Module 8: Tries and String Matching

CS 240 - Data Structures and Data Management

Romain Lebreton
Lectures notes by Arne Storjohann
Based on lecture notes by R. Dorrigiv and D. Roche

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

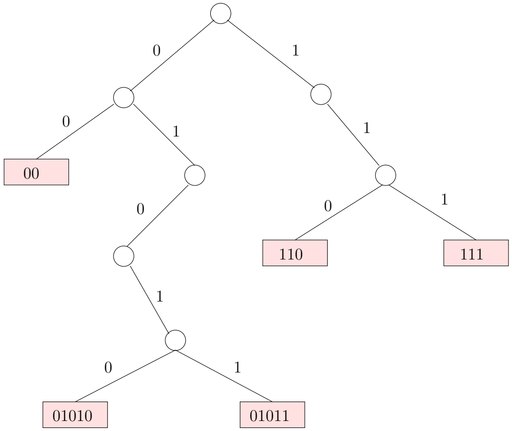
Spring 2014

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:
  - Items (keys) are stored only in the leaf nodes
  - A left child corresponds to a 0 bit
  - A right child corresponds to a 1 bit
- Keys can have different number of bits
- prefix-free: no key is a prefix of another key
- A prefix of a string $S[0..n-1]$:
  a substring $S[0..i]$ of S for some $0 \leq i \leq n-1$

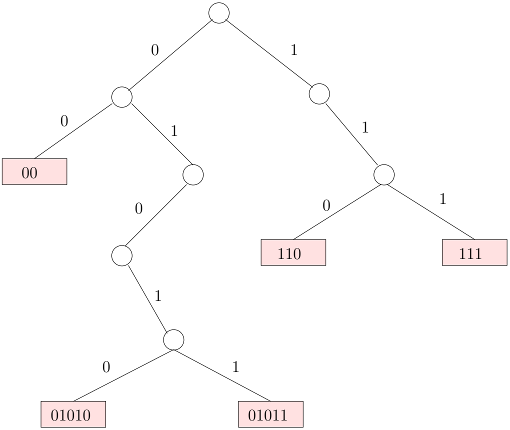
Tries

- Example: A trie for $S = \{00, 110, 111, 01010, 01011\}$



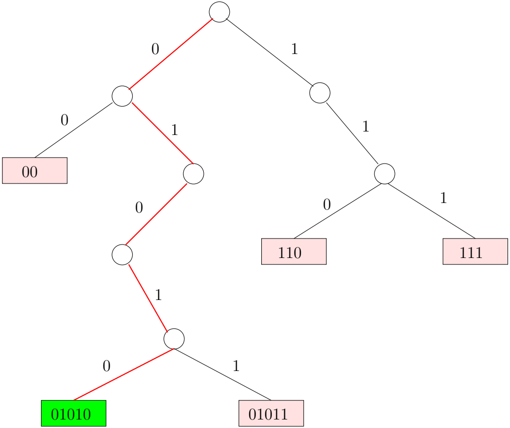
Tries: Search

- Search: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(01010)



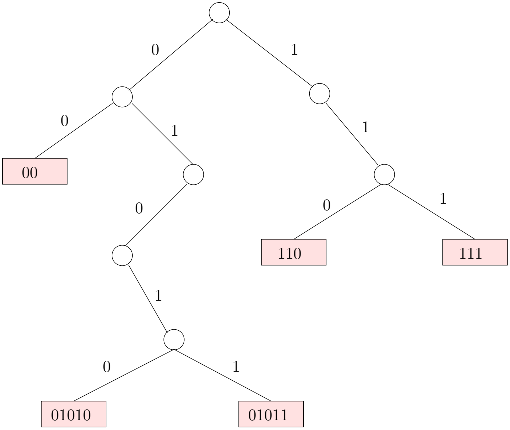
Tries: Search

- Search: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(01010) successful



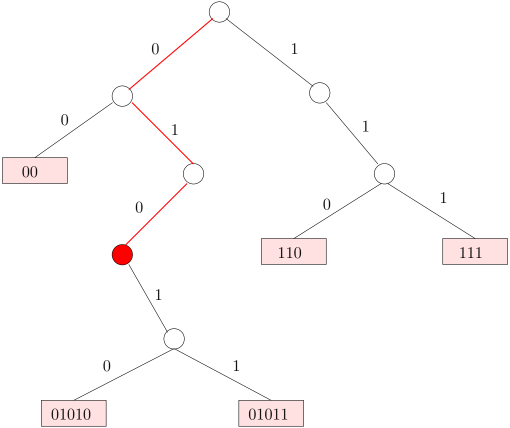
Tries: Search

- Search: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(0100)



