CSE221

Lecture 22:
Strings and Pattern Matching

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Tries
- Compressed Tries
- Pattern matching algorithms
  - Brute force
  - Boyer-Moore
  - Knuth-Norris-Pratt

Outline

- Tries
- Compressed Tries
- Pattern matching algorithms
  - Brute force
  - Boyer-Moore
  - Knuth-Norris-Pratt

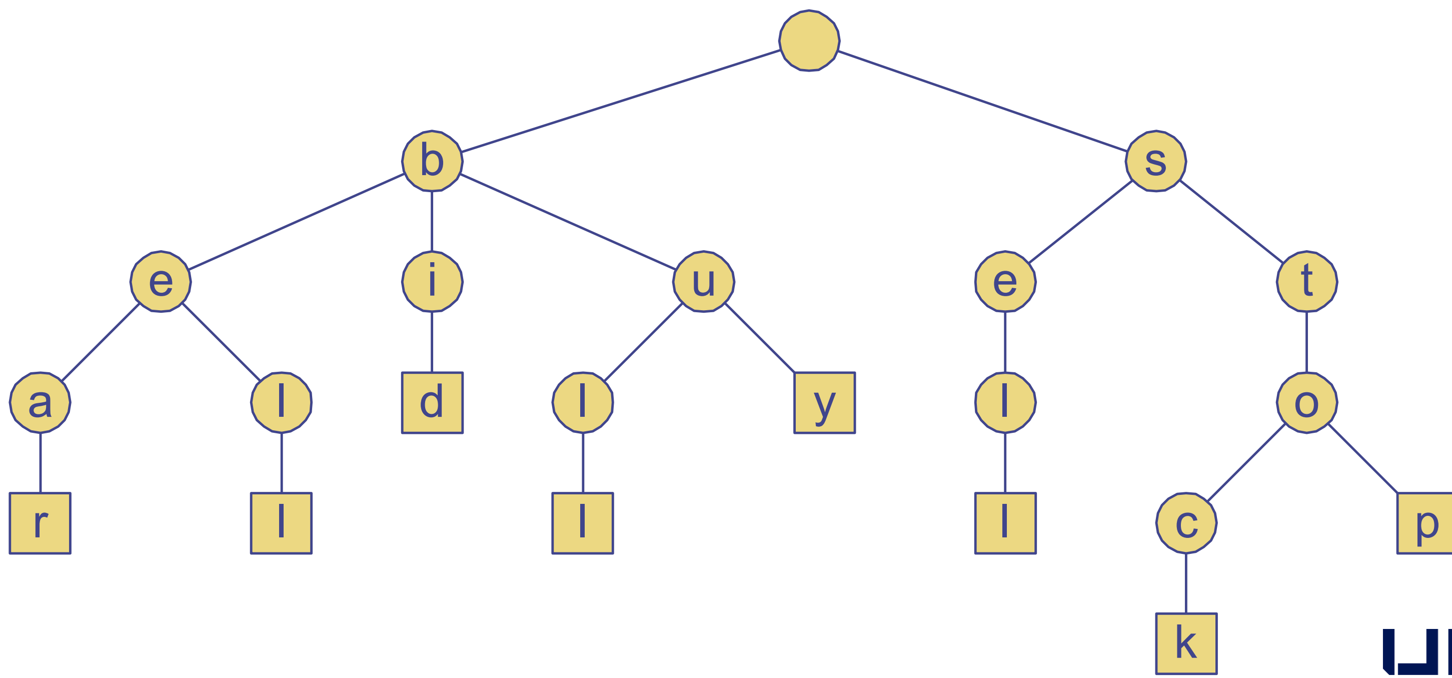
Preprocessing Strings

- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - E.g., S = { bear, bell, bid, bull, buy, sell, stock, stop }
- A trie is also called a digital tree, a radix tree, or a prefix tree.
- A trie can be considered as a search tree in which the keys are strings.
- A tries supports pattern matching queries (e.g., whether a word exists in an article) in time proportional to the pattern size.

Standard Tries

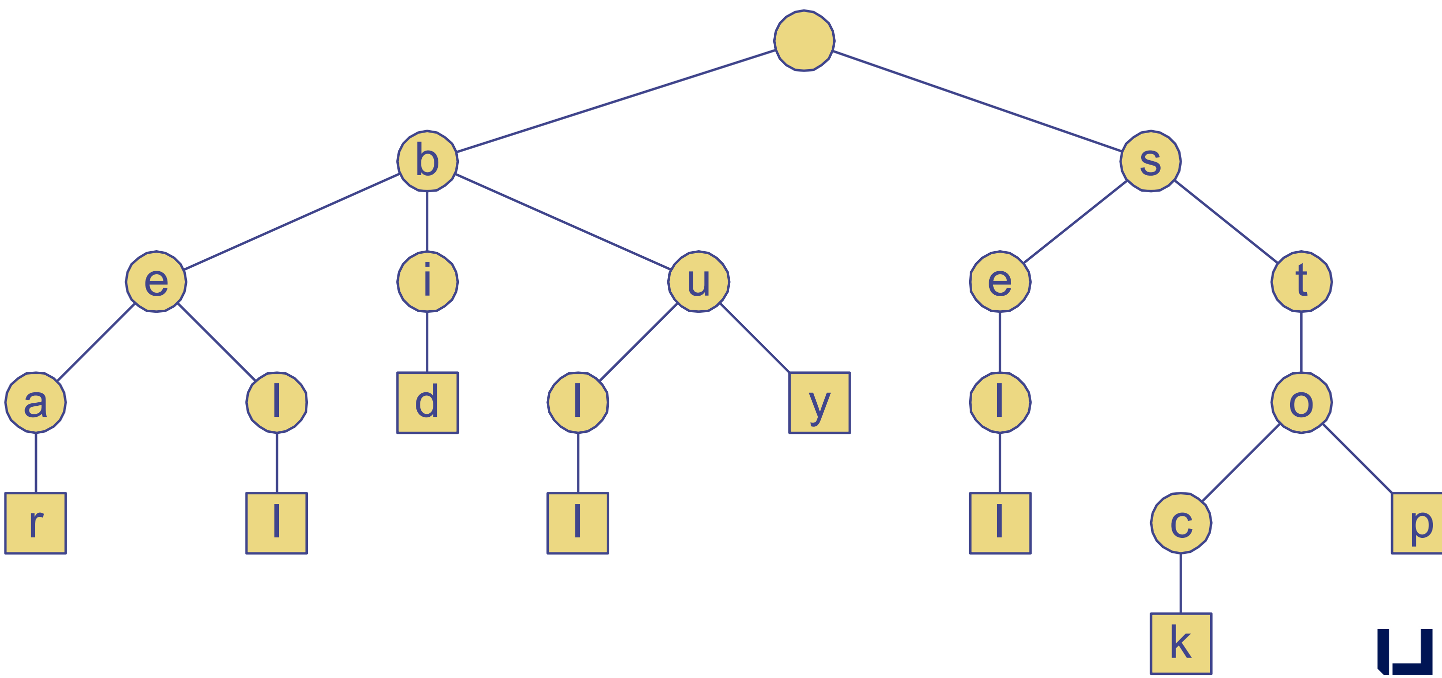
- The standard trie for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings

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



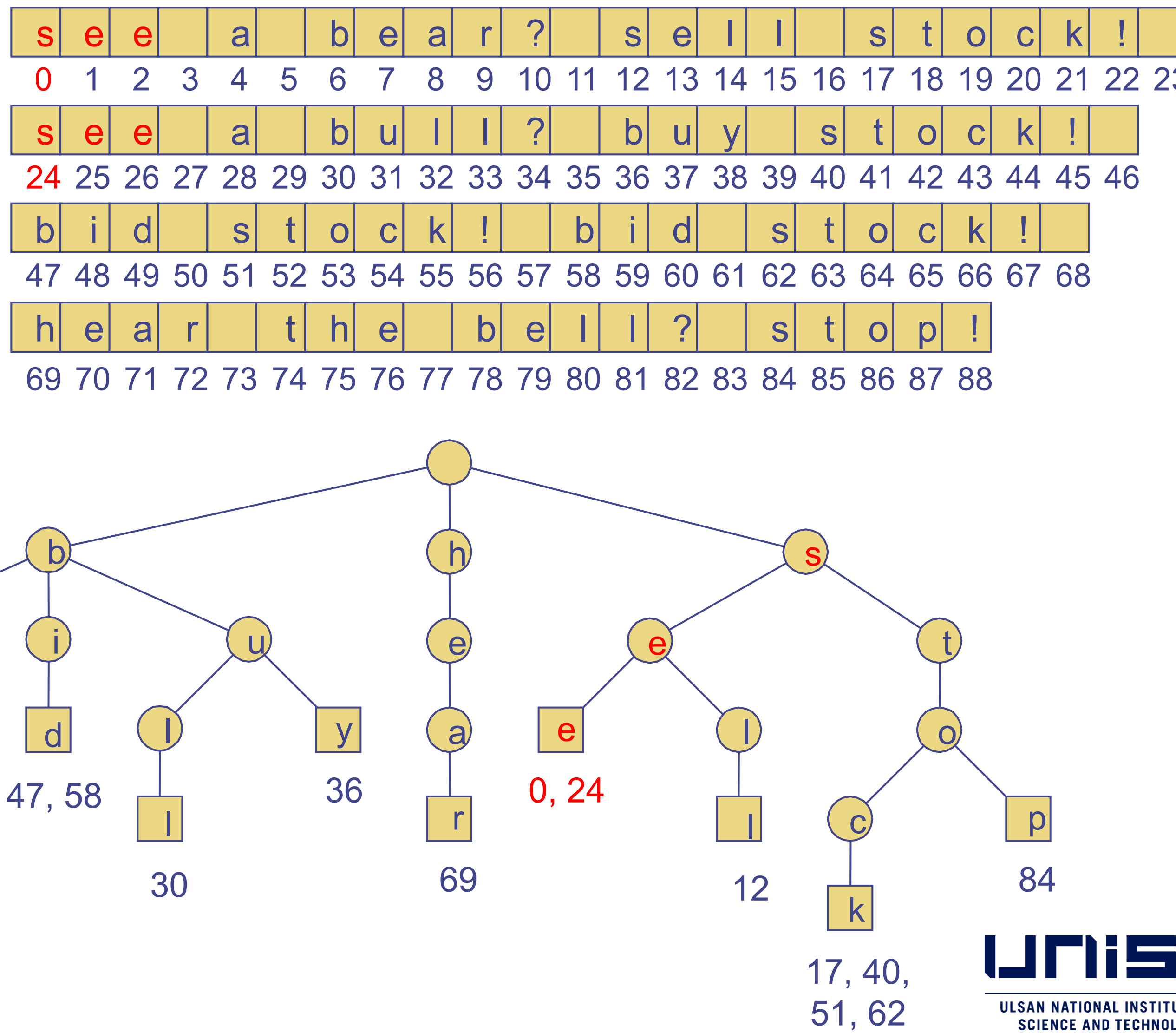
Analysis of Standard Tries

- 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

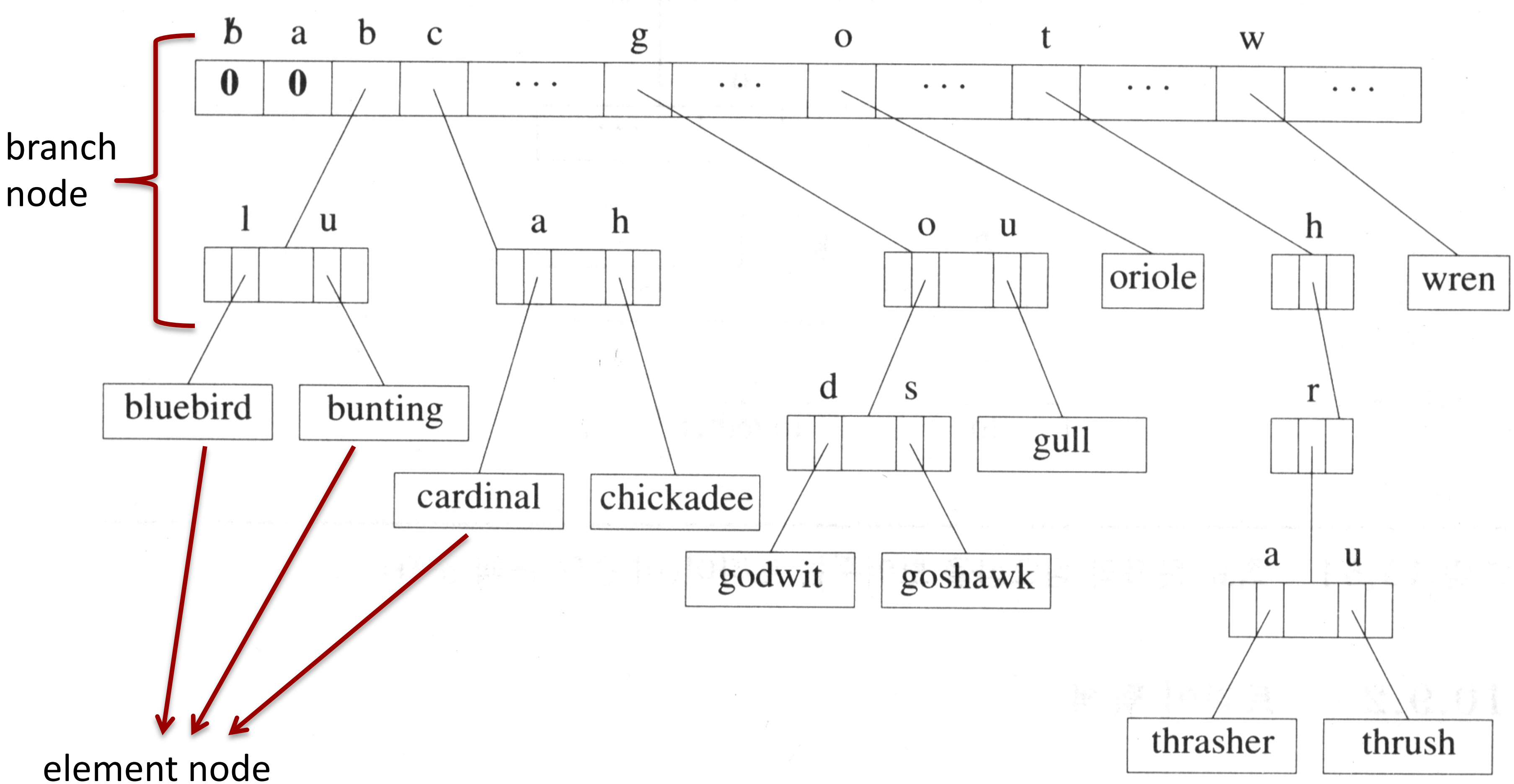


Word Matching with a Trie

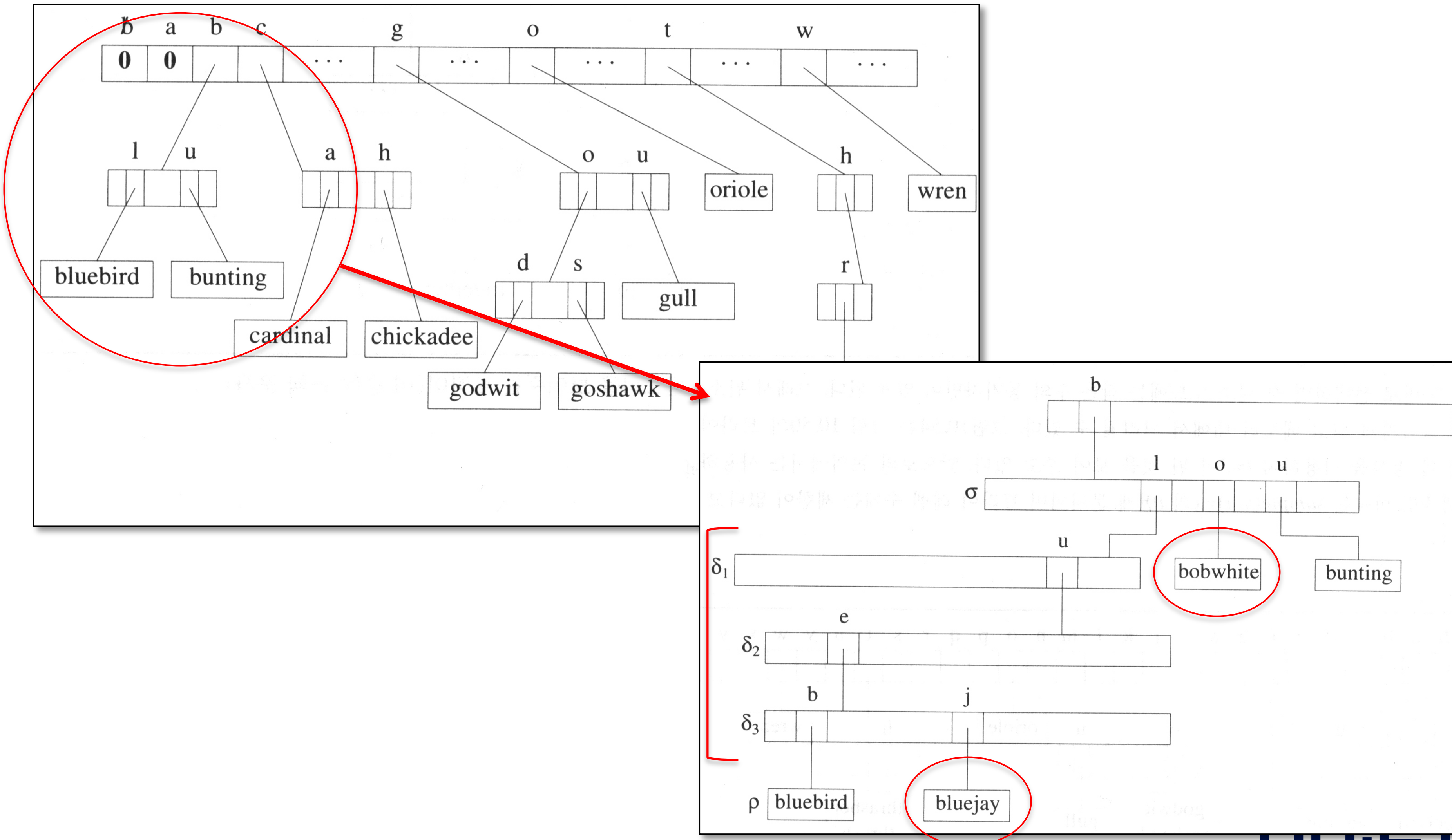
- Insert the words of the text into trie
- Each leaf is associated with one particular word
- Leaf stores indices where associated word begins ("see" starts at index 0 & 24, leaf for "see" stores those indices)



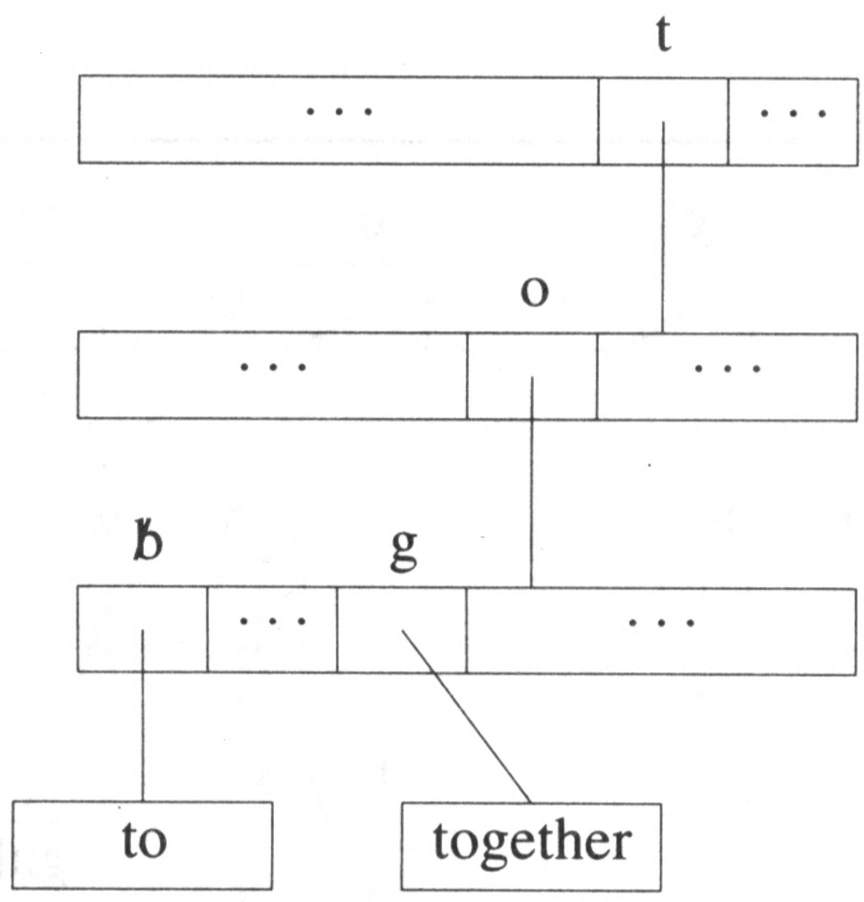
Multiway Trie Example



Insert / Delete



Terminal Character



Outline

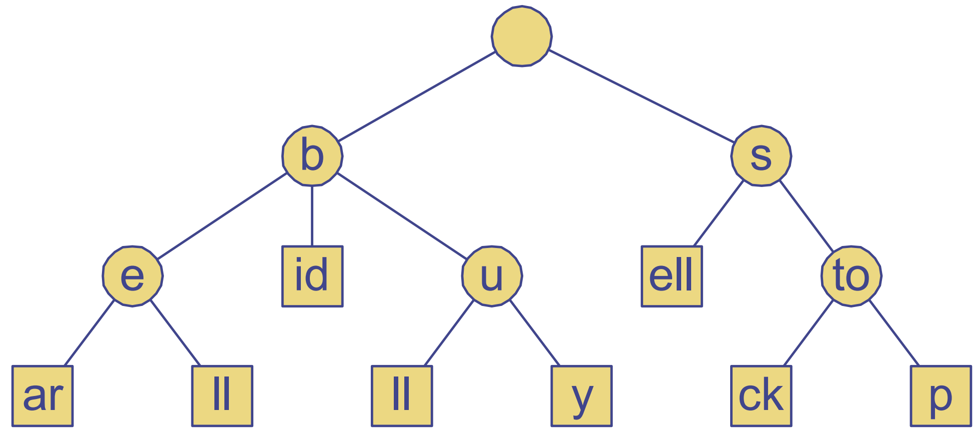
- Tries
- Compressed Tries
- Pattern matching algorithms
  - Brute force
  - Boyer-Moore
  - Knuth-Norris-Pratt

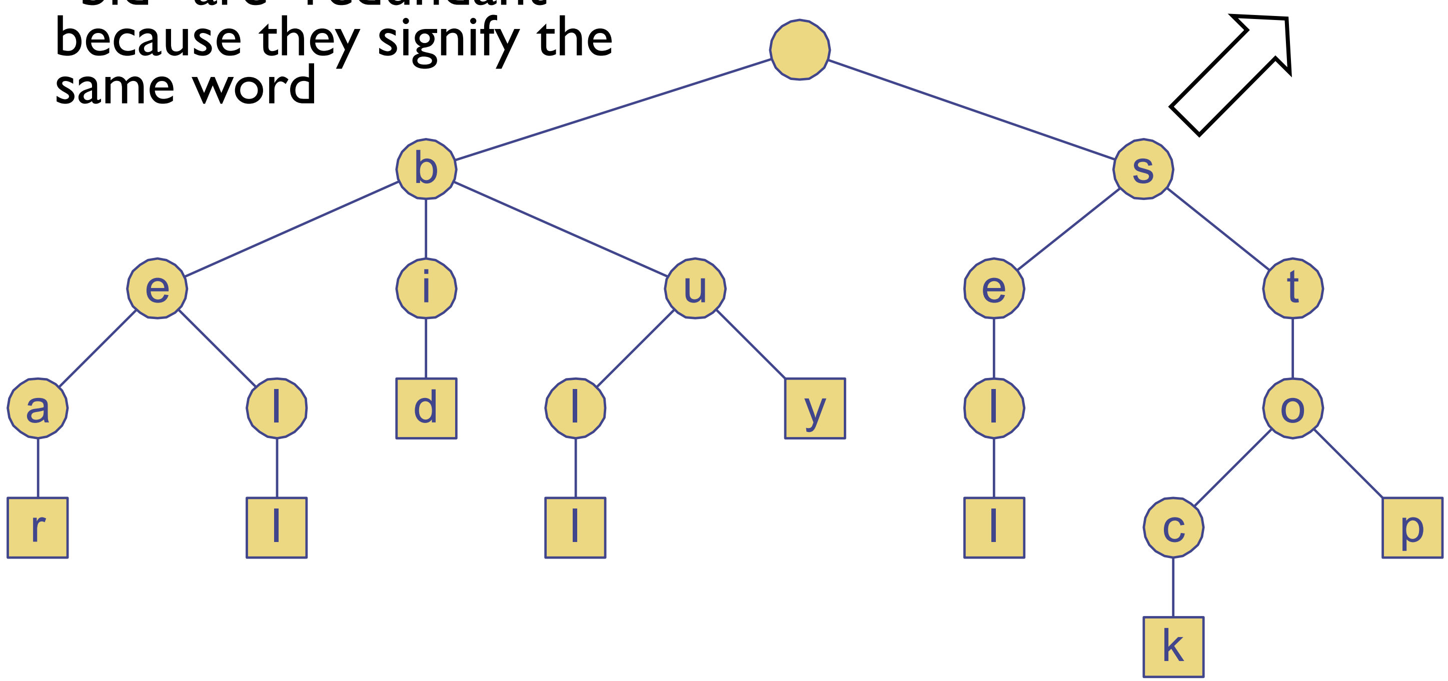
Compressed Tries

- Observation
  - Branch node v is redundant if v has one child and is not the root
- Approach 1:
  - A chain of redundant branch nodes can be represented with a single node
- Approach 2:
  - Use digit number

