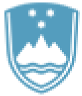


REPUBLIC OF SLOVENIA
MINISTRY OF EDUCATION, SCIENCE AND SPORT

University *of Ljubljana*

Investing in your future
OPERATION PART FINANCED BY THE EUROPEAN UNION
European Social Fund

LADS^3 2014, Sept. 1-5

Ljubljana Algorithms and Data Structures Summer School 2014

Search in unstructured text

Andrej Brodnik
University of Ljubljana

Ljubljana Algorithms and Data Structures Summer School 2014

Search in unstructured text

Andrej Brodnik
University of Ljubljana

Ljubljana, September 1-5, 2014

Text

lad /lad/ noun; noun: lad; plural noun: lads

1. informal: a boy or young man (often as a form of address). ”come in, lad, and shut the door” synonyms: boy, schoolboy, youth, youngster, juvenile, stripling, young fellow

...

Google

- consiting of words separated by spaces and/or punctiation marks
- in general, words are sequences of charactres from an alphabet $\Sigma = \{A, B, \ldots, z\}$.
- roughly speaking, the query is to find some word
- we can build an index of words using hashing

- perhaps we want to search for a words starting with some prefix (e.g. “you”) or ending by some postfix (e.g. “boy”)
- or even for words with a typo: “bay”, “jouvenille”, ...

Text – a bit different

TAACCCTA ACCCTAAC CCTAACCC TAACCCTA ACCCTAAC CCTAACCC TAACCCTA ACCCTAAC CCTAACCC TAACCCTA ACCCTAAC CCTAACCC TAAC ...

begining of a human genome, `http://lusy.fri.uni-lj.si/en/node/102`

- no separator to make words
- still sequence of charactres from an alphabet $\Sigma = \{A, C, G, T\}$.
- roughly speaking, the query is to find a patern p in a text T
- we can build .... what?

- perhaps one can use notion of coding and noncoding DNA?
- similar problem with RNA (4 characters), amino acids (23), music (12), ..., any text (up to size of UNICODE)

Problem definition

We have an (unstructured) text T built of characters from an alpabet Σ, that is $T = \Sigma^*$.

The text is not updated (or it is very seldomly) – i.e. we have a semi-dynamic data strucutre.

Besides we have query patterns p_1 and p_2 and we can ask the following queries on T:

Q1 : Is p_1 in T?
Q2 : How many times does p_1 occure in T?
Q3 : Is p_1 before or after p_2 in T?
Q4 : What is the smallest distance between p_1 and p_2 in T?
Q5 : How many times is p_1 at most k characters away from p_2 in T?
...

Trivial solution

Assume

$$n = |T| \quad m = |p_1| + |p_2| \quad .$$

We can answer queries in

$$O(|T| + |p_1| + |p_2|) = O(m + n)$$

using Knuth-Morris-Pratt string matching algorithm (KMP77) or on average even in

$$O(|T| + |p_1| + |p_2|) = O(m + n)$$

using Boyer-Moor algorithm (BM75).

But can we do it in

$$O(|p_1| + |p_2|) = O(m)$$

at additional cost for preprocessing of T?

It is reasonable to expect the preprocessing cost is $O(n^2) = O(|T|^2)$ or even $O(n \log n) = O(|T| \log |T|$. It would be great to have $O(n) = O(|T|)$.

Solution Sketch

From a text T build an index data structure I_T, that permits efficient queries of the form

$$\texttt{Positions}(I_T\texttt{, } p\texttt{) ->} \ R = \{x_1, x_2, \ldots, x_r\}$$

where the returned values are positions of a pattern p in text T.

Obviously, we can sort the returned values in time $O(r \log r)$ or even in $O(n)$ as they are from a finite set $1, 2, \ldots, n$.

Answering queries

$$\texttt{Positions(}I_T\texttt{, }p_1\texttt{) -> } R_1 = \{x_1, x_2, ..., x_{r1}\}$$
$$\texttt{Positions(}I_T\texttt{, }p_2\texttt{) -> } R_2 = \{y_1, y_2, ..., y_{r2}\}$$

Assume running time of `Positions` is $O(m)$ and sorting R_i takes $O(s_i)$.

