Module 8: Tries and String Matching

CS 240 - Data Structures and Data Management

Shahin Kamali, Yakov Nekrich, Olga Zorin

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

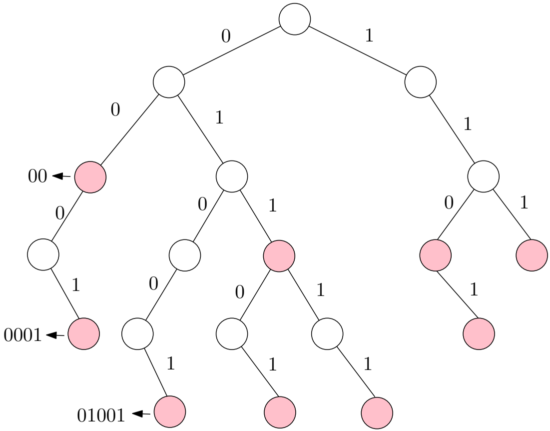
Spring 2015

Tries

- Trie (Radix Tree): A dictionary for binary strings
  - Comes from retrieval, but pronounced "try"
  - A binary tree based on bitwise comparisons
  - Similar to radix sort: use individual bits, not the whole key
- Structure of trie:
  - A left child corresponds to a 0 bit
  - A right child corresponds to a 1 bit
- Keys can have different number of bits
- Keys are not stored in the trie: a node x is flagged if the path from root to x is a binary string present in the dictionary

Tries

- Example: A trie for
  $S = \{00, 0001, 01001, 011, 01101, 01111, 110, 1101, 111\}$



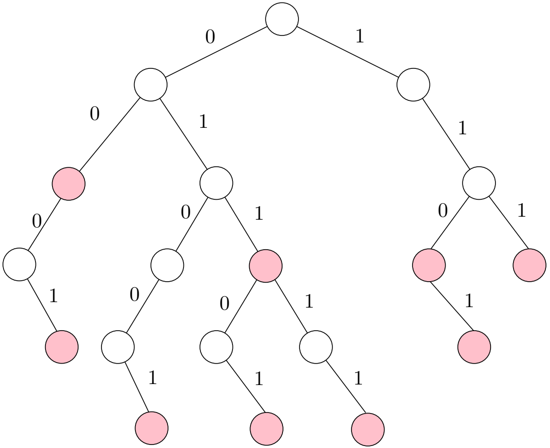
Tries: Search

Search(x):

- start from the root
- take the left link if the current bit in x is 0 and take the right link if it is 1 (return failure if the link is missing)
- if there are no extra bits in x left and the current node is flagged then - success (x is found)
- else, if the current node is a leaf, then - failure (x is missing)
- recurse

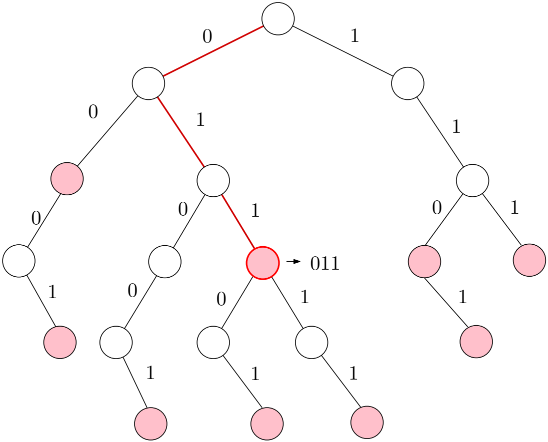
Tries: Search

Example: Search(011)



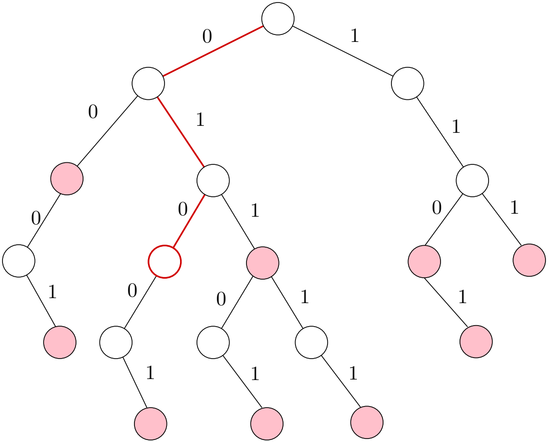
Tries: Search

Example: Search(011) successful



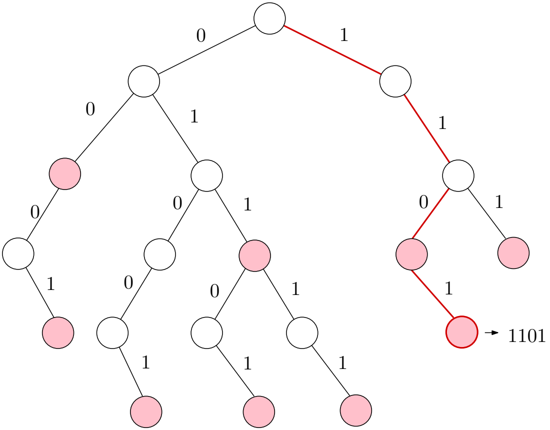
Tries: Search

Example: Search(0101) unsuccessful



Tries: Search

Example: Search(1101) successful

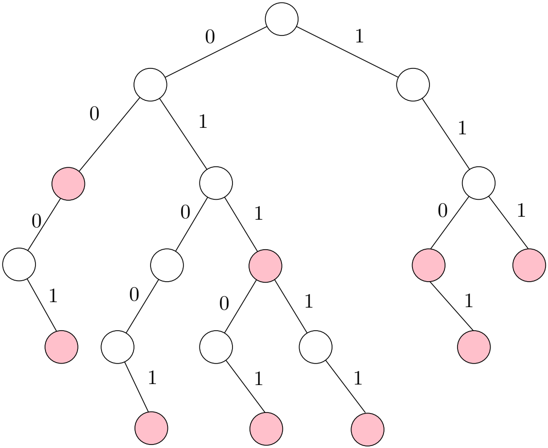


Tries: Insert

- Insert(x)
  - First search for x
  - If we finish at a leaf with key x, then x is already in trie: do nothing
  - If we finish at a leaf v and x has extra bits then flag v and expand the trie from the node v by adding necessary nodes that correspond to extra bits.
  - If we finish at an internal node and there are no extra bits: the node is then flagged
  - If we finish at an internal node and there are extra bits: expand trie by adding necessary nodes that correspond to extra bits

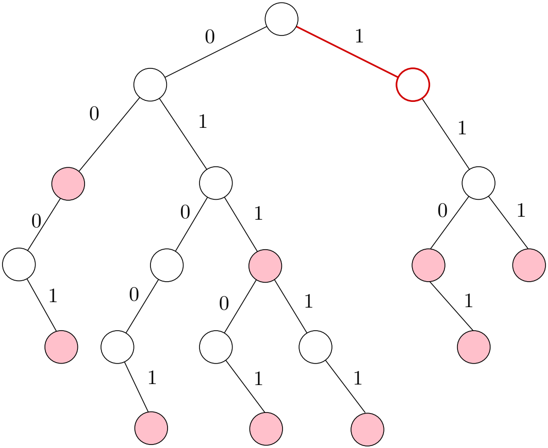
Tries: Insert

Example: Insert(101)



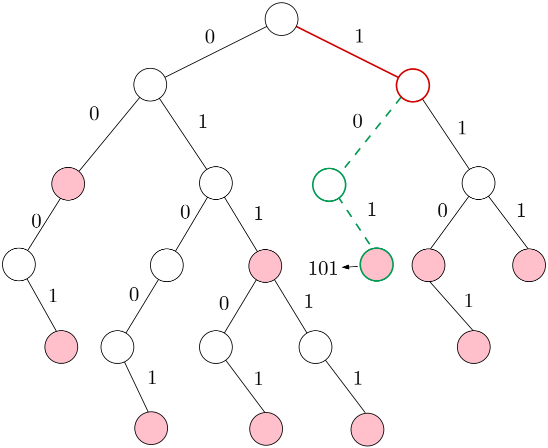
Tries: Insert

Example: Insert(101)



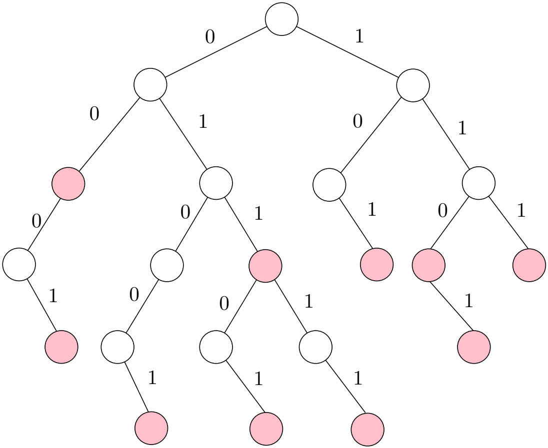
Tries: Insert

Example: Insert(101)