Approach 1: Compressed Tries

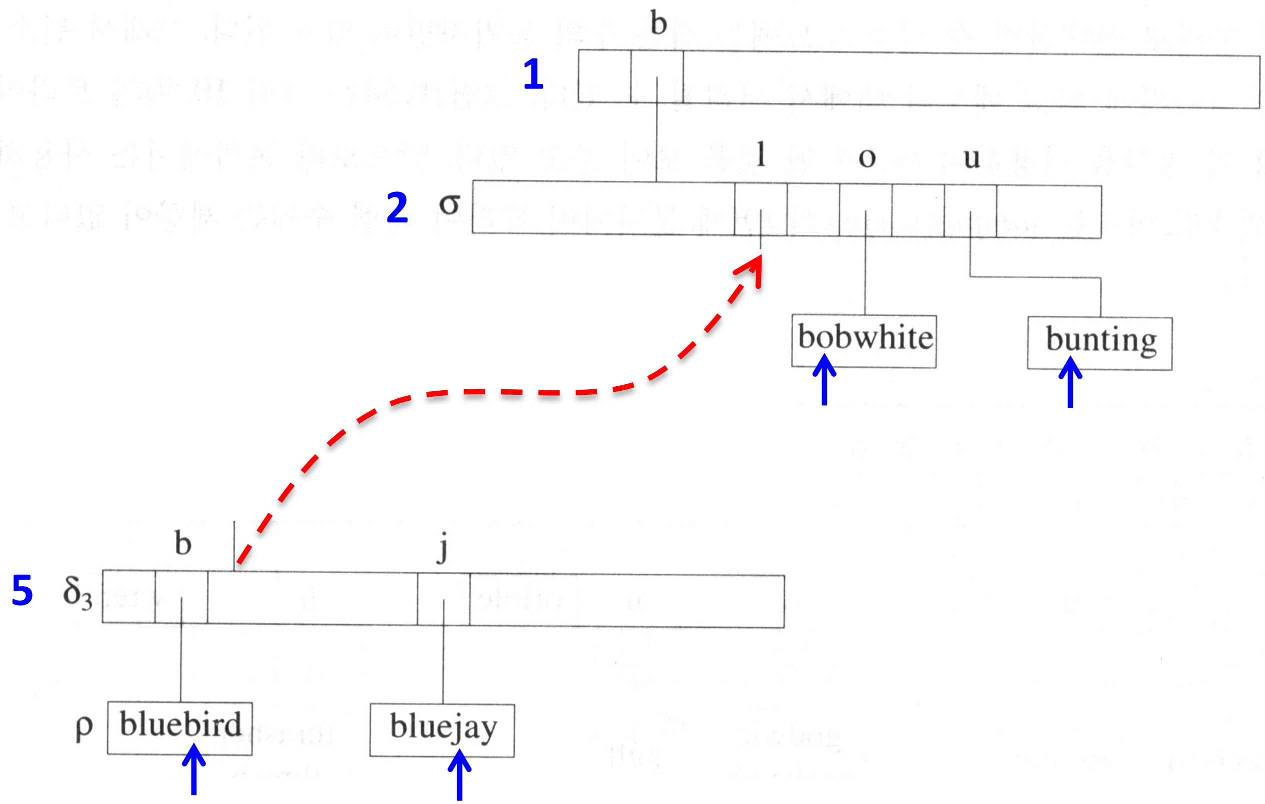
- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes
- ex. the “i” and “d” in “bid” are “redundant” because they signify the same word





Approach 2: Using Digit Numbers

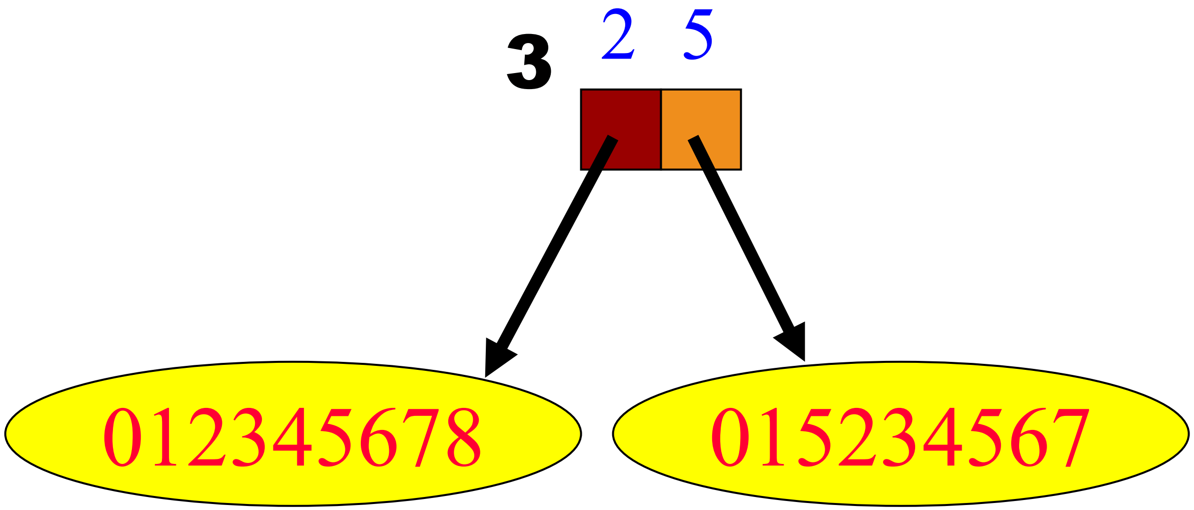
- Digit where branch is occur



Insert

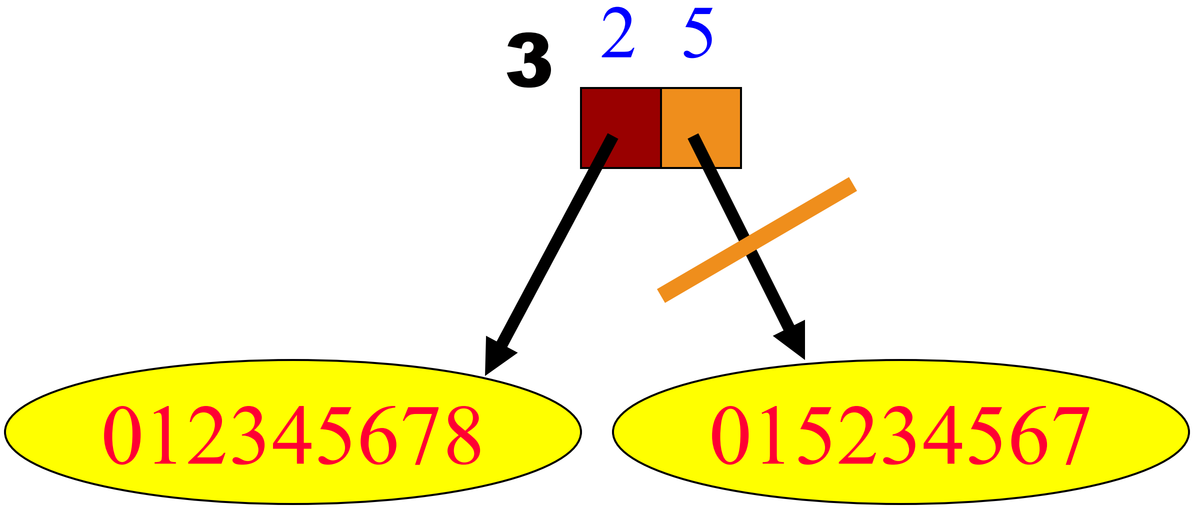
Insert 012345678.

Insert



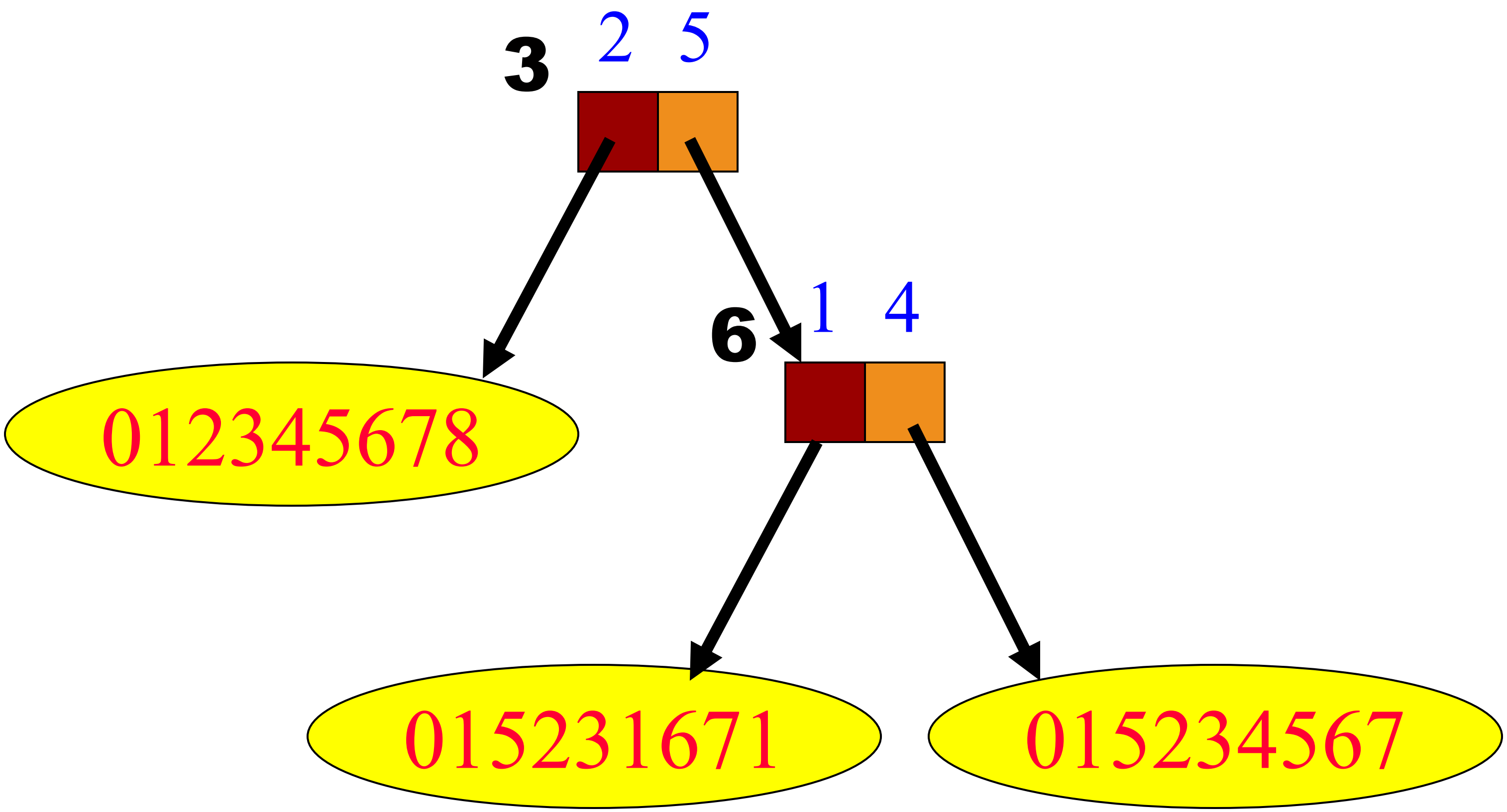
Insert 015234567.