Tries: Search

- Search: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(0100) unsuccessful

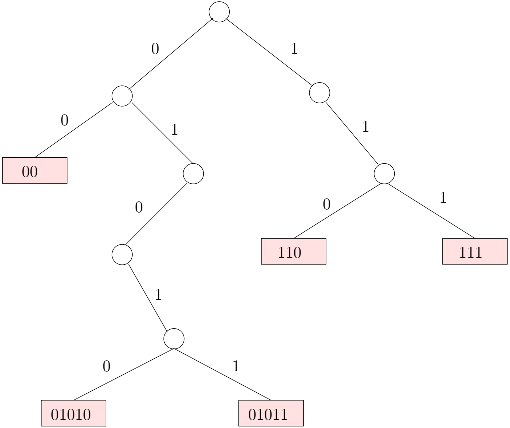


Tries: Insert

- 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 with a key $y \neq x$, then y is a prefix of x: not possible because our keys are prefix-free
  - If we finish at an internal node and there are no extra bits: not possible because our keys are prefix-free
  - If we finish at an internal node and there are extra bits: expand trie by adding necessary nodes that correspond to extra bits

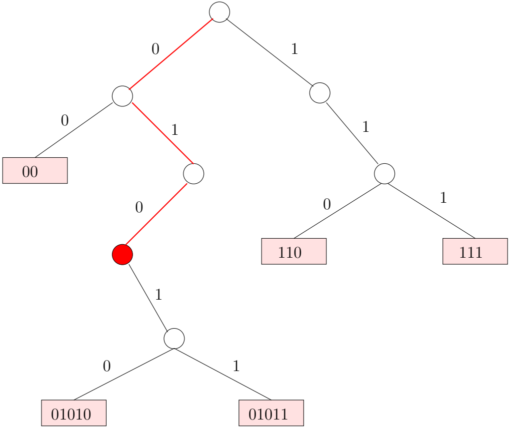
Tries: Insert

- Insert(x)
- Example: Insert(01000)



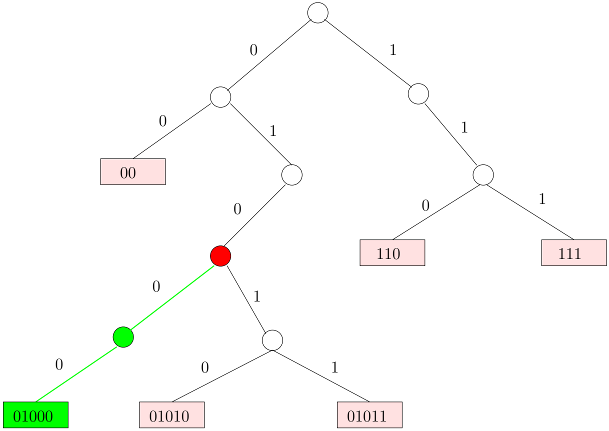
Tries: Insert

- Insert(x)
- Search(01000) unsuccessful Extra bits: 00



Tries: Insert

- Insert(x)
- Search(01000) unsuccessful Extra bits: 00

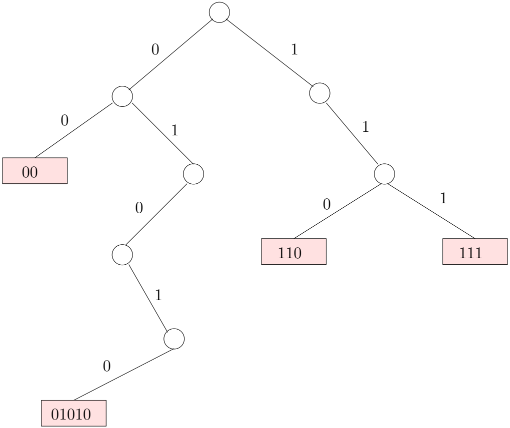


Tries: Delete

- Delete(x)
  - Search for x to find the leaf v_x
  - Delete v_x and all ancestors of v_x until we reach an ancestor that has two children