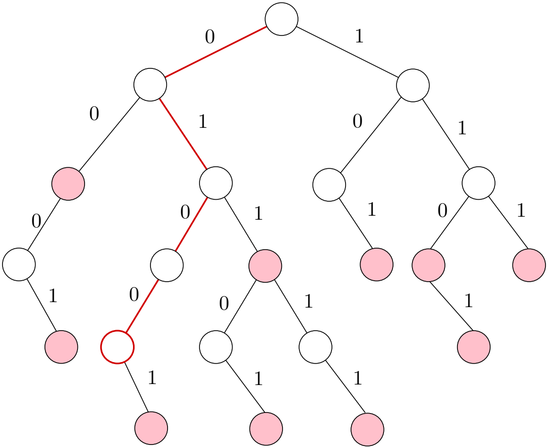
Tries: Insert

Example: Insert(0100)



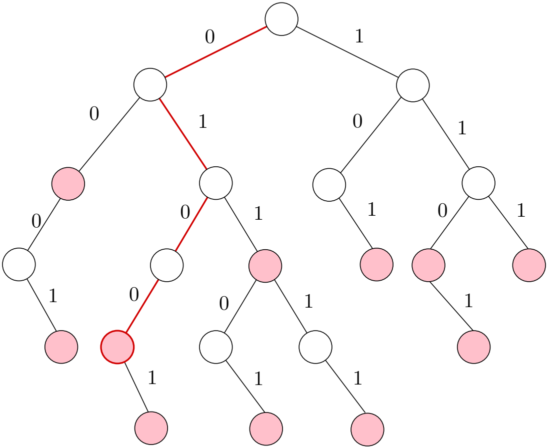
Tries: Insert

Example: Insert(0100)



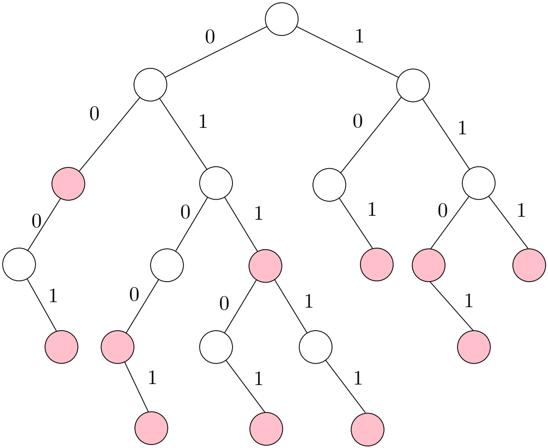
Tries: Insert

Example: Insert(0100)



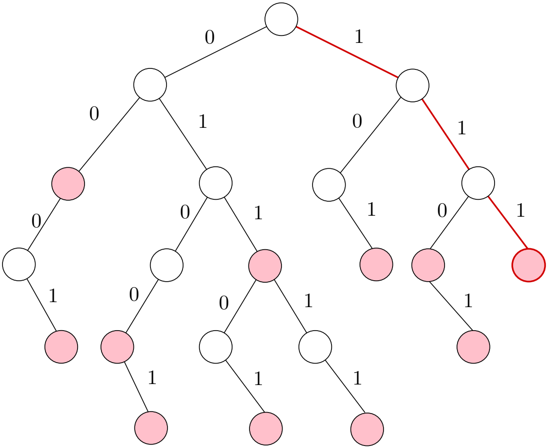
Tries: Insert

Example: Insert(11101)



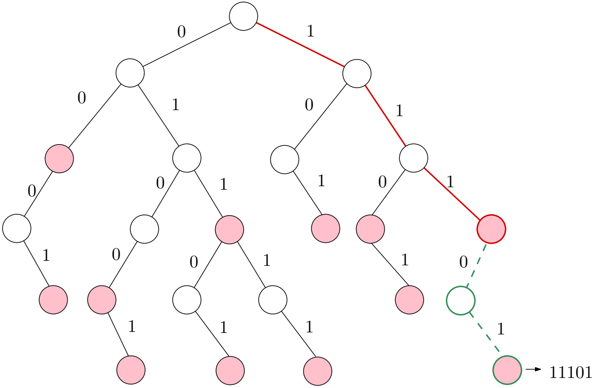
Tries: Insert

Example: Insert(11101)



Tries: Insert

Example: Insert(11101)

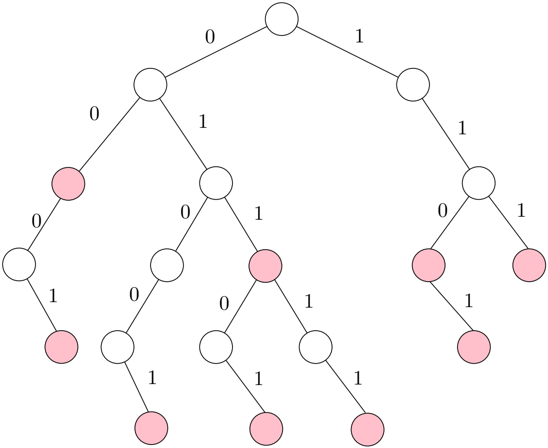


Tries: Delete

- Delete(x)
  - Search for x
  - if x found at an internal flagged node, then unflag the node
  - if x found at a leaf v_x, delete the leaf and all ancestors of v_x until
    - we reach an ancestor that has two children or
    - we reach a flagged node

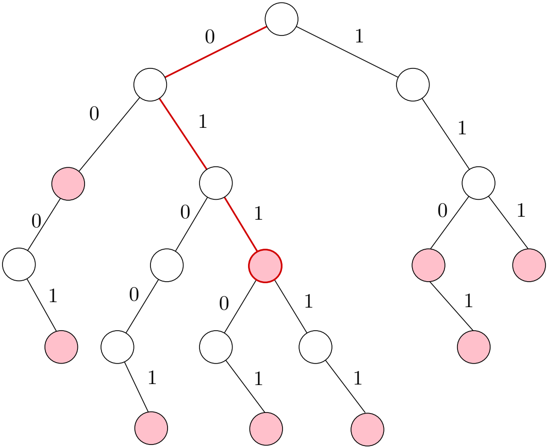
Tries: Delete

Example: Delete(011)



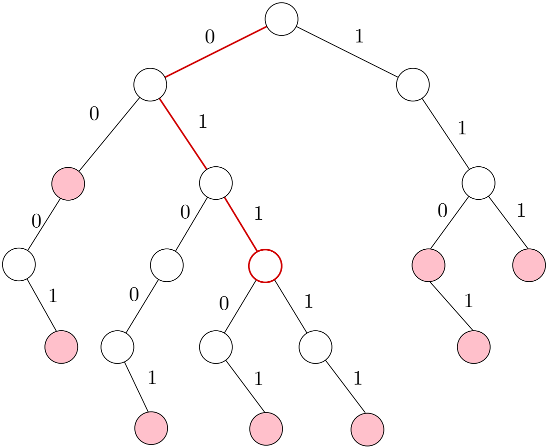
Tries: Delete

Example: Delete(011)



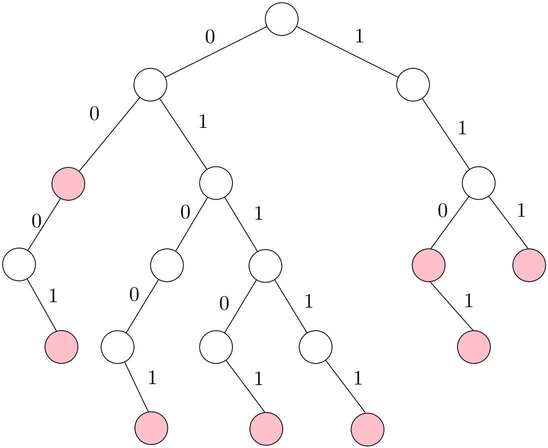
Tries: Delete

Example: Delete(011)



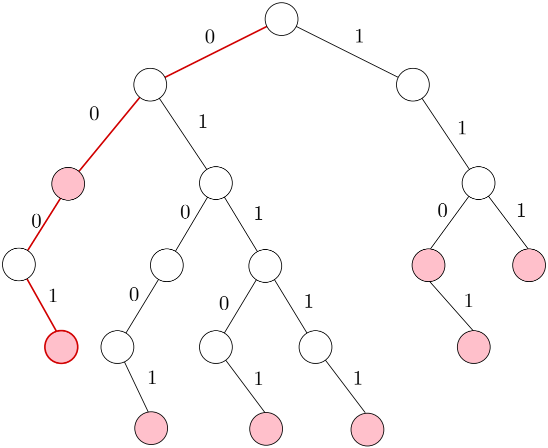
Tries: Delete

Example: Delete(0001)



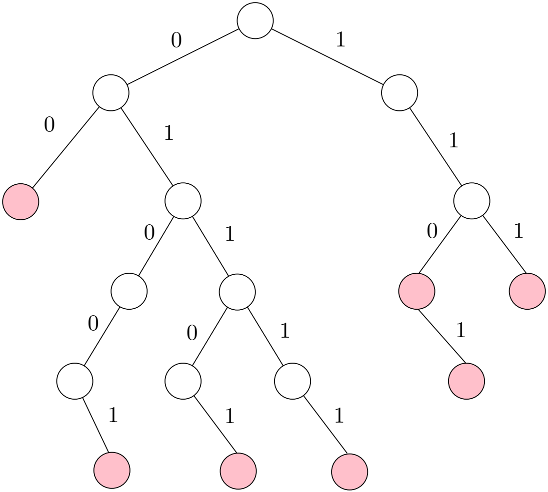
Tries: Delete

Example: Delete(0001)