Insert



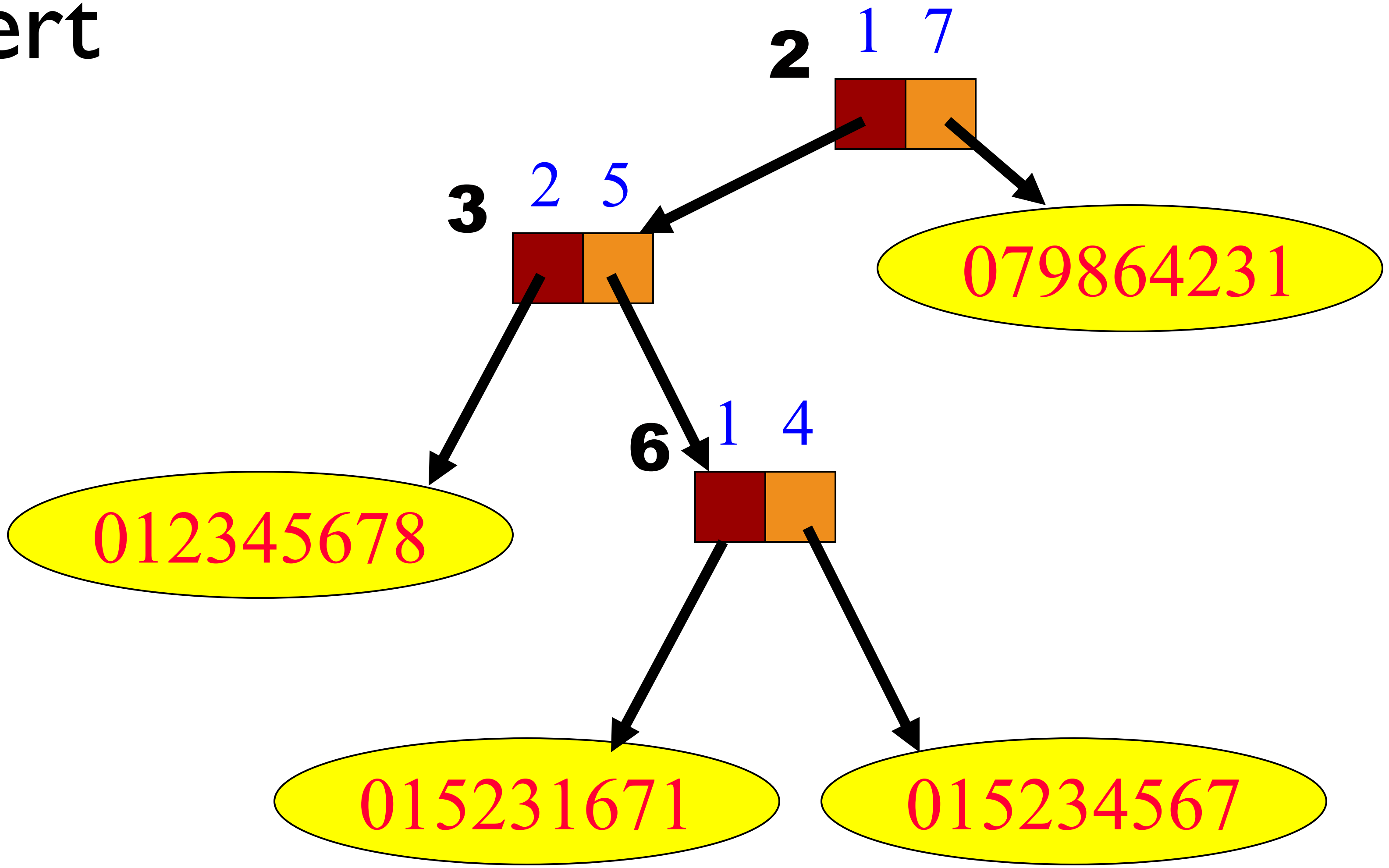
Insert 015231671.

Insert



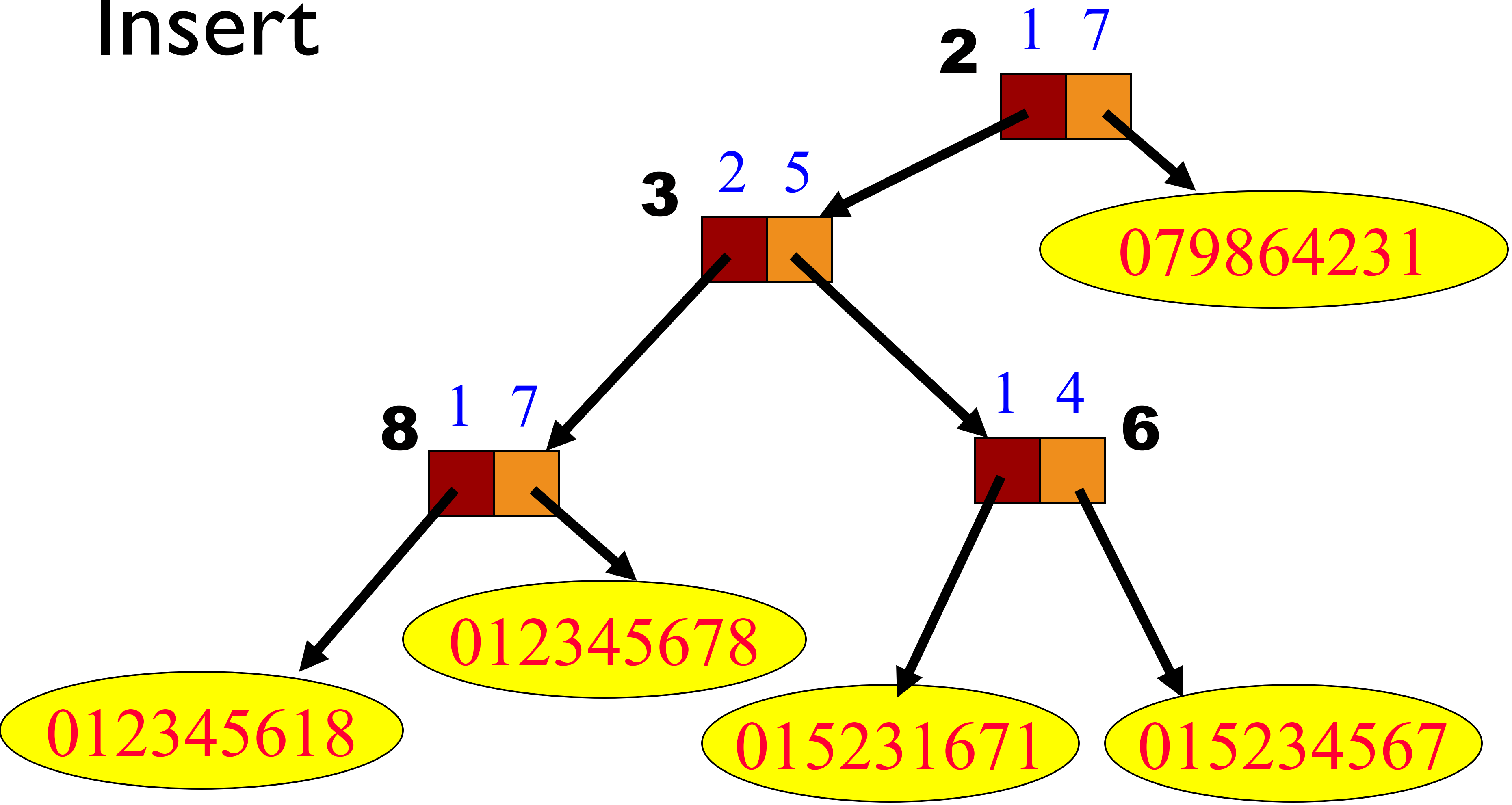
Insert 015231671.

Insert



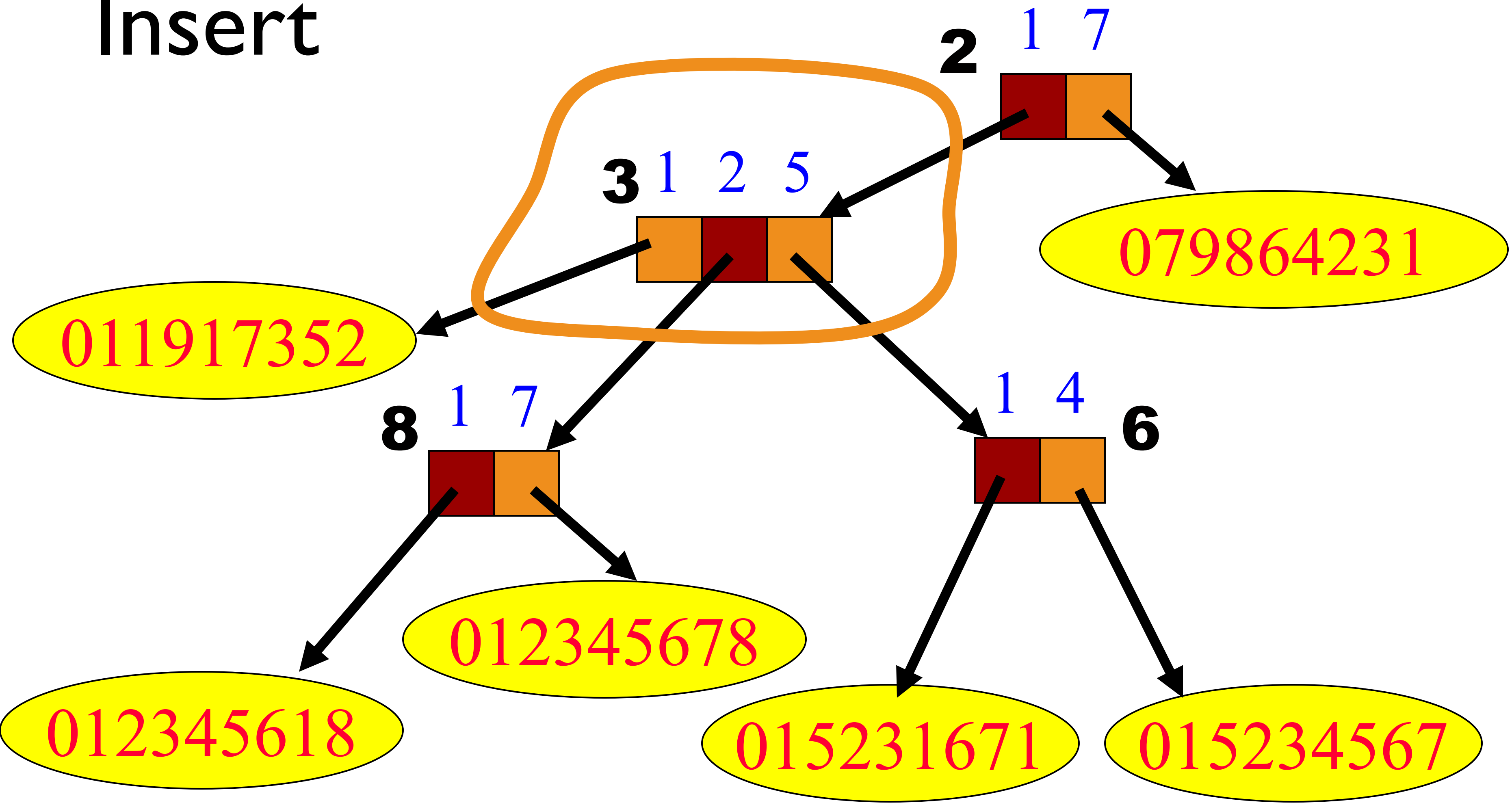
Insert 079864231.