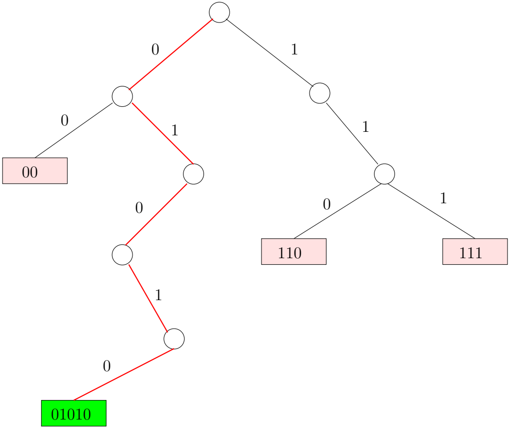
Tries: Delete

- Delete(x)
- Example: Delete(01010)



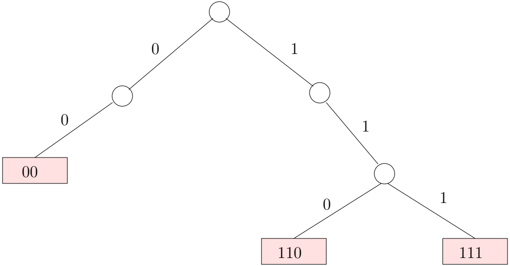
Tries: Delete

- Delete(x)
- Example: Delete(01010)
- Search(01010) successful



Tries: Delete

- Delete(x)
- Example: Delete(01010)



Tries: Operations

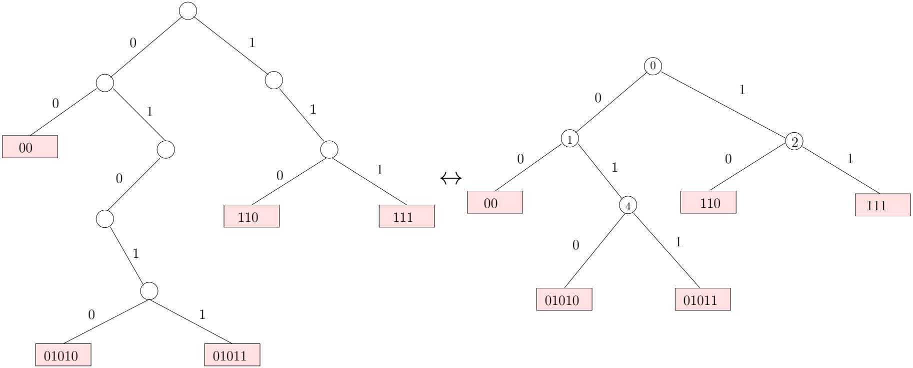
- Search(x)
- Insert(x)
- Delete(x)
- Time Complexity of all operations: $\Theta(|x|)$
  $|x|$: length of binary string x, i.e., the number of bits in x

Compressed Tries (Patricia Tries)

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

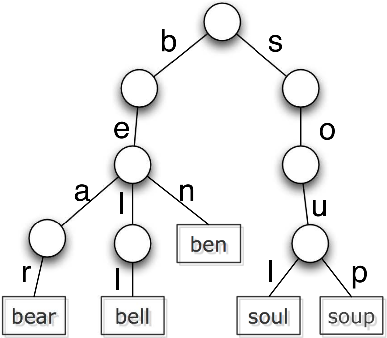
- Search(x):
  - Follow the proper path from the root down in the tree to a leaf
  - If search ends in an internal node, it is unsuccessful
  - In search ends in a leaf, we need to check again if the key stored at the leaf is indeed x
- Delete(x):
  - Perform Search(x) to find a leaf
  - Delete the leaf and its parent

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.