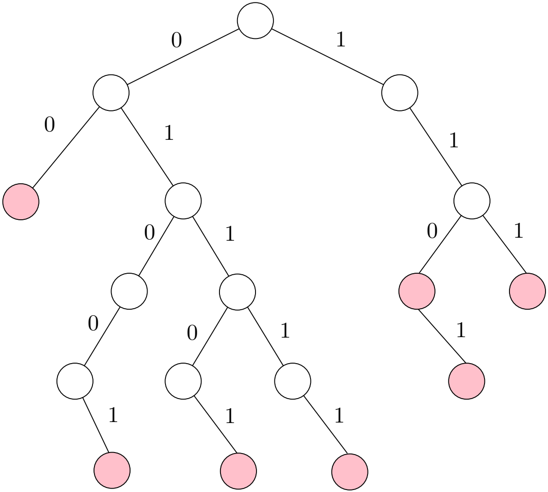
Tries: Delete

Example: Delete(0001)



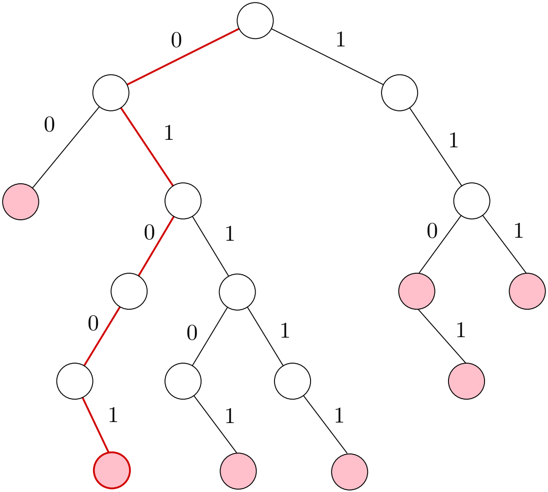
Tries: Delete

Example: Delete(01001)



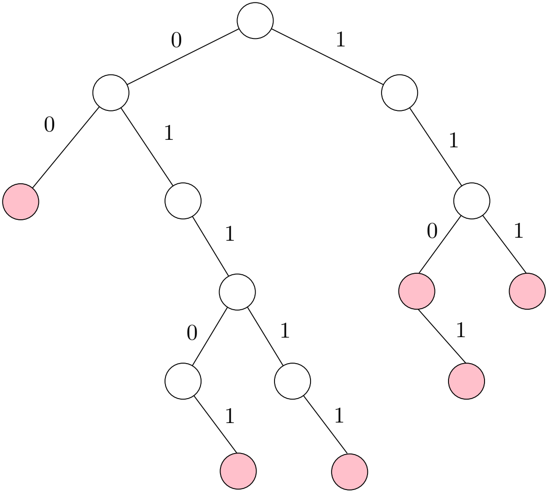
Tries: Delete

Example: Delete(01001)



Tries: Delete

Example: Delete(01001)



Tries: Operations

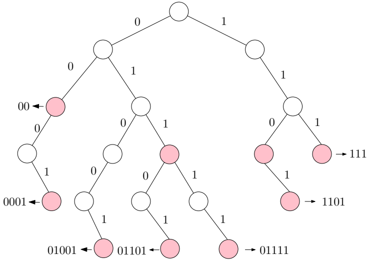
- Search(x)
- Insert(x)
- Delete(x)
- Time Complexity of all operations: $\Theta(|x|)$
  $|x|$: length of binary string x, i.e., the number of bits in x

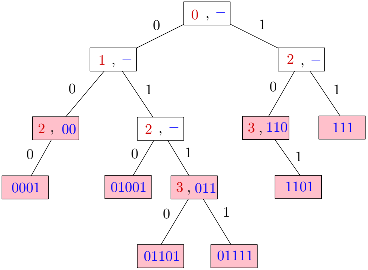
Compressed Tries (Patricia Tries)

- Patricia: Practical Algorithm To Retrieve Information Coded in Alphanumeric
- Introduced by Morrison (1968)
- Reduces storage requirement: eliminate unflagged nodes with only one child
- Every path of one-child unflagged nodes is compressed to a single edge
- Each node stores an index indicating the next bit to be tested during a search (index$= 0$ for the first bit, index$= 2$ for the second bit, etc)
- A compressed trie storing n keys always has at most $n - 1$ internal (non-leaf) nodes

Compressed Tries (Patricia Tries)

- Each node stores an index indicating the next bit to be tested during a search
- Example: A trie and the equivalent compressed trie



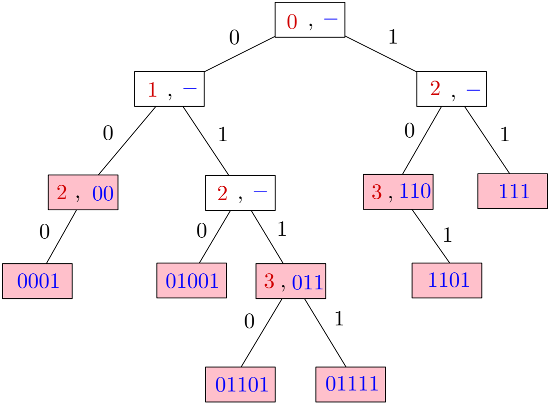


Compressed Tries: Operations

- Search(x):
  - Follow the proper path from the root down in the tree to a leaf
  - If search ends in an internal flagged node, it is successful
  - If search ends in an internal unflagged node, it is unsuccessful
  - If search ends in a leaf, we need to check again if the key stored at the leaf is indeed x

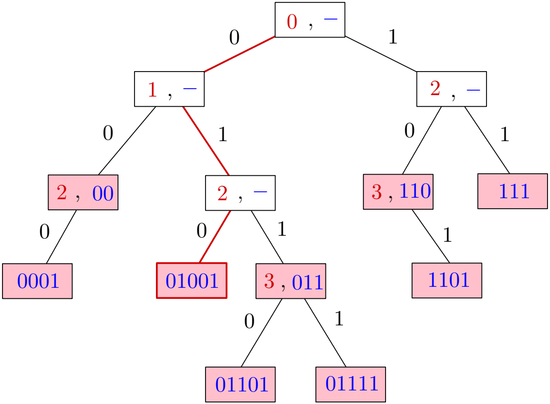
Compressed Tries: Operations

Example: Search(01001)



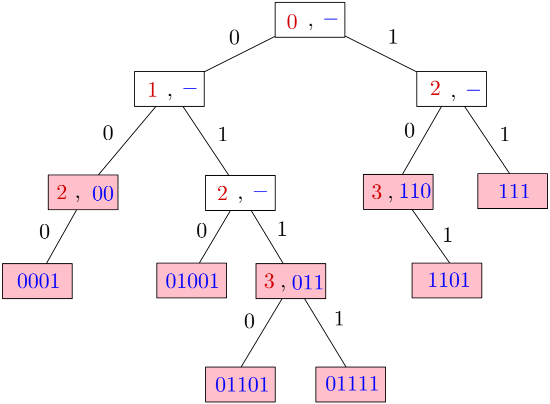
Compressed Tries: Operations

Example: Search(01001) - successful



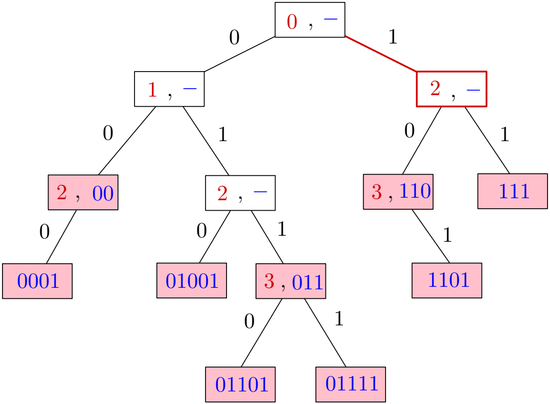
Compressed Tries: Operations

Example: Search(11)



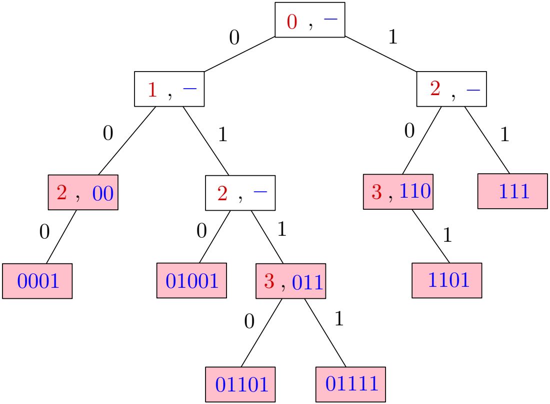
Compressed Tries: Operations

Example: Search(11) - unsuccessful



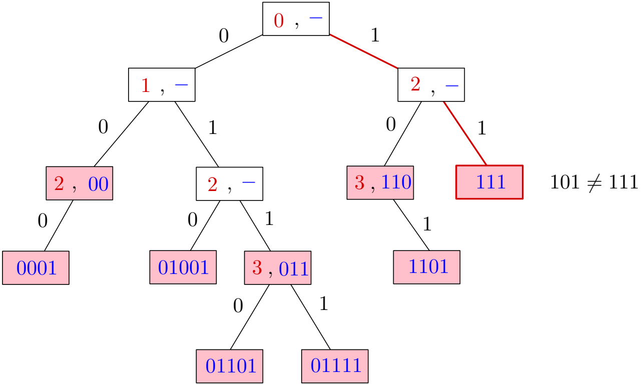
Compressed Tries: Operations

Example: Search(101)