Insert



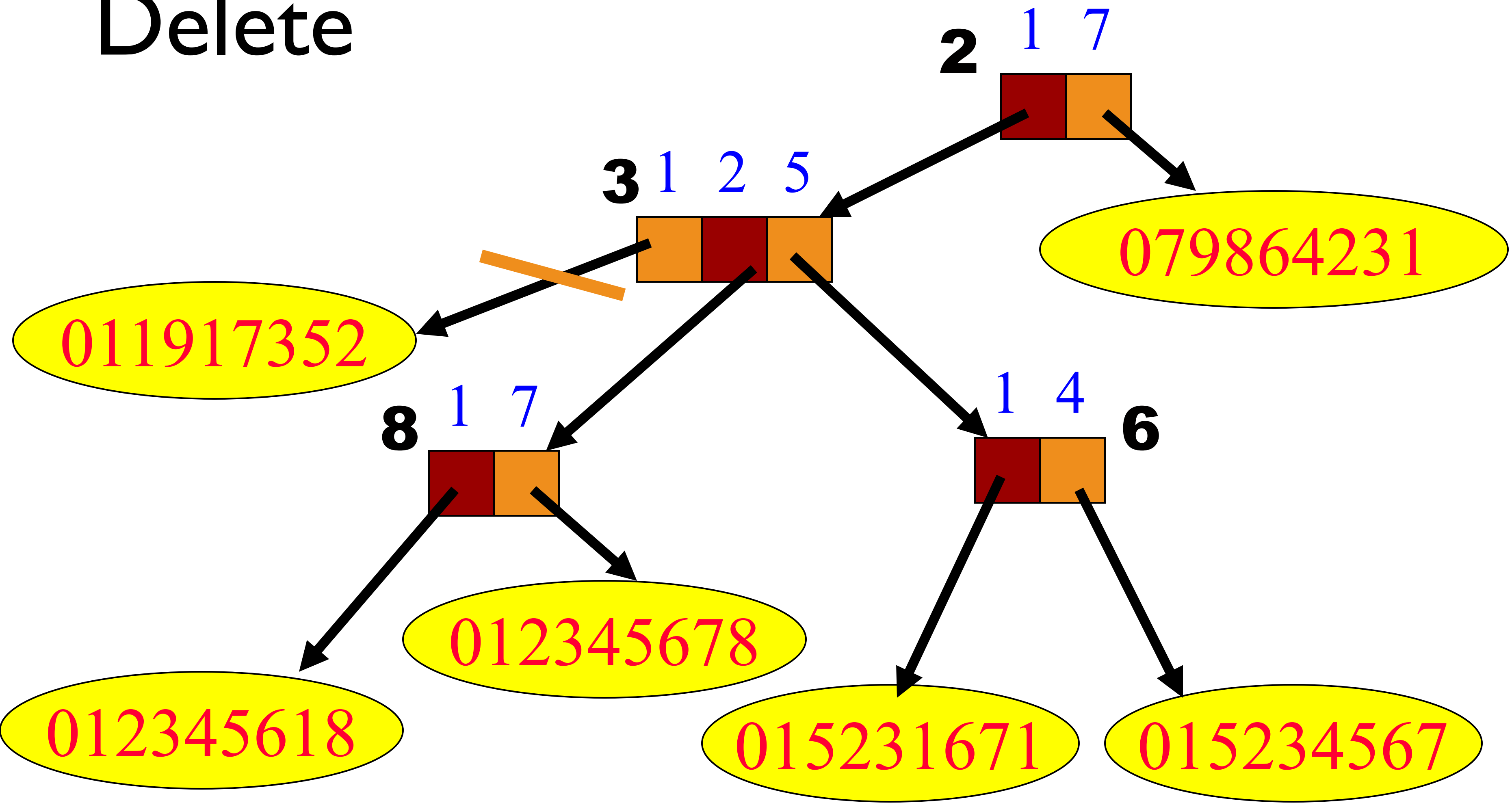
Insert 012345618.

Insert



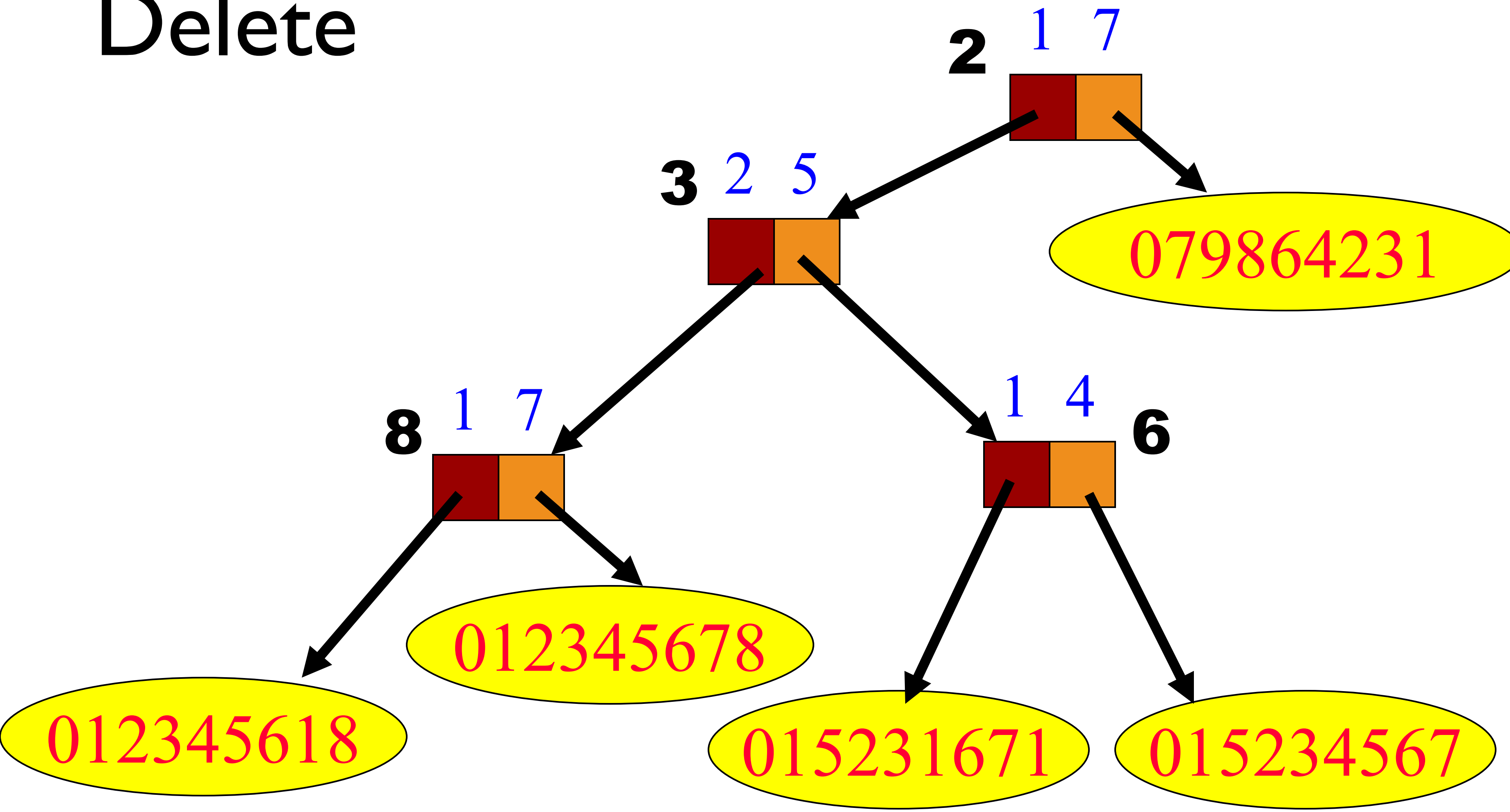
Insert 011917352.

Delete



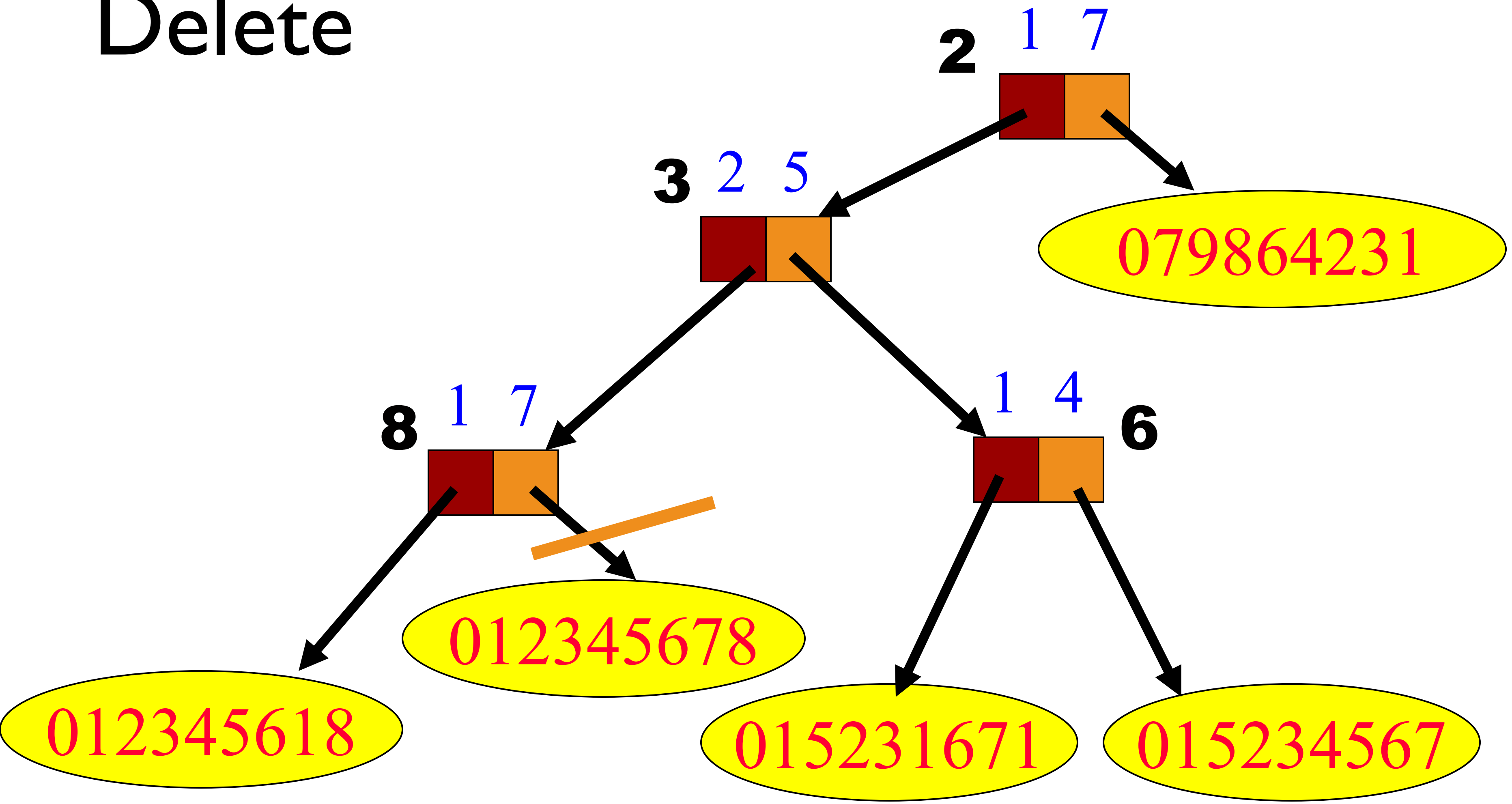
Delete 011917352.

Delete



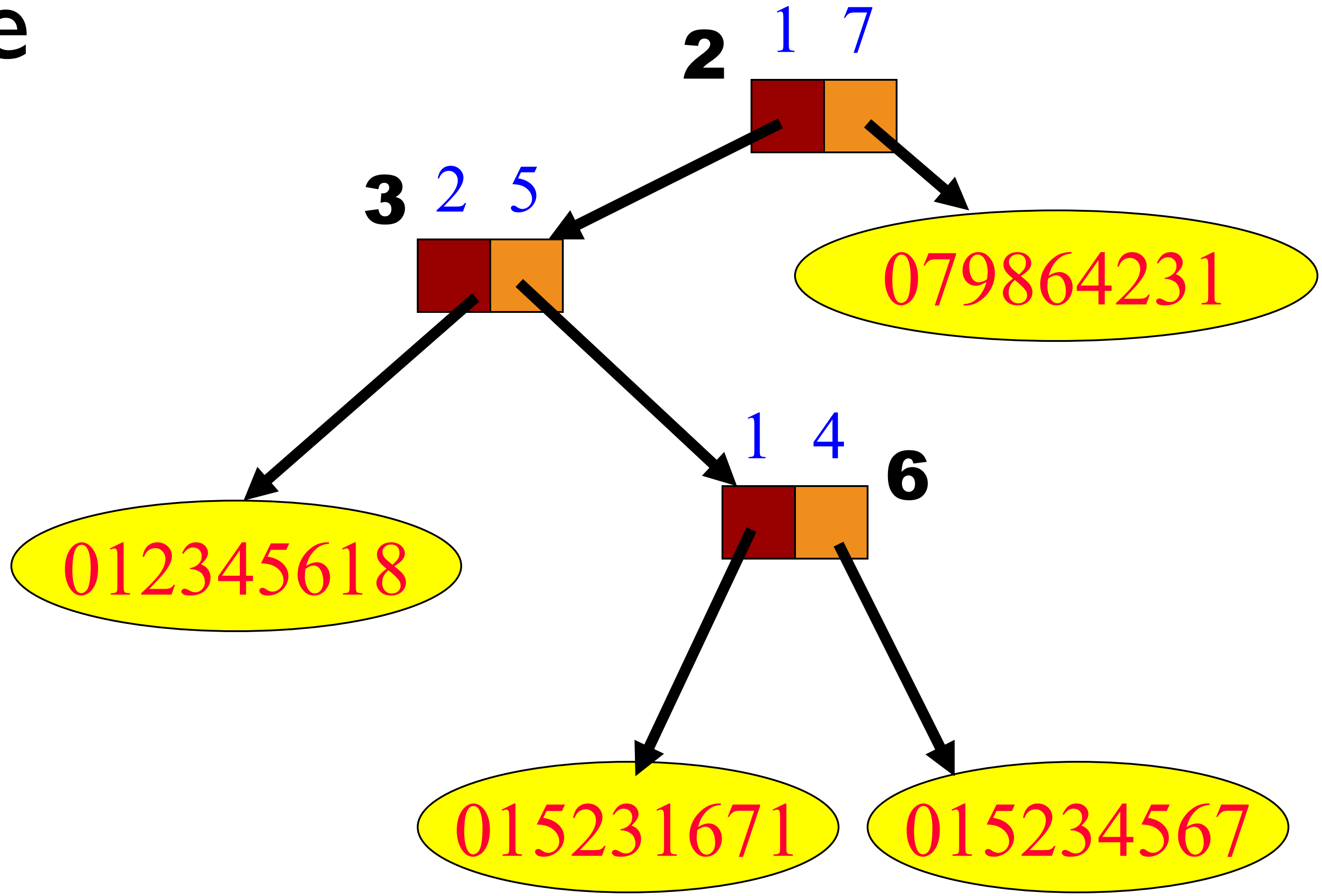
011917352 is deleted.

