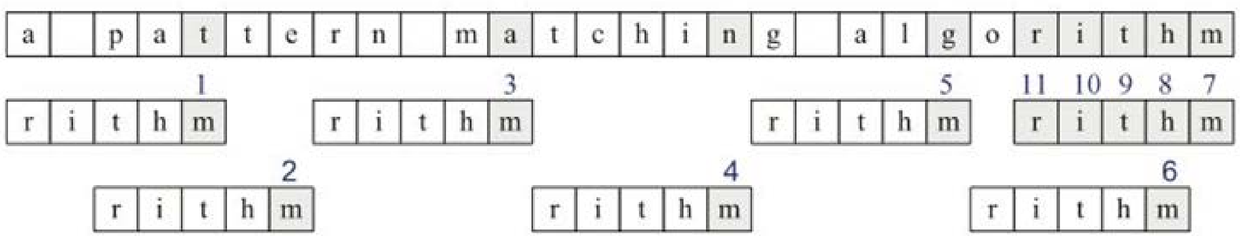


Figure 12.6: An example of a Boyer-Moore execution on English text.

The Knuth-Morris-Pratt (KMP) Algorithm

- When no match found on a character, don't throw away the comparison information so far => Worst case time complexity is $O(n+m)$
- Failure function $f(j)$: length of the longest prefix of P that is a suffix of $P[1..j]$ (not $P[0..j]$)
  - *It "encodes" repeated substrings inside P*
  - $P = "abacab"$

j	0 1 2 3 4 5
$P[j]$	a b a c a b
$f(j)$	0 0 1 0 1 2

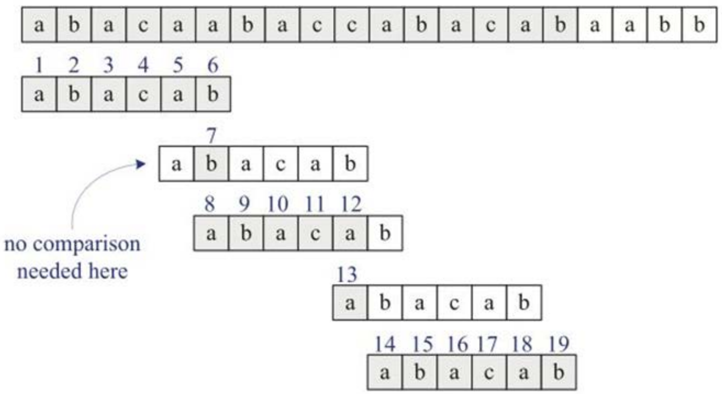


Figure 12.7: The KMP pattern matching algorithm. The failure function f for this pattern is given in Example 12.5. The algorithm performs 19 character comparisons, which are indicated with numerical labels.

```
Algorithm KMPMatch(T, P):
  Input: Strings T (text) with n characters and P (pattern) with m characters
  Output: Starting index of the first substring of T matching P, or an indication
    that P is not a substring of T
  f ← KMPFailureFunction(P)      {construct the failure function f for P}
  i ← 0
  j ← 0
  while i < n do
    if P[j] = T[i] then
      if j = m − 1 then
        return i − m + 1      {a match!}
      i ← i + 1
      j ← j + 1
    else if j > 0 {no match, but we have advanced in P} then
      j ← f(j − 1)      {j indexes just after prefix of P that must match}
    else
      i ← i + 1
  return "There is no substring of T matching P."
```

Code Fragment 12.6: The KMP pattern matching algorithm.