Compressed Tries: Operations

Example: Search(101) - unsuccessful

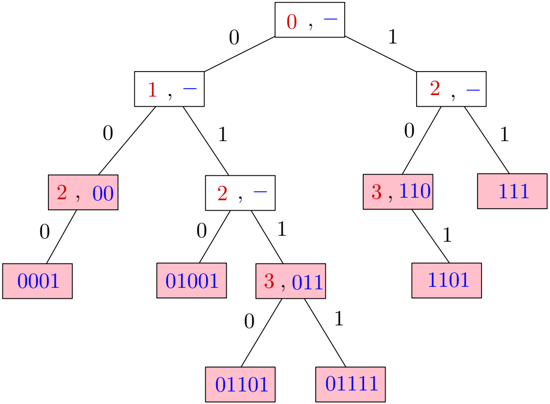


Compressed Tries: Operations

- Delete(x):
  - Perform Search(x)
  - if search ends in an internal node, then
    - if the node has two children, then unflag the node and delete the key
    - else delete the node and make his only child, the child of its parent
  - if search ends in a leaf, then delete the leaf and
  - if its parent is unflagged, then delete the parent

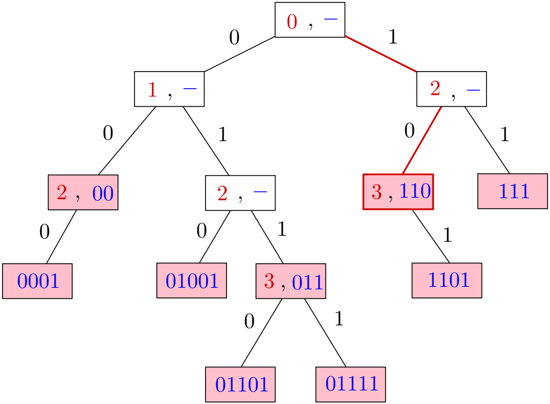
Compressed Tries: Operations

Example: Delete(110)



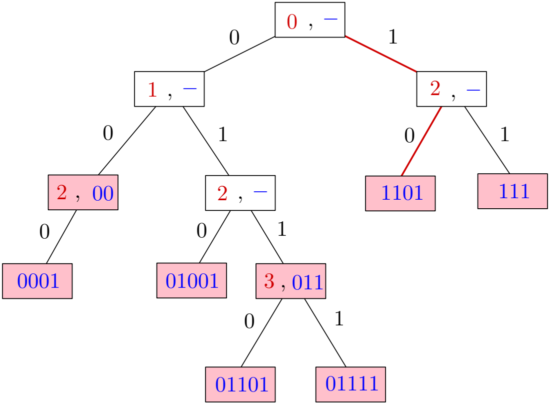
Compressed Tries: Operations

Example: Delete(110)



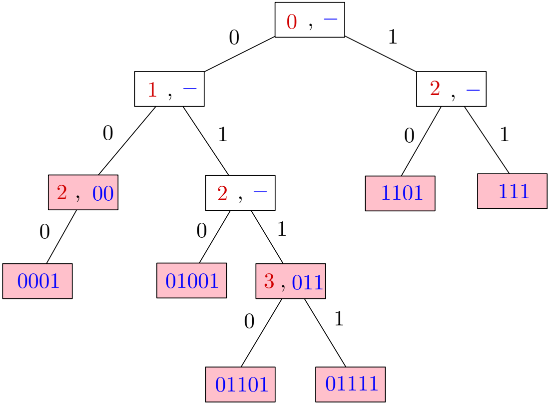
Compressed Tries: Operations

Example: Delete(110)



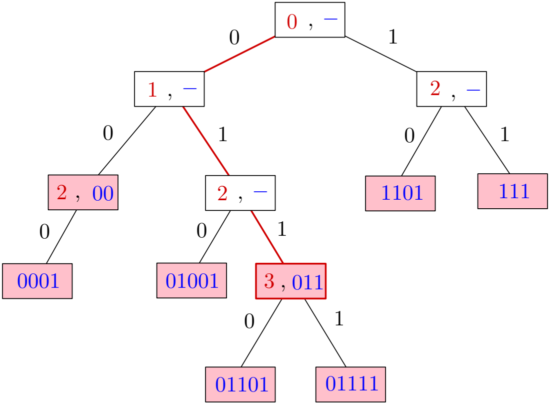
Compressed Tries: Operations

Example: Delete(011)



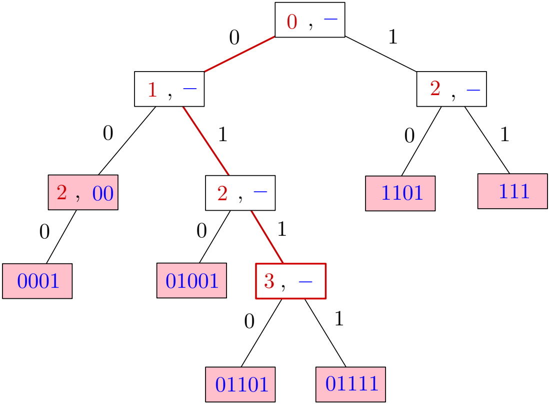
Compressed Tries: Operations

Example: Delete(011)



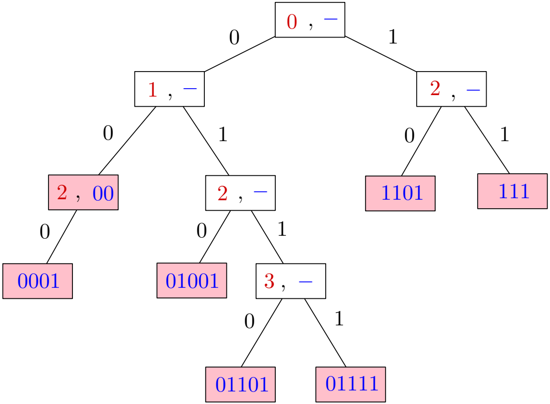
Compressed Tries: Operations

Example: Delete(011)



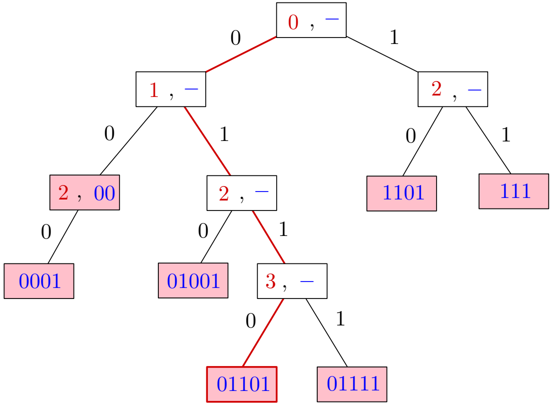
Compressed Tries: Operations

Example: Delete(01101)



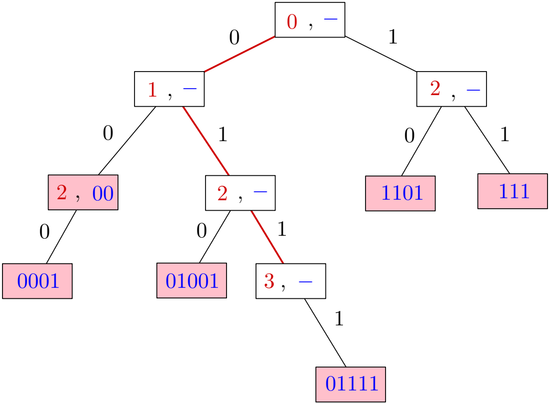
Compressed Tries: Operations

Example: Delete(01101)



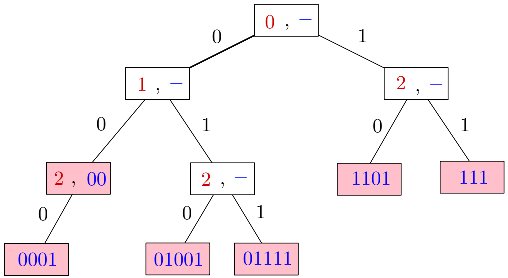
Compressed Tries: Operations

Example: Delete(01101)



Compressed Tries: Operations

Example: Delete(01101)

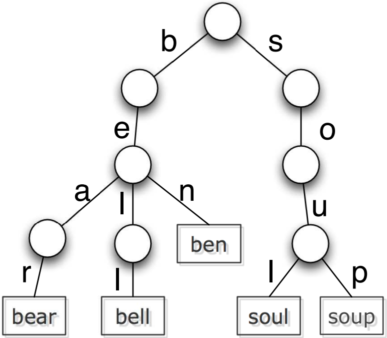


Compressed Tries: Operations

- Insert(x):
  - Perform Search(x)
  - If the search ends at a leaf L with key y, compare x against y to determine the first index i where they disagree.
    Create a new node N with index i.
    Insert N along the path from the root to L so that the parent of N has index $< i$ and one child of N is either L or an existing node on the path from the root to L that has index $> i$.
    The other child of N will be a new leaf node containing x.
  - If the search ends at an internal node, we find the key corresponding to that internal node and proceed in a similar way to the previous case.