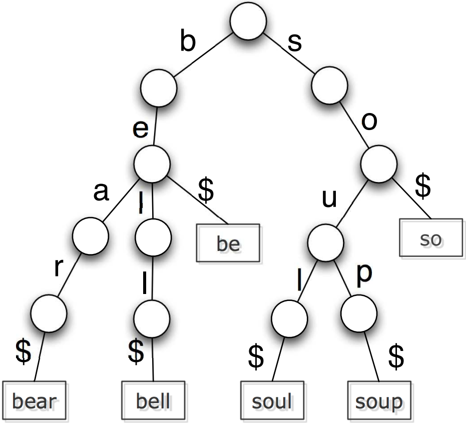
Multiway Tries

- 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}



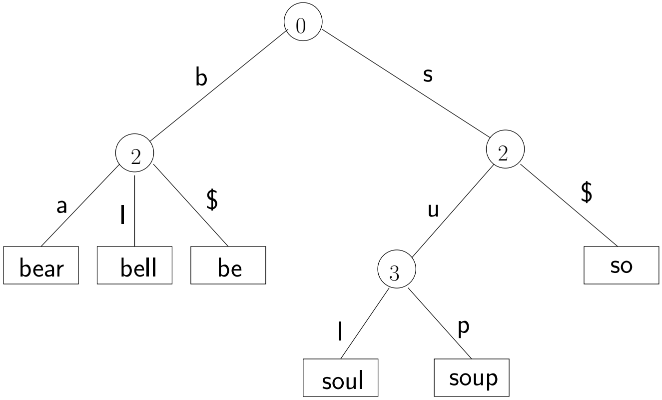
Multiway Tries

- Allow strings that are prefixes of other strings:
  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 $T[i..j]$ $0 \leq i \leq j < n$: 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 $0 \leq i < n$
- A suffix of T:
  a substring $T[i..n-1]$ of T for some $0 \leq i \leq n-1$

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 ← 0 to n − m do
        match ← true
        j ← 0
        while j < m and match do
            if T[i + j] = P[j] then
                j ← j + 1
            else
                match ← 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

- Preprocess the pattern to find matches of prefixes of the pattern with the pattern itself
- The failure array F of size m: $F[j]$ is defined as the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- $F[0] = 0$
- If a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F[j-1]$
- 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 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

$P =$ abacaba
$T =$ abaxyabacabbaababacaba

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 $T =$ abaxyabacabbaababacaba

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: 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 occurance 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

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

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

			o								
							o				
										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

			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

			o								
							o				
								a	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

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

			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)			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.

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P = $ bonobobo

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$								

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$								6

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$							2	6

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$						-1	2	6

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: P = bonobobo

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$					2	-1	2	6

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$				-3	2	-1	2	6

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$	-6	-5	-4	-3	2	-1	2	6

- Computed similarly to KMP failure array in $\Theta(m)$ time.

Boyer-Moore Algorithm

```
boyer-moore(T,P)
1.    L ← last occurrance array computed from P
2.    S ← suffix skip array computed from P
3.    i ← m − 1,      j ← m − 1
4.    while i < n and j ≥ 0 do
5.        if T[i] = P[j] then
6.            i ← i − 1
7.            j ← j − 1
8.        else
9.            i ← i + m − 1 − min(L[T[i]], S[j])
10.           j ← 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 Trees (Suffix Tries)

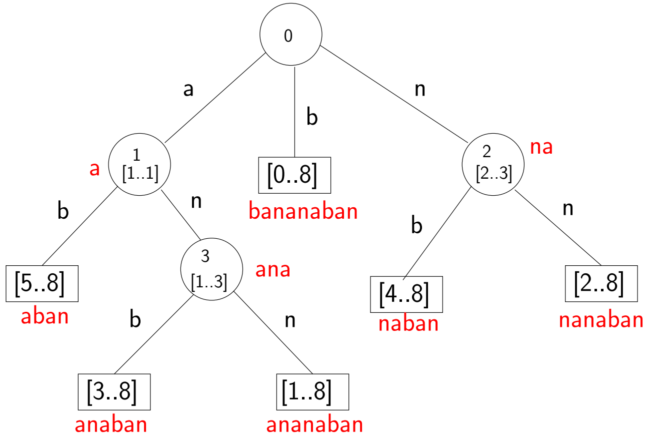
- What if we want to search for for many patterns P within the same fixed text T?
- Idea: Preprocess the text T rather than the pattern P
- Observation: P is a substring of T if and only if P is a prefix of some suffix of T.

Suffix Trees

- Build a compressed trie that stores all suffixes of text T
- Insert suffixes in decreasing order of length
- If a suffix is a prefix of another suffix, we do not insert it
- Store two indexes l, r on each node v (both internal nodes and leaves) where node v corresponds to substring $T[l..r]$

Suffix Trees: Example

T =`bananaban`



0	1	2	3	4	5	6	7	8
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 of that string to see if there indeed is a match

Pattern Matching Conclusion

	Brute-Force	KMP	Boyer-Moore	Suffix trees
Preprocessing:	–	$O(m)$	$O(m + \lvert\Sigma\rvert)$	$O(n^2)$
Search time:	$O(nm)$	$O(n)$	$O(n)$ (often better)	$O(m)$
Extra space:	–	$O(m)$	$O(m + \lvert\Sigma\rvert)$	$O(n)$