Delete



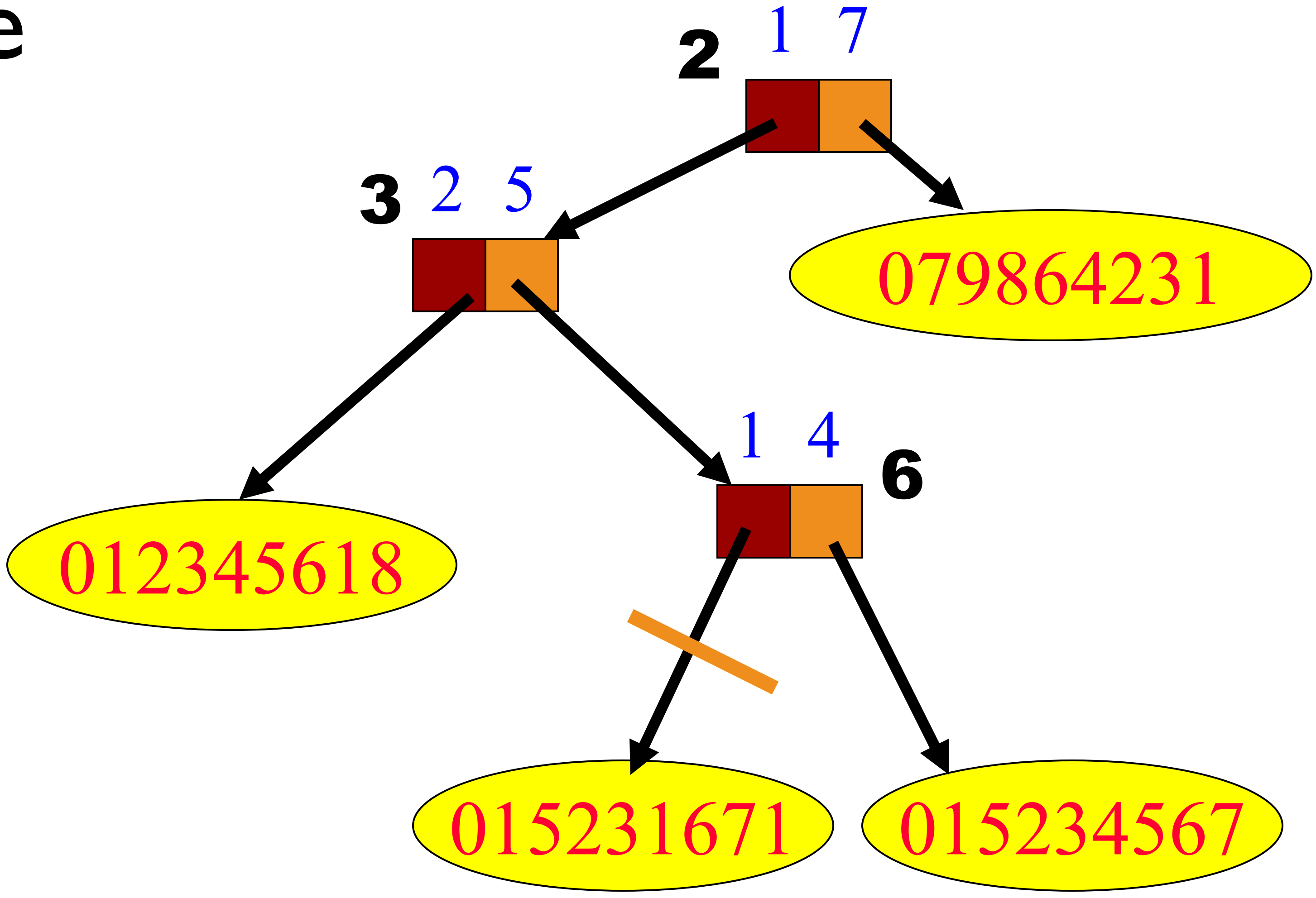
Delete 012345678.

Delete



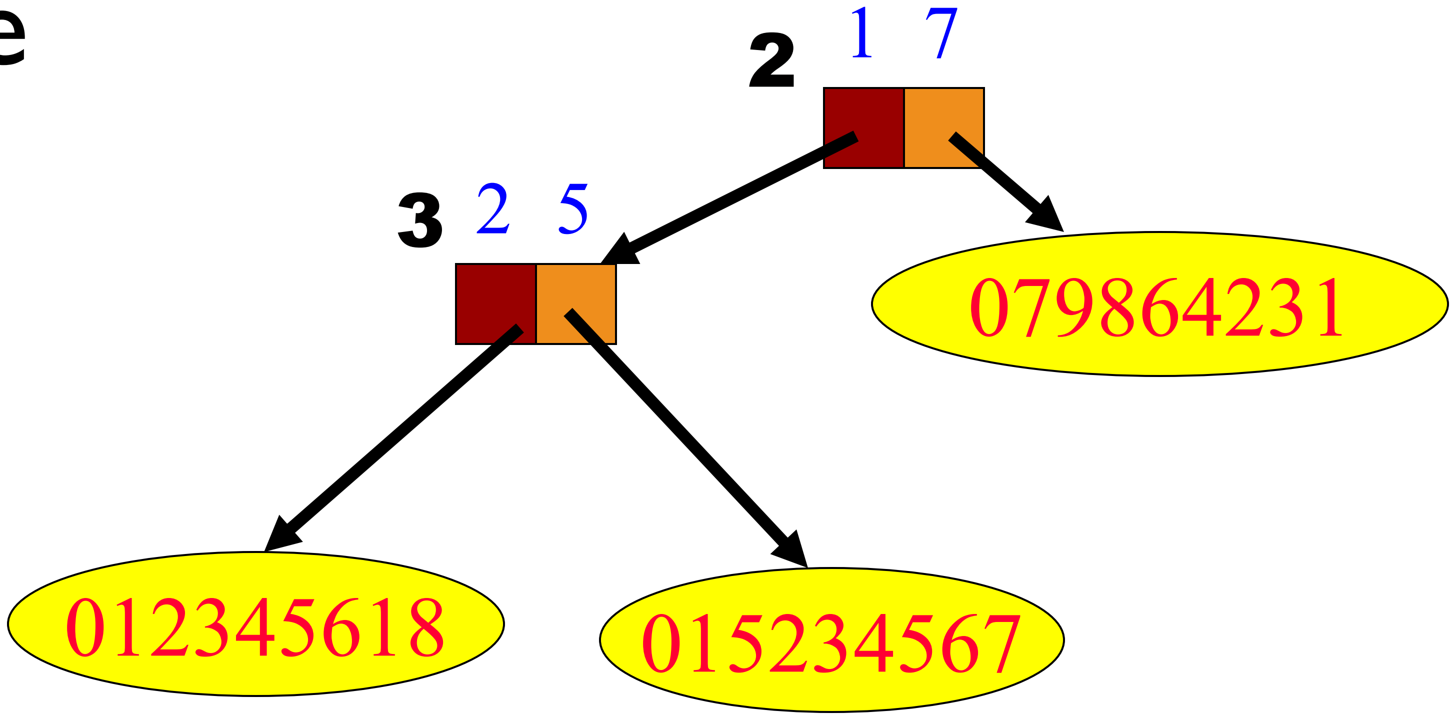
012345678 is deleted.

Delete



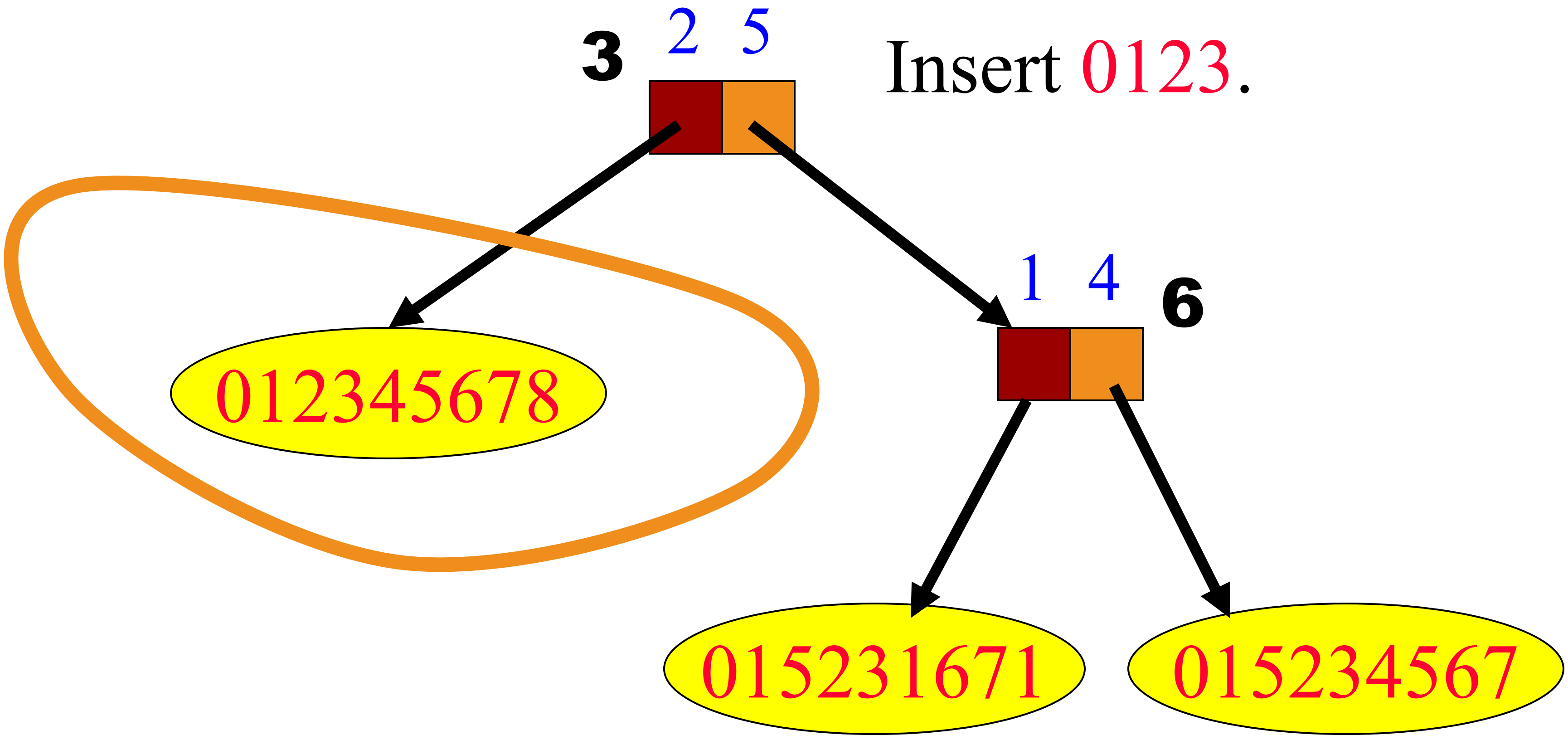
Delete 015231671.

Delete



015231671 is deleted.