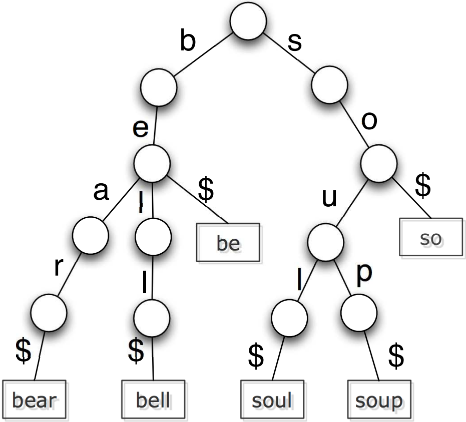
Multiway Tries

- To represent Strings over any fixed alphabet Σ
- Any node will have at most $|\Sigma|$ children
- Example: A trie holding strings $\{$bear, bell, ben, soul, soup$\}$



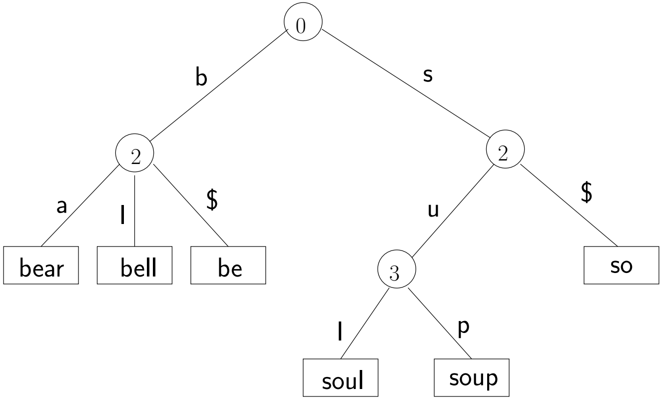
Multiway Tries

- Append a special end-of-word character, say \$, to all keys
- Example: A trie holding strings {bear, bell, be, so, soul, soup}



Multiway Tries

- Compressed multi-way tries
- Example: A compressed trie holding strings {bear, bell, be, so, soul, soup}



Pattern Matching

- Search for a string (pattern) in a large body of text
- $T[0..n-1]$ – The text (or haystack) being searched within
- $P[0..m-1]$ – The pattern (or needle) being searched for
- Strings over alphabet Σ
- Return the first i such that

$$P[j] = T[i+j] \quad \text{for} \quad 0 \leq j \leq m-1$$

- This is the first occurrence of P in T
- If P does not occur in T, return `FAIL`
- Applications:
  - Information Retrieval (text editors, search engines)
  - Bioinformatics
  - Data Mining

Pattern Matching

Example:

- $T =$ "Where is he?"
- $P_1 =$ "he"
- $P_2 =$ "who"

Definitions:

- Substring $T[i..j]$ $0 \le i \le j < n$: a string of length $j - i + 1$ which consists of characters $T[i], \ldots T[j]$ in order
- A prefix of T:
  a substring $T[0..i]$ of T for some $0 \le i < n$
- A suffix of T:
  a substring $T[i..n-1]$ of T for some $0 \le i \le n-1$

General Idea of Algorithms

Pattern matching algorithms consist of guesses and checks:

- A **guess** is a position i such that P might start at $T[i]$. Valid guesses (initially) are $0 \leq i \leq n - m$.
- A **check** of a guess is a single position j with $0 \leq j < m$ where we compare $T[i + j]$ to $P[j]$. We must perform m checks of a single correct guess, but may make (many) fewer checks of an incorrect guess.

We will diagram a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single guess.

Brute-force Algorithm

Idea: Check every possible guess.

BruteforcePM$(T[0..n-1], P[0..m-1])$
T: String of length n (text), P: String of length m (pattern)

for $i \leftarrow 0$ **to** $n-m$ **do**
    match $\leftarrow$ *true*
    $j \leftarrow 0$
    while $j < m$ **and** *match* **do**
        if $T[i+j] = P[j]$ **then**
            $j \leftarrow j+1$
        else
            match $\leftarrow$ *false*
    if *match* **then**
        return i
return `FAIL`

Example

- Example: $T =$ abbbababbab, $P =$ abba

a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							
	a									
		a								
			a							
				a	b	b				
					a					
						a	b	b	a	

- What is the worst possible input?
  $P = a^{m-1}b, \; T = a^n$
- Worst case performance $\Theta((n-m+1)m)$
- $m \leq n/2 \Rightarrow \Theta(mn)$

Pattern Matching

More sophisticated algorithms

- **KMP** and **Boyer-Moore**
- Do extra preprocessing on the pattern P
- We eliminate guesses based on completed matches and mismatches.

KMP Algorithm

- Knuth-Morris-Pratt algorithm (1977)
- Compares the pattern to the text in left-to-right
- Shifts the pattern more intelligently than the brute-force algorithm
- When a mismatch occurs, what is the most we can shift the pattern (reusing knowledge from previous matches)?

$T =$	a	b	c	d	c	a	b	c	?	?	?
	a	b	c	d	c	a	b	a			
						a	b	c	d	c	a

- KMP Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$

KMP Failure Array

T:	a	b	b	c	a	b	c	d	...
P:	a	b	b	c	a	b	a	a	

KMP Failure Array

T:	a	b	b	c	a	b	c	d	...
P:	a	b	b	c	a	b	a	a	

what next slide would match with the text?

KMP Failure Array

T:	a	b	b	c	a	b	c	d	…
P:	a	b	b	c	a	b	a	a	
		a	b	b	c	a	b	a	a

KMP Failure Array

T:	a	b	b	c	a	b	c	d ...		
P:	a	b	b	c	a	b	a	a		
		a	b	b	c	a	b	a	a	
			a	b	b	c	a	b	a	a

KMP Failure Array

T:	a	b	b	c	a	b	c	d	...		
P:	a	b	b	c	a	b	a	a			
		a	b	b	c	a	b	a	a		
			a	b	b	c	a	b	a	a	
				a	b	b	c	a	b	a	a

KMP Failure Array

T:	a	b	b	c	a	b	c	d	…				
P:	a	b	b	c	a	b	a	a					
✗		a	b	b	c	a	b	a	a				
✗			a	b	b	c	a	b	a	a			
✗				a	b	b	c	a	b	a	a		
✓					a	b	b	c	a	b	a	a	

KMP Failure Array

- Define $F[j]$ as the value of the first sliding position past the current one that matches the text T, up to position $T[i-1]$
- This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example).

KMP Failure Array

- Define $F[j]$ as the value of the first sliding position past the current one that matches the text T, up to position $T[i-1]$
- This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example).
- Observation 1: $T[i-j..i-1] = P[0..j-1]$

KMP Failure Array

- Define $F[j]$ as the value of the first sliding position past the current one that matches the text T, up to position $T[i-1] = P[j-1]$
- This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example).
- Observation 1: $T[i-j..i-1] = P[0..j-1]$

KMP Failure Array

- Define $F[j]$ as the value of the first sliding position past the current one that matches the text T, up to position $T[i-1] = P[j-1]$
- This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example).
- Observation 1: $T[i-j..i-1] = P[0..j-1]$
- Observation 2: $T[i-F[j-1]..i-1] = P[0..F[j-1]-1]$

KMP Failure Array

- Define $F[j]$ as the value of the first sliding position past the current one that matches the text T, up to position $T[i-1] = P[j-1]$
- This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example).
- Observation 1: $T[i-j..i-1] = P[0..j-1]$
- Observation 2: $T[i-F[j-1]..i-1] = P[0..F[j-1]-1]$
- Observation 3: $F[j]$ is the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$

KMP Failure Array

- Define $F[j]$ as the value of the first sliding position past the current one that matches the text T, up to position $T[i-1] = P[j-1]$
- This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example).
- Observation 1: $T[i-j..i-1] = P[0..j-1]$
- Observation 2: $T[i-F[j-1]..i-1] = P[0..F[j-1]-1]$
- Observation 3: $F[j]$ is the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- so we can preprocess the pattern to find matches of prefixes of the pattern with the pattern itself

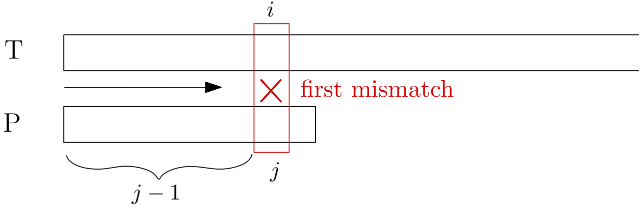
KMP Failure Array

- $F[0] = 0$
- $F[j]$, for $j > 0$, is the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Consider $P =$ abacaba

j	$P[1..j]$	P	$F[j]$
0	—	abacaba	0
1	b	abacaba	0
2	ba	abacaba	1
3	bac	abacaba	0
4	baca	abacaba	1
5	bacab	abacaba	2
6	bacaba	abacaba	3