Huffman code

- ASCII code (7 bits) or Unicode (16 bits) are fixed-length coding systems
- Huffman coding is variable-length coding
  - *Encode high frequency characters with short code-word strings and encode low frequency characters with long code-word strings*
  - *Prefix code: no code word is a prefix of another code word*
  - *Optimal prefix coding*
  - *Greedy method*
  - *Greedy-choice property*

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 Q.size() > 1 do
   f1 ← Q.min()
   T1 ← Q.removeMin()
   f2 ← Q.min()
   T2 ← Q.removeMin()
   Create a new binary tree T with left subtree T1 and right subtree T2.
   Insert T into Q with key f1 + f2.
return tree Q.removeMin()
```

Code Fragment 12.9: Huffman-coding algorithm.

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

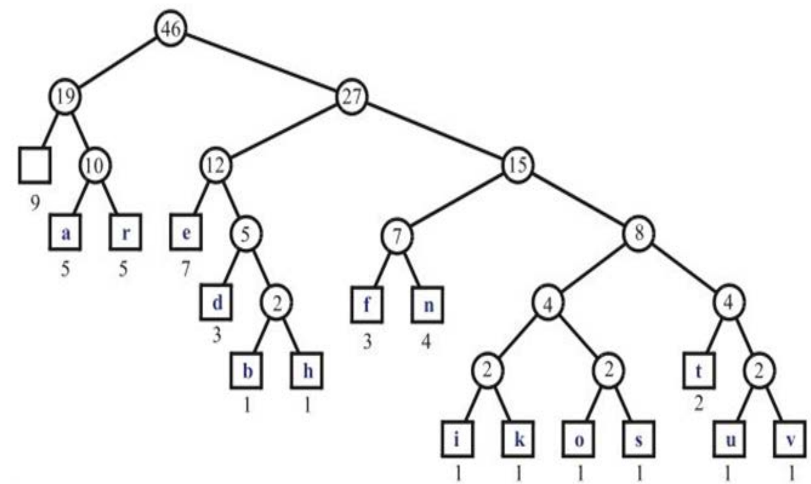


Figure 12.8: An example Huffman code for the input string X = "a fast runner need never be afraid of the dark": (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for "a" is 010, and the code for "f" is 1100.

Tries (or Prefix tree)

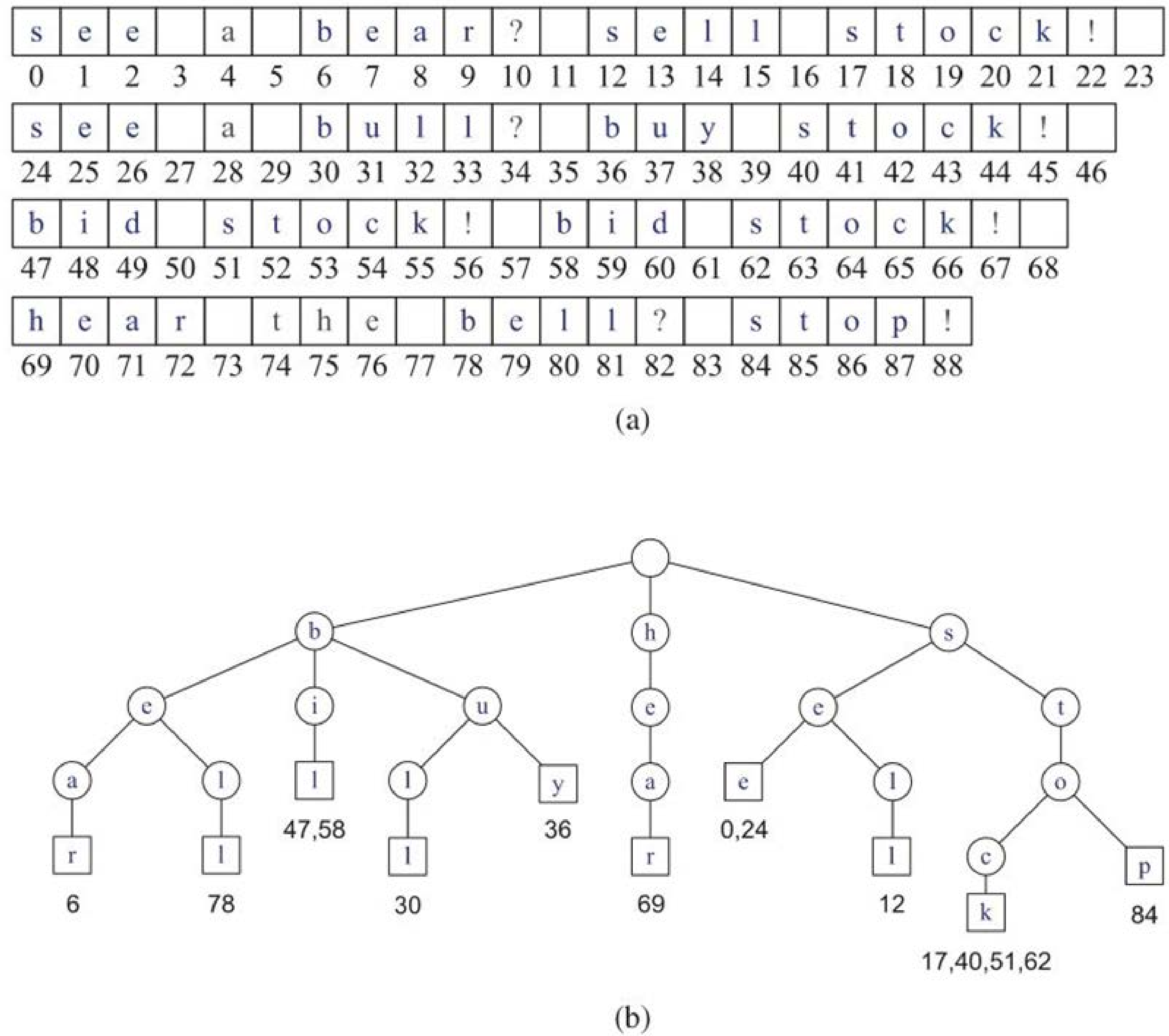