Variable Length Keys

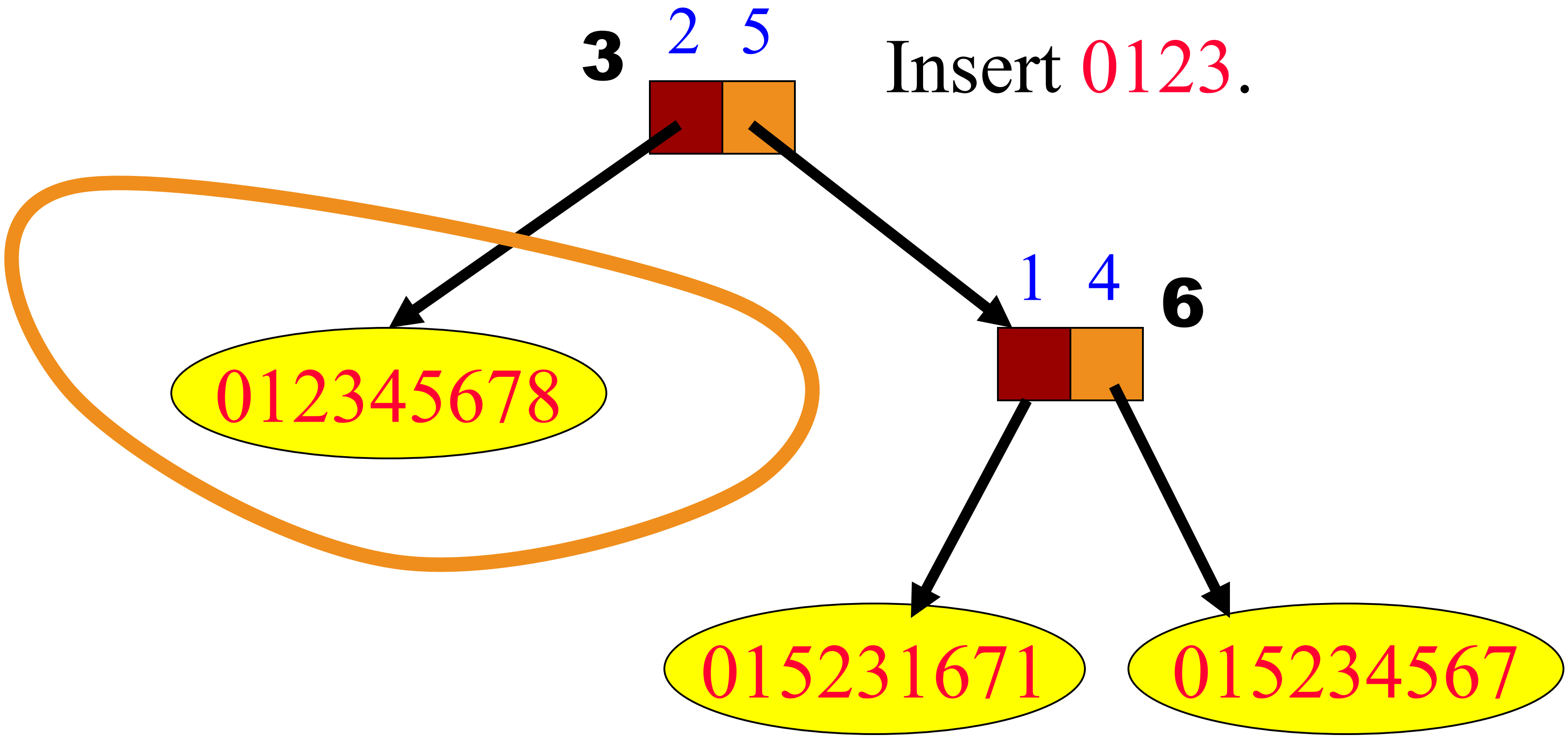


Insert 0123.

Problem arises only when one key is a (proper) prefix of another.

Variable Length Keys

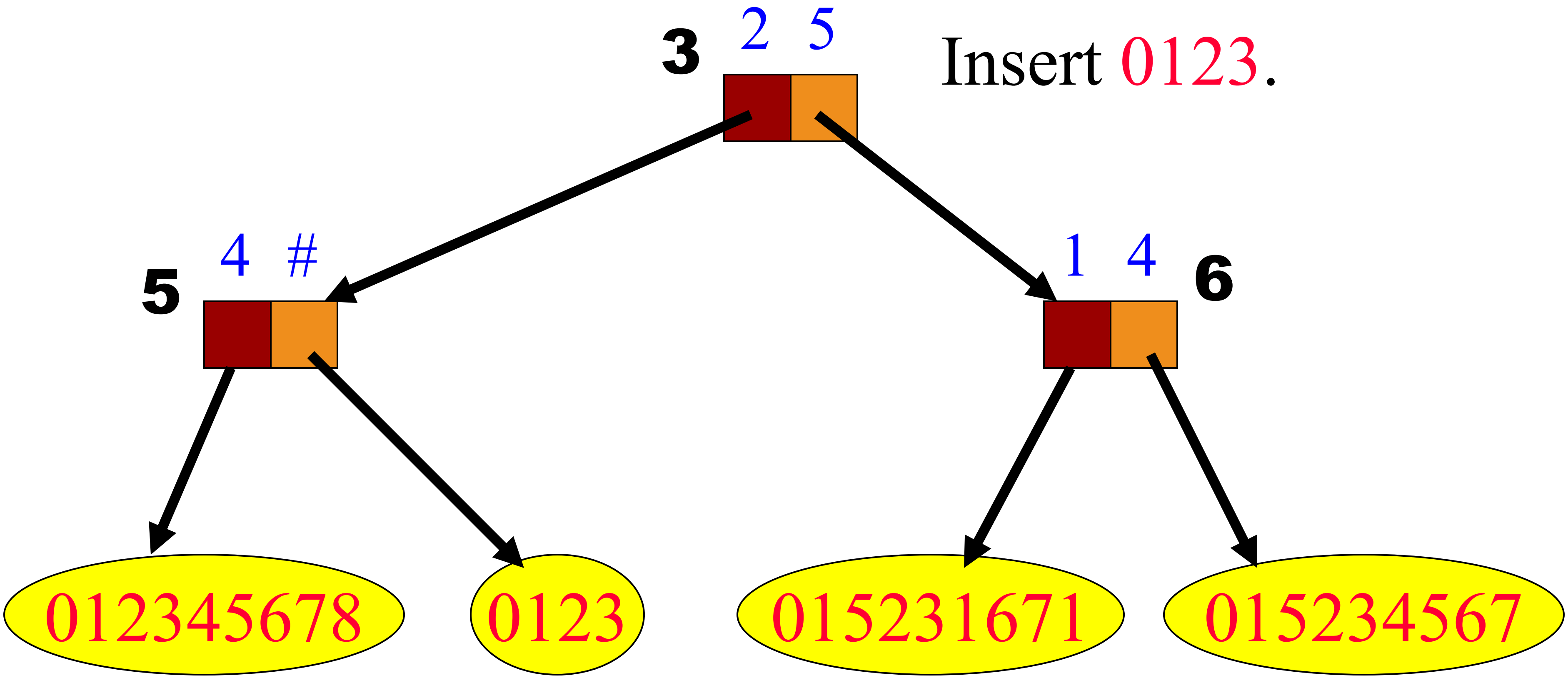
Insert 0123.



Add a special end of key character (#) to each key to eliminate this problem.

Variable Length Keys

Insert 0123.



End of key character (#) not shown.

Discussion

- Successful search terminates on leaf node
- Height depends on the key length
  - Search, insert, delete - O(s), s : max key length
  - Other search trees – O(log n), n : # of keys
  - Efficient for large number of records with small key size
- Insert / delete is easy
- Applications
  - Command completion, web browser, dictionary

Outline

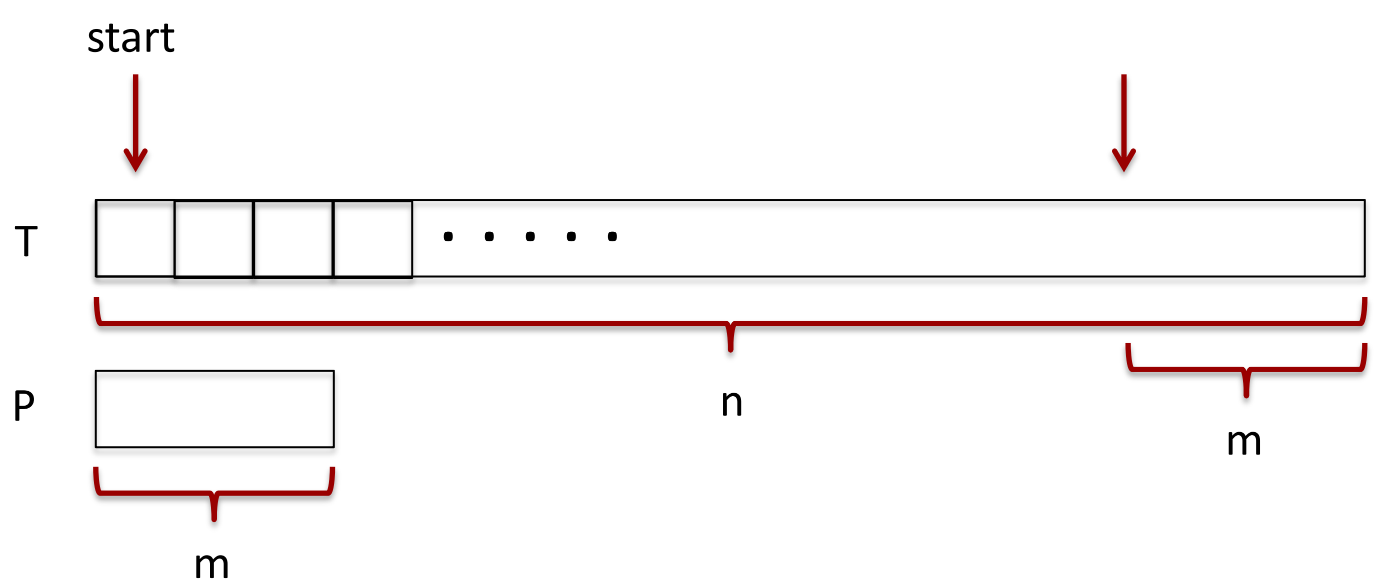
- Tries
- Compressed Tries
- Pattern matching algorithms
  - Brute force
  - Boyer-Moore
  - Knuth-Norris-Pratt

String ADT

- $S=s_0,s_1,\ldots,s_{n-1}$ where s_i are characters, n: length of character
- n=0: null (empty) character
- Operations
  - Comparing
  - Inserting
  - Removing
  - Finding a pattern

Simple String Pattern Matching

- Brute-force comparison
  - Worst case complexity : $(n-m)*m = O(n*m)$
    - T = aaaa……ah, P = aaah

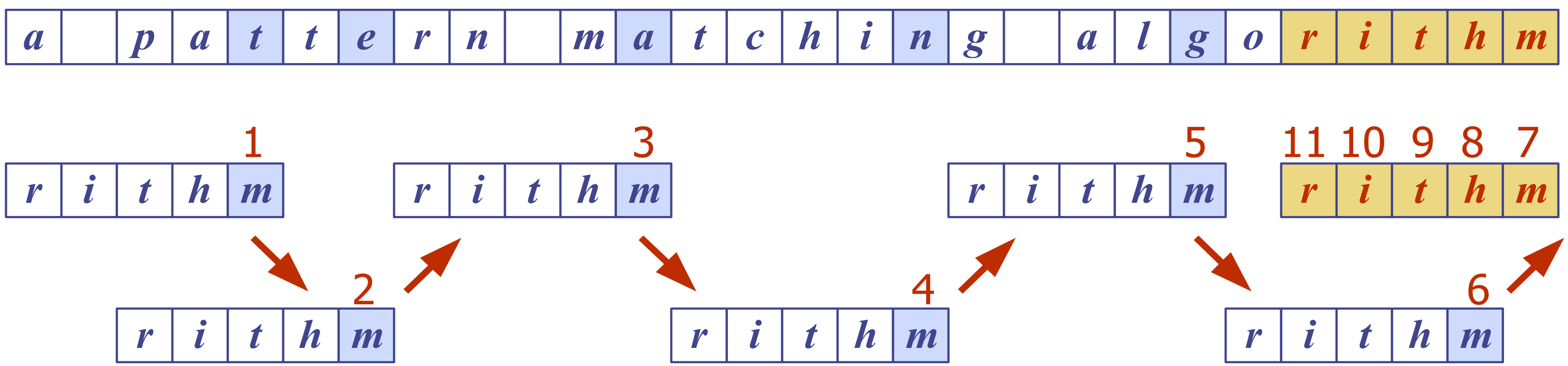


The Boyer-Moore Algorithm

- 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]