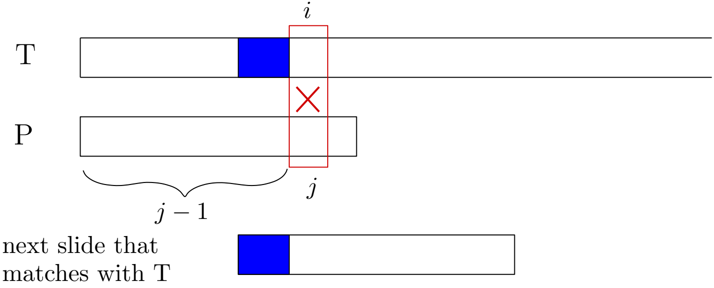
KMP Failure Array

Schematically:



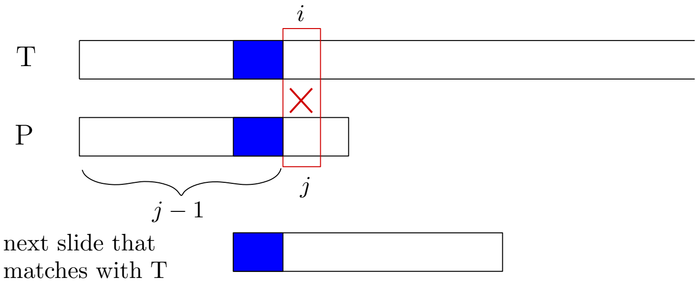
KMP Failure Array

Schematically:



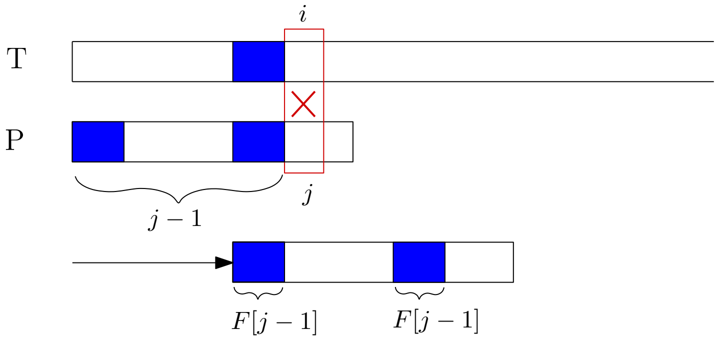
KMP Failure Array

Schematically:



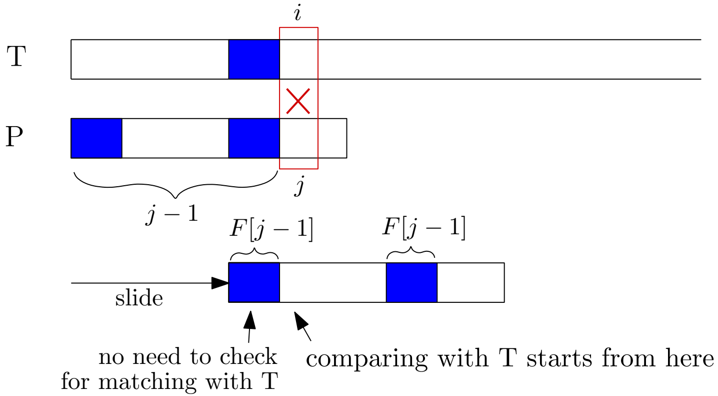
KMP Failure Array

Schematically:



KMP Failure Array

Schematically:



Computing the Failure Array

```
failureArray(P)
P: String of length m (pattern)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m do
        if P[i] = P[j] then
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            j ← F[j − 1]
        else
            F[i] ← 0
            i ← i + 1
```

KMP Algorithm

```
KMP(T, P)
T: String of length n (text), P: String of length m (pattern)
    F ← failureArray(P)
    i ← 0
    j ← 0
    while i < n do
        if T[i] = P[j] then
            if j = m − 1 then
                return i − j //match
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0 then
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 // no match
```

KMP: Example

$P = \texttt{abacaba}$

$T = \underline{\texttt{abaxyabacabb}}\texttt{aababacaba}$

j	0	1	2	3	4	5	6
$F[j]$	0	0	1	0	1	2	3

0	1	2	3	4	5	6	7	8	9	10	11
a	b	a	x	y	a	b	a	c	a	b	b
a	b	a	c								
		(a)	b								
			a				✓				
				a							
					a	b	a	c	a	b	a
									(a)	(b)	a

Exercise: continue with $T = \texttt{abaxyabacabb}\underline{\texttt{aababacaba}}$

KMP: Analysis

failureArray

- At each iteration of the while loop, either
  1. i increases by one, or
  2. the guess index $i - j$ increases by at least one $(F[j-1] < j)$
- There are no more than $2m$ iterations of the while loop
- Running time: $\Theta(m)$

KMP: Analysis

failureArray

- At each iteration of the while loop, either
  1. i increases by one, or
  2. the guess index $i - j$ increases by at least one $(F[j-1] < j)$
- There are no more than $2m$ iterations of the while loop
- Running time: $\Theta(m)$

KMP

- failureArray can be computed in $\Theta(m)$ time
- At each iteration of the while loop, either
  1. i increases by one, or
  2. the guess index $i - j$ increases by at least one $(F[j-1] < j)$
- There are no more than $2n$ iterations of the while loop
- Running time: $\Theta(n)$

KMP: Another Example

- $T =$abacaabaccabacabaabb
- $P =$abacab

Boyer-Moore Algorithm

Based on three key ideas:

- Reverse-order searching: Compare P with a subsequence of T moving backwards
- Bad character jumps: When a mismatch occurs at $T[i] = c$
  - If P contains c, we can shift P to align the last occurrence of c in P with $T[i]$
  - Otherwise, we can shift P to align $P[0]$ with $T[i+1]$
- Good suffix jumps: If we have already matched a suffix of P, then get a mismatch, we can shift P forward to align with the previous occurence of that suffix (with a mismatch from the actual suffix). Similar to failure array in KMP.
- Can skip large parts of T

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

			o								

P = m o o r e

T = b o y e r m o o r e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

			o								
							o				

P = m o o r e

T = b o y e r m o o r e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

			o								
							o				
											o

P = m o o r e

T = b o y e r m o o r e

Bad character examples

P = a l d o
T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
										d	o

P = m o o r e
T = b o y e r m o o r e

b	o	y	e	r	m	o	o	r	e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

			o								
							o				
									l	d	o

P = m o o r e

T = b o y e r m o o r e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

P = m o o r e

T = b o y e r m o o r e

b	o	y	e	r	m	o	o	r	e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

6 comparisons (checks)

P = m o o r e

T = b o y e r m o o r e

b	o	y	e	r	m	o	o	r	e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

6 comparisons (checks)

P = m o o r e

T = b o y e r m o o r e

b	o	y	e	r	m	o	o	r	e
				e					

Bad character examples

P = a l d o
T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

6 comparisons (checks)

P = m o o r e
T = b o y e r m o o r e

b	o	y	e	r	m	o	o	r	e
				e					
				(r)	e				

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

			o								
							o				
								a	l	d	o

6 comparisons (checks)

P = m o o r e

T = b o y e r m o o r e

				e					
				(r)	e				
					(m)				e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

6 comparisons (checks)

P = m o o r e

T = b o y e r m o o r e

b	o	y	e	r	m	o	o	r	e
				e					
				(r)	e				
					(m)			r	e

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

			o								
							o				
								a	l	d	o

6 comparisons (checks)

P = m o o r e

T = b o y e r m o o r e

				e					
				(r)	e				
					(m)	o	o	r	e

7 comparisons (checks)

Good suffix examples

P = sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s

P = odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
								(e)	(l)	(l)	(s)							s

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							s	(e)	(l)	(l)	(s)	␣	s	h	e	l	l	s

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							s	(e)	(l)	(l)	(s)	␣	s	h	e	l	l	s

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										

Good suffix examples

P = sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							s	(e)	(l)	(l)	(s)	␣	s	h	e	l	l	s

P = odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										
				(e)						d								

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							s	(e)	(l)	(l)	(s)	␣	s	h	e	l	l	s

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										
							(o)	(d)							d			

- Good suffix moves further than bad character for 2nd guess.

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							s	(e)	(l)	(l)	(s)	␣	s	h	e	l	l	s

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										
							(o)	(d)							d			

- Good suffix moves further than bad character for 2nd guess.
- Bad character moves further than good suffix for 3rd guess.
- This is out of range, so pattern not found.

Last-Occurrence Function

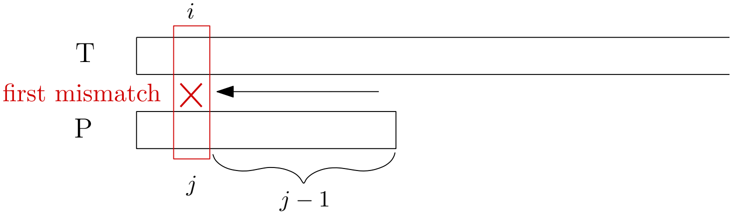
- Preprocess the pattern P and the alphabet Σ
- Build the last-occurrence function L mapping Σ to integers
- $L(c)$ is defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- Example: $\Sigma = \{a, b, c, d\}, P = abacab$

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

- The last-occurrence function can be computed in time $O(m + |\Sigma|)$
- In practice, L is stored in a size-$|\Sigma|$ array.