Figure 12.10: Word matching and prefix matching with a standard trie: (a) text to be searched; (b) standard trie for the words in the text (articles and prepositions, which are also known as ***stop words***, excluded), with external nodes augmented with indications of the word positions.

Tries

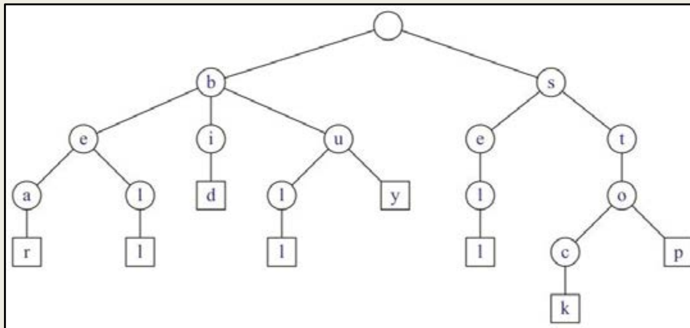


Figure 12.9: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

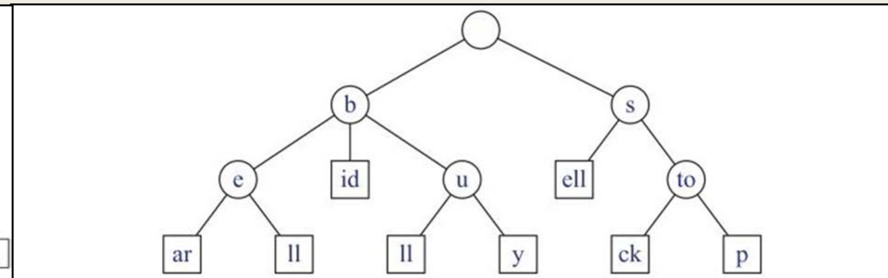


Figure 12.11: Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}. Compare this with the standard trie shown in Figure 12.9.