The Boyer-Moore Algorithm

- Example



Last-Occurrence Algorithm

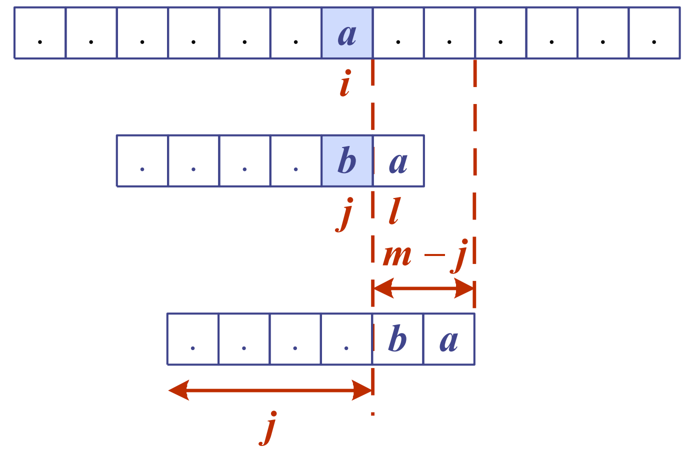
- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet S to build the last-occurrence function L mapping S to integers, where L(c) is defined as
  - the largest index i such that P[i] = c or
  - -1 if no such index exists
  - O(m+s)
- Example:
  - S = {a, b, c, d}
  - P = abacab

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

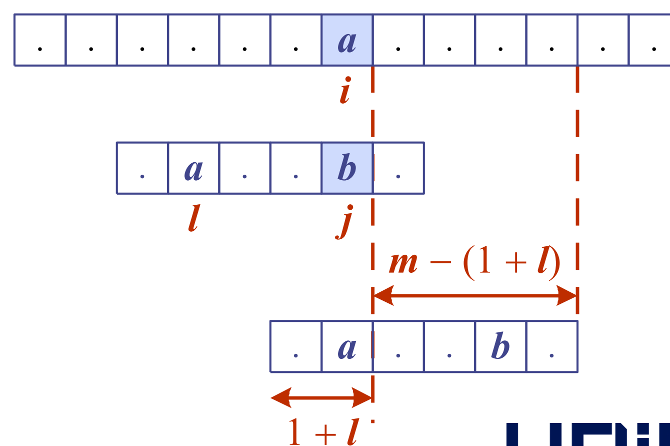
The Boyer-Moore Algorithm

```
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 }
```

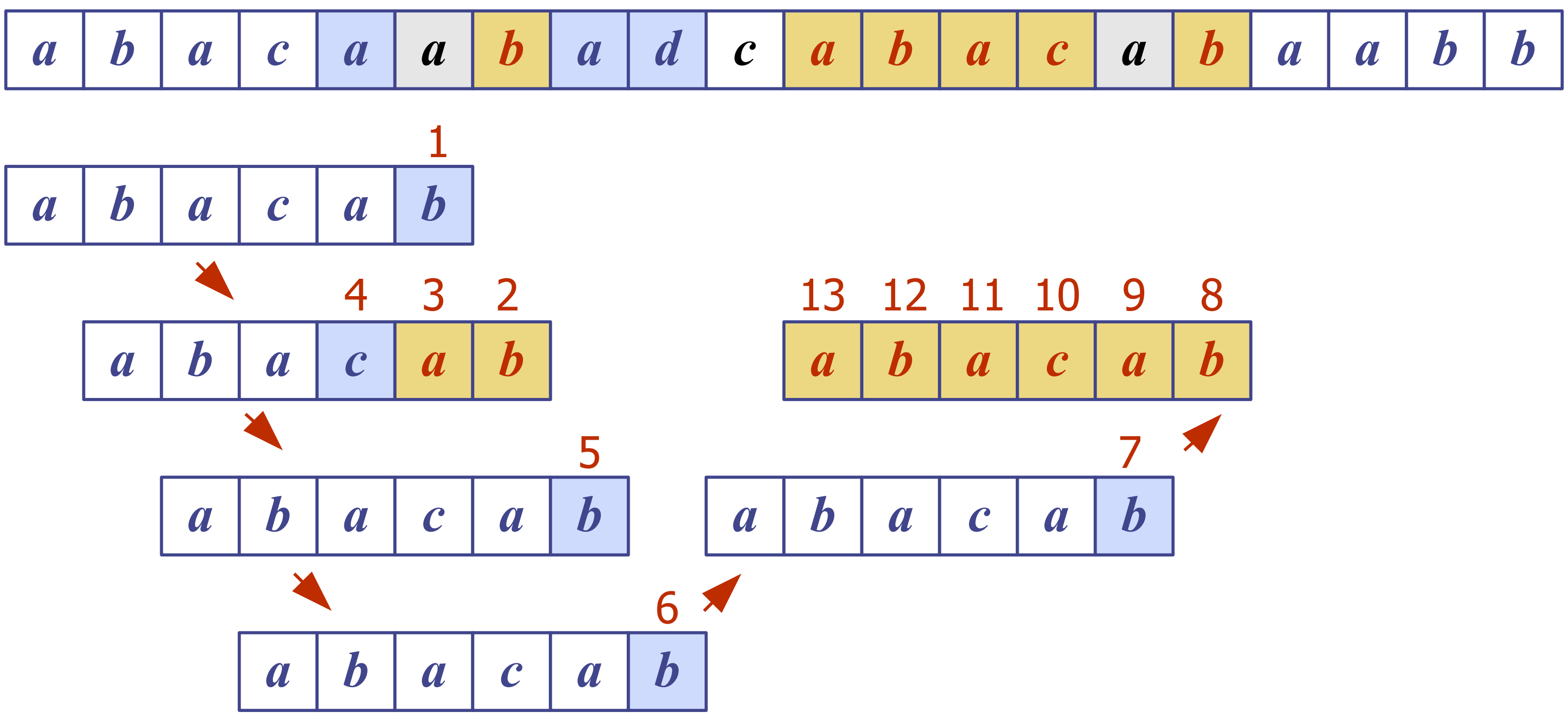
Case 1: $j < 1 + l$



Case 2: $1 + l \leq j$

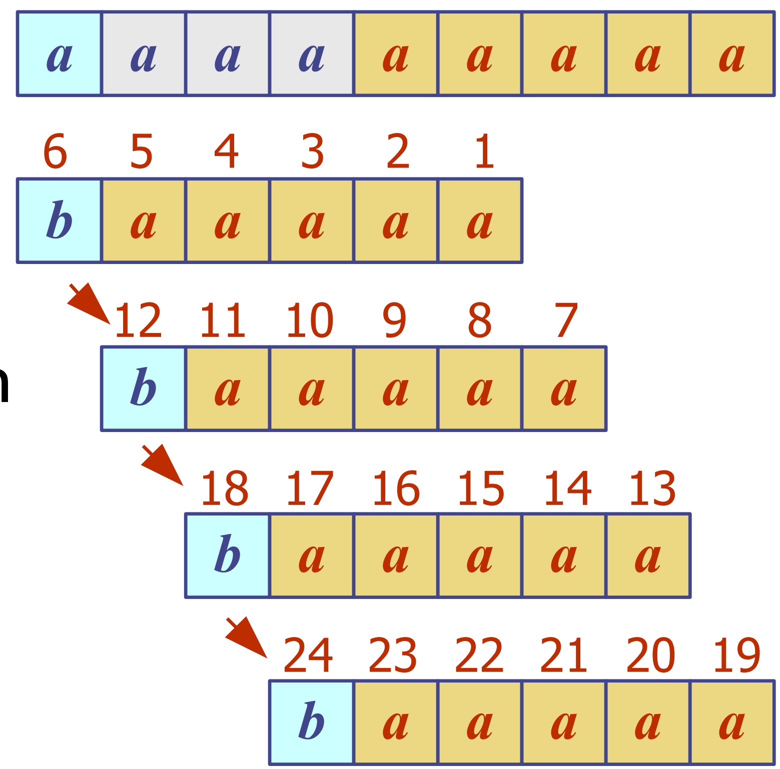


Example


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

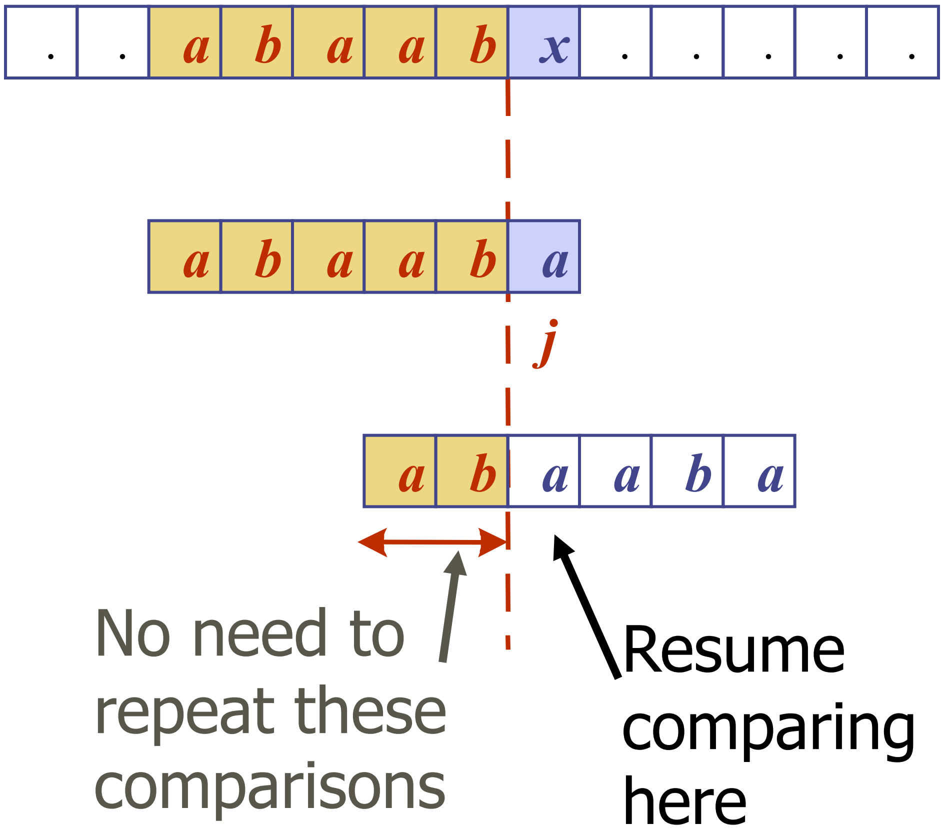
Analysis

- Boyer-Moore's algorithm runs in time O(nm + s)
- Example of worst case:
  - T = aaa … 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 (Knutt-Morris-Pratt) Algorithm

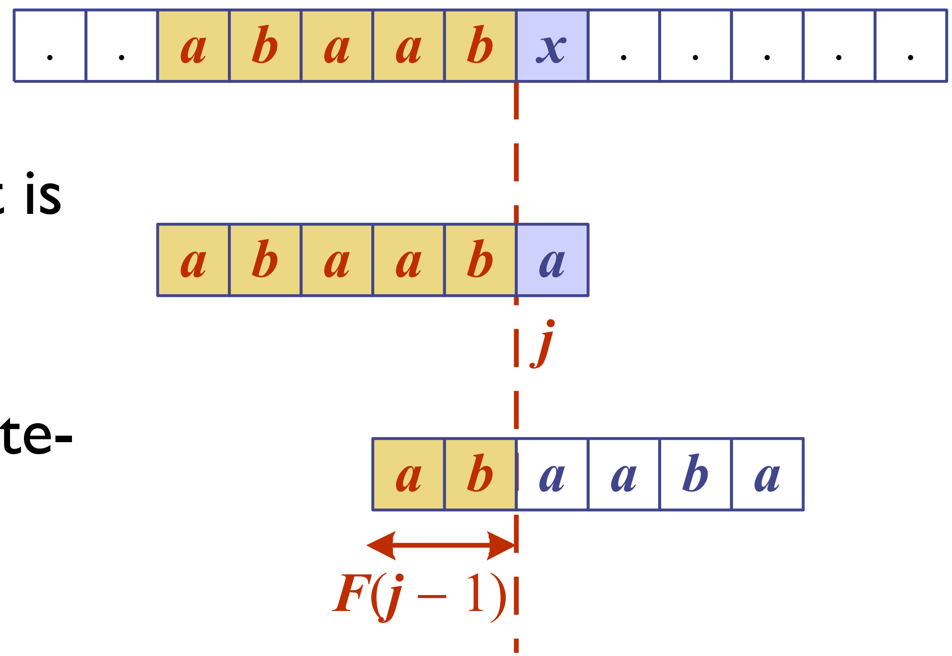
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of P[0..j]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



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] ≠ T[i] we set j = 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 = 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

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

Questions?