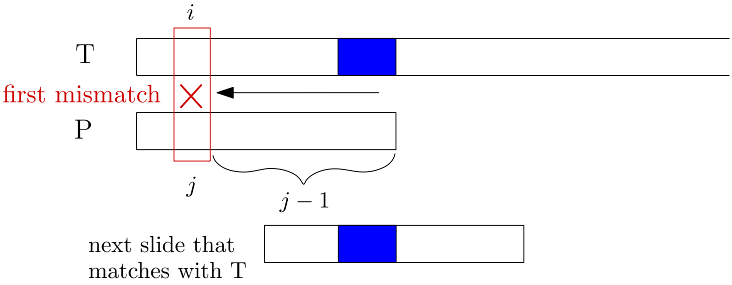
Good Suffix array

Schematically:



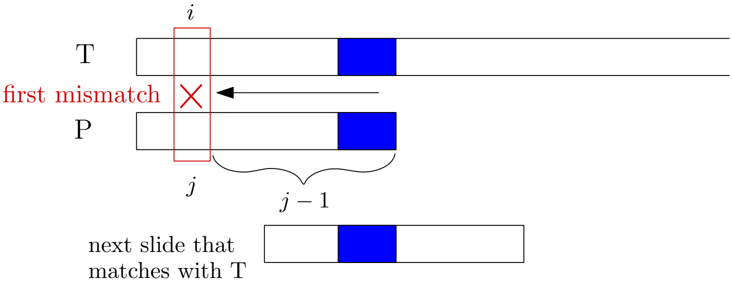
Good Suffix array

Schematically:



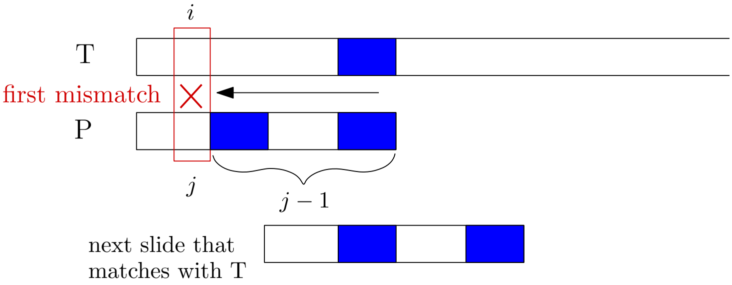
Good Suffix array

Schematically:



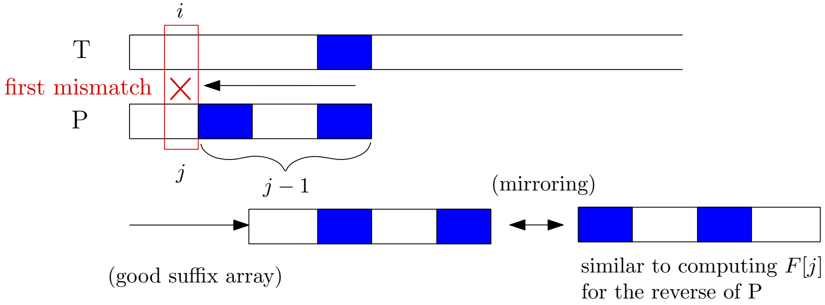
Good Suffix array

Schematically:



Good Suffix array

Schematically:



Boyer-Moore Algorithm

boyer-moore(T,P)

1. $L \leftarrow$ last occurrance array computed from P
2. $S \leftarrow$ good suffix array computed from P
3. $i \leftarrow m-1, \qquad j \leftarrow m-1$
4. **while** $i < n$ **and** $j \geq 0$ **do**
5.   **if** $T[i] = P[j]$ **then**
6. $i \leftarrow i-1$
7. $j \leftarrow j-1$
8.   **else**
9. $i \leftarrow i + m - 1 - \min(L[T[i]], S[j])$
10. $j \leftarrow m-1$
11. **if** $j = -1$ **return** $i+1$
12. **else return** `FAIL`

Exercise: Prove that $i - j$ always increases on lines 9–10.

Boyer-Moore algorithm conclusion

- Worst-case running time $\in O(n + |\Sigma|)$
- This complexity is difficult to prove.
- What is the worst case?
- On typical English text the algorithm probes approximately 25% of the characters in T
- Faster than KMP in practice on English text.

Suffix Tries and Suffix Trees

- What if we want to search for many patterns P within the same fixed text T?
- Idea: Preprocess the text T rather than the pattern P
- Observation: P is a substring of T if and only if P is a prefix of some suffix of T.

We will call a **suffix trie**, a trie that stores all suffixes of a text T, and a **suffix tree** the compressed suffix trie of T.

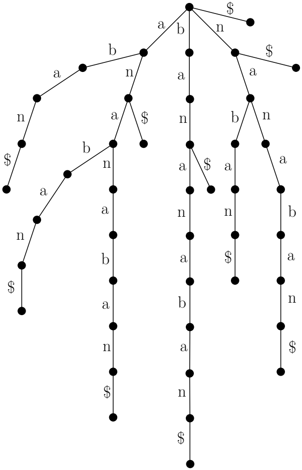
Suffix Trees

- Build the suffix trie, i.e. the trie containing all the suffixes of the text
- Build the suffix tree by compressing the trie above (like in Patricia trees)
- Store two indexes l, r on each node v (both internal nodes and leaves) where node v corresponds to substring $T[l..r]$

Suffix Trie: Example

T =bananaban

{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n}

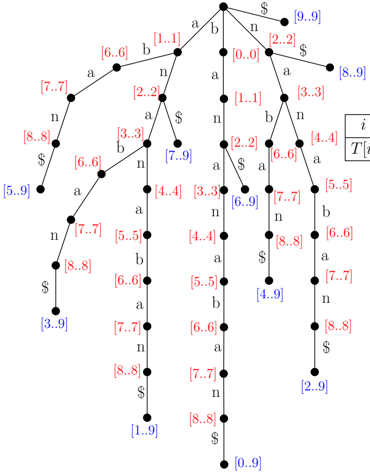


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	$\$$

Suffix Trie: Example

T =bananaban

{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n}

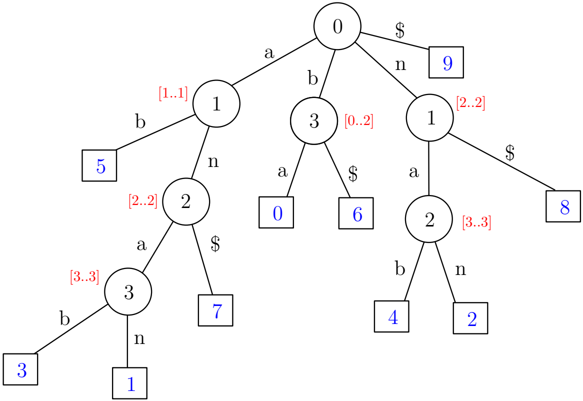


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree (compressed suffix trie): Example

T =bananaban

{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n}



i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	$

Suffix Trees: Pattern Matching

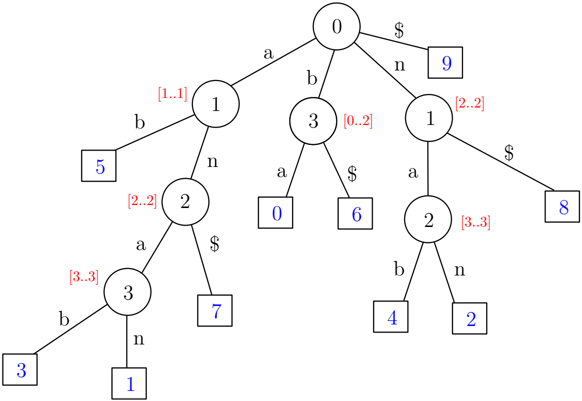
To search for pattern P of length m:

- Similar to Search in compressed trie with the difference that we are looking for a prefix match rather than a complete match
- If we reach a leaf with a corresponding string length less than m, then search is unsuccessful
- Otherwise, we reach a node v (leaf or internal) with a corresponding string length of at least m
- It only suffices, to check the first m characters against the substring of the text between indices of the node, to see if there indeed is a match