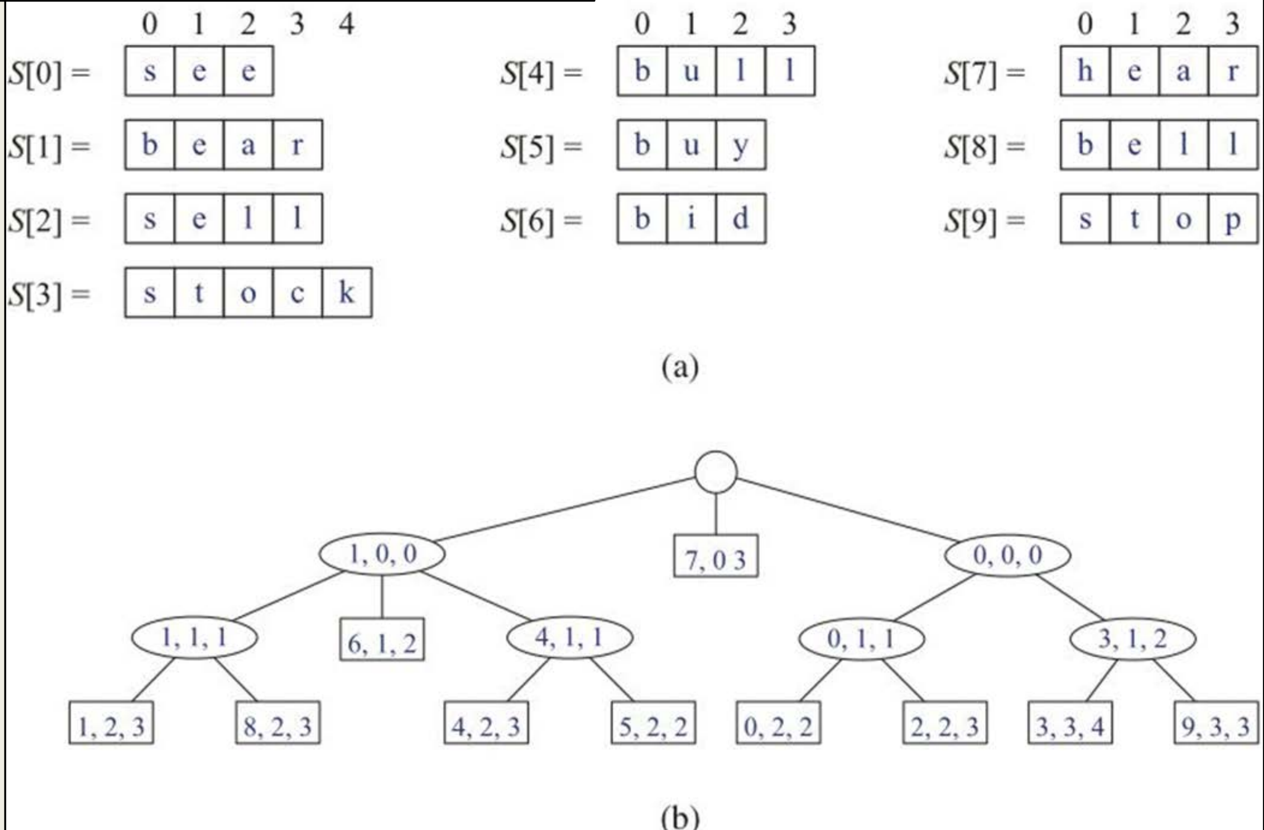


Figure 12.12: (a) Collection S of strings stored in an array. (b) Compact representation of the compressed trie for S.

Suffix Trie (or Suffix Tree)

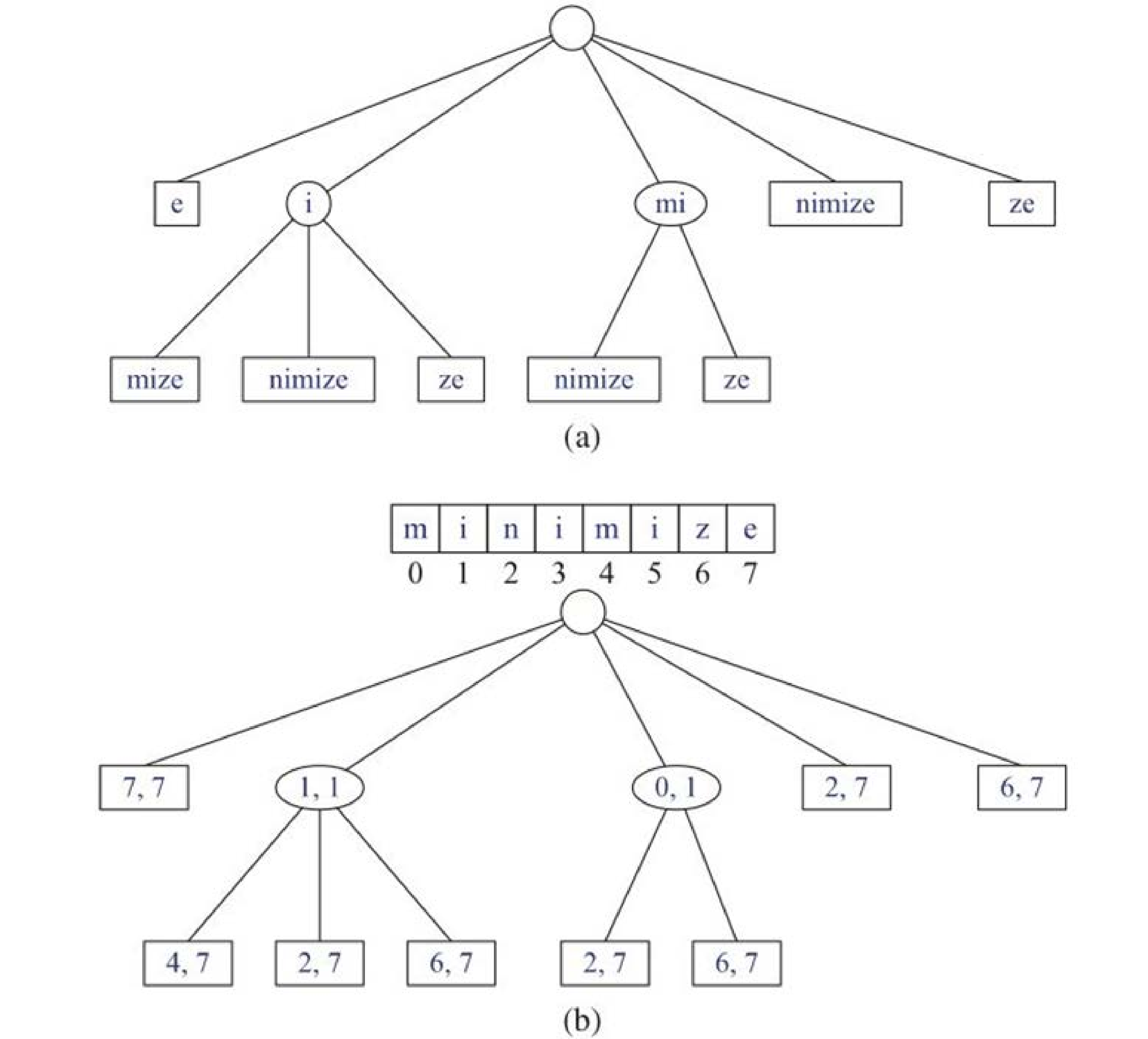


Figure 12.13: (a) Suffix trie T for the string X = ``minimize''. (b) Compact representation of T, where pair (i, j) denotes $X[i..j]$.

Search Engines

- Web crawler
- Inverted index
  - *Compressed trie for terms*
  - *Occurrence list for each external node*
- Multiple keywords query?
  - *Intersection*
- Ranking