Suffix Tree: Example

T = `bananaban`

P = `ana`

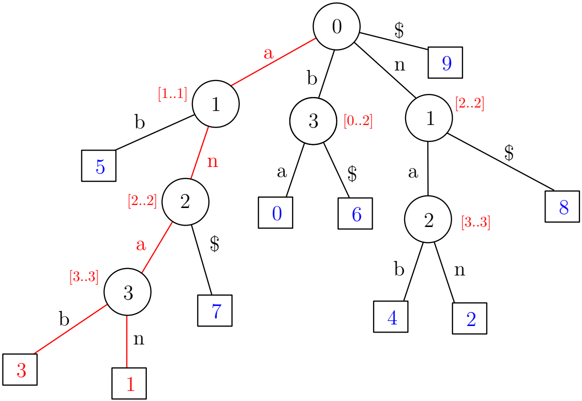


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree: Example

T = `bananaban`

P = `ana`

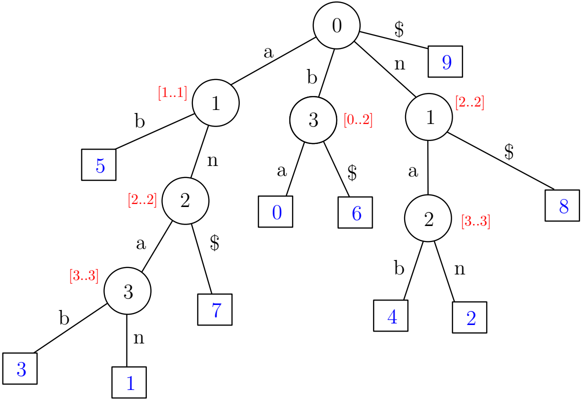


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree: Example

$T =$ `bananaban`

$P =$ `ban`

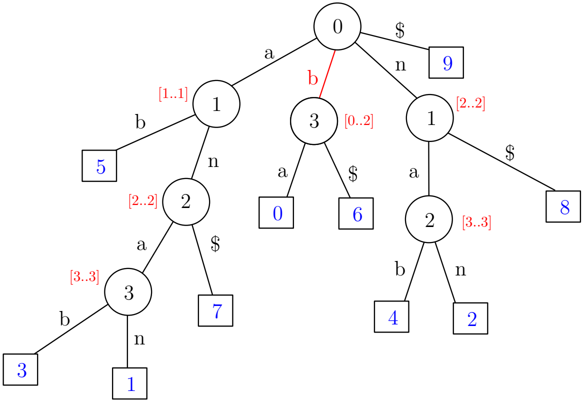


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	$

Suffix Tree: Example

T = `bananaban`

P = `ban`

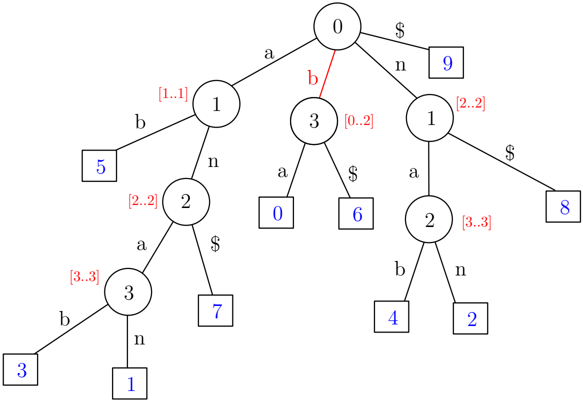


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree: Example

T = bananaban

P = ban

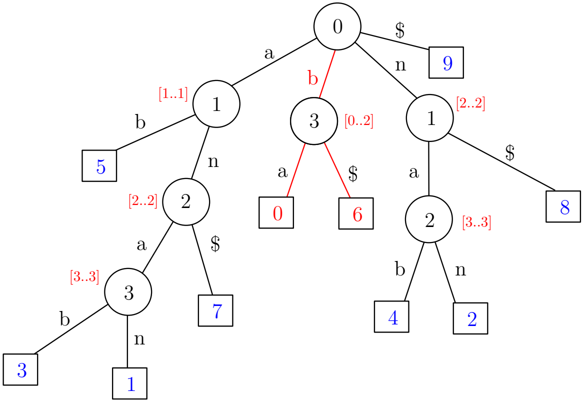


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree: Example

T = `bananaban`

P = `ban`

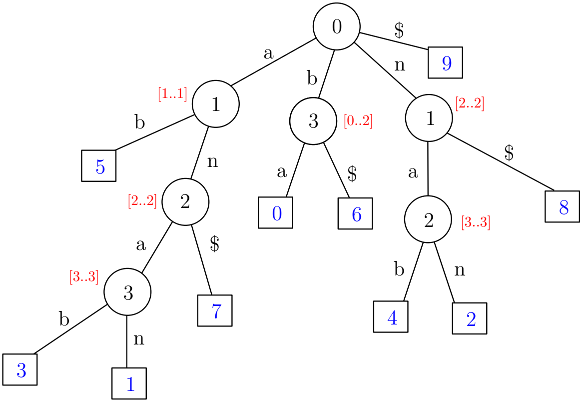


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree: Example

T = `bananaban`

P = `nana`

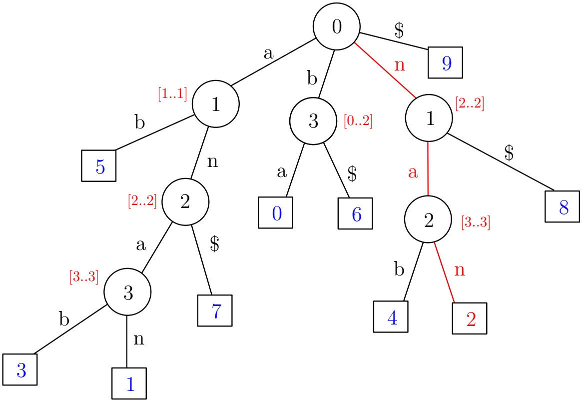


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	$\$$

Suffix Tree: Example

T = `bananaban`

P = `nana`

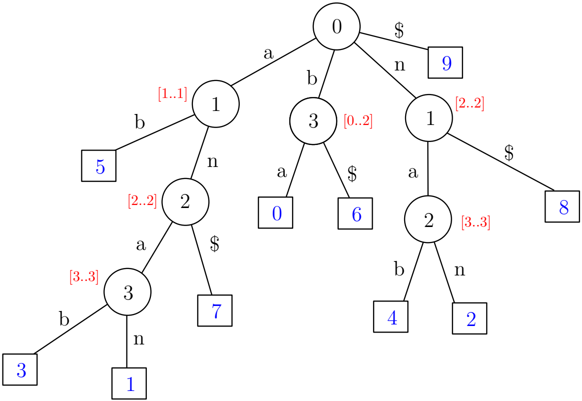


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree: Example

T = `bananaban`

P = `bbn`

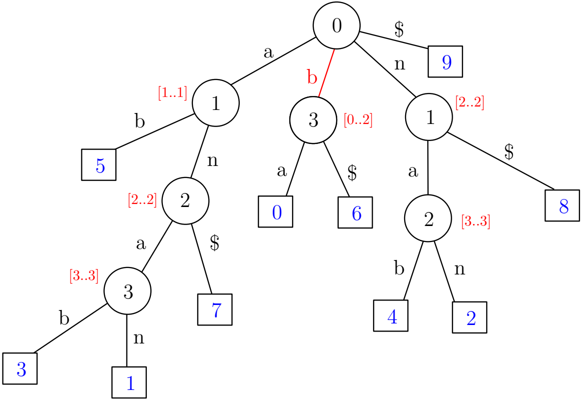


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree: Example

$T =$ `bananaban`

$P =$ `bbn`

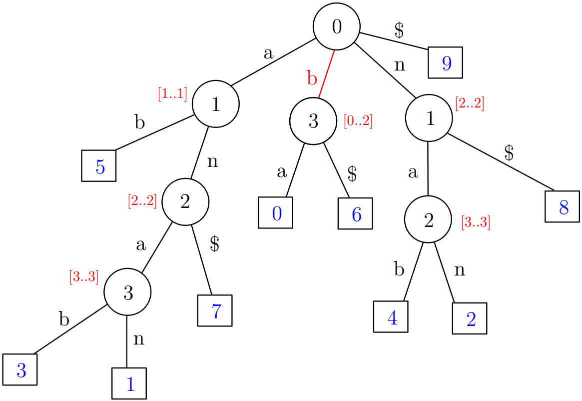


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	$\$$

Suffix Tree: Example

T = bananaban

P = bbn



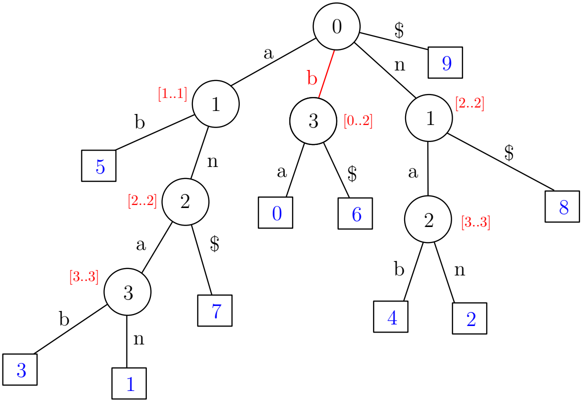
i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

b b n

Suffix Tree: Example

T = `bananaban`

P = `bbn` not found



i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$
	b	b	n							

Pattern Matching Conclusion

	Brute-Force	KMP	BM	Suffix trees
Preproc.:	–	$O(m)$	$O(m + \|\Sigma\|)$	$O(n^2)$ $(\rightarrow O(n))$
Search time:	$O(nm)$	$O(n)$	$O(n)$ (often better)	$O(m)$
Extra space:	–	$O(m)$	$O(m + \|\Sigma\|)$	$O(n)$