Q1 : Is p_1 in T?
If $r > 0$ the answer is true and otherwise false.

$O(m + 1)$

Q2 : How many times does p_1 occure in T?
Just return $r1$.

$O(m + 1)$

Q3 : Is p_1 before or after p_2 in T?

Find the smallest x_i and y_j and return the proper result.

$$O(m + r1 + r2)$$

Q4 : What is the smallest distance between p_1 and p_2 in T?

Sort R_1 and R_2 and find the smallest distance.

$$O(m + s_1 + s_2)$$

Q5 : How many times is p_1 at most k characters away from p_2 in T?

Similarly, sort R_1 and R_2 and walk through the lists and report all pairs with a distnace smaller than k.

$$O(m + s_1 + s_2)$$

...

It remains to implement `Positions`.

NOTE: if the returned R_i is already sorted the running times change accordingly.

Example

```
0        1         2         3         4         5
12345678901234567890123456789012345678901234567890
GGTCAGCTCAAAGGTTCCGAACTCGGGCCTGGGATGTGAGGGGGTACTGA

5        6         7         8         9         0
12345678901234567890123456789012345678901234567890
ACCTAAGTTCTCCGAATCTTTAGCCCTGGGTCCCTAAGTCGCATCCTTAA
```

`Positions(`I_T`, AAG) ->` $\{11, 86, 55\}$

`Positions(`I_T`, TG) ->` $\{48, 37, 30, 77, 35\}$

Q1 : Is `AAG` in T? – `true`

Q2 : How many times does `AAG` occure in T? – 3

Q3 : Is `AAG` before or after `TG` in T? – `true`

Q4 : What is the smallest distance between `AAG` and `TG` in T? – 7

Q5 : How many times is `AAG` at most 17 characters away from `TG` T? – ???

...

Model of Computation

- Comparison based
- RAM – cell-probe based

When RAM, the access time is important:

- Memory hierarchy: registers; L1 & L2 cache; main memory; disk (SSD or HDD).
- Size and delay increasing.
- Sequential access (scanning) is faster per memory cell: n/B, B is the amount of data transferred in the hierarchy.
- Cache aware algorithms and cache oblivious algorithms.

- Multiprocessor models

From text to suffixes

Assume we have a text T:

```
0         1         2
12345678901234567890
TAACCCTAACCCTAACCCTA
```

and we consider all suffixes t_i of T\$, where \$ $\notin \Sigma$:

$(\texttt{TAACCCTAACCCTAACCCTA\$}, 1)$	$(\texttt{AACCCTAACCCTA\$}, 8)$	$(\texttt{ACCCTA\$}, 15)$
$(\texttt{AACCCTAACCCTAACCCTA\$}, 2)$	$(\texttt{ACCCTAACCCTA\$}, 9)$	$(\texttt{CCCTA\$}, 16)$
$(\texttt{ACCCTAACCCTAACCCTA\$}, 3)$	$(\texttt{CCCTAACCCTA\$}, 10)$	$(\texttt{CCTA\$}, 17)$
$(\texttt{CCCTAACCCTAACCCTA\$}, 4)$	$(\texttt{CCTAACCCTA\$}, 11)$	$(\texttt{CTA\$}, 18)$
$(\texttt{CCTAACCCTAACCCTA\$}, 5)$	$(\texttt{CTAACCCTA\$}, 12)$	$(\texttt{TA\$}, 19)$
$(\texttt{CTAACCCTAACCCTA\$}, 6)$	$(\texttt{TAACCCTA\$}, 13)$	$(\texttt{A\$}, 20)$
$(\texttt{TAACCCTAACCCTA\$}, 7)$	$(\texttt{AACCCTA\$}, 14)$	$(\texttt{\$}, 21)$

From text to suffixes – 1

If we can make a dictionary-like data structure that:

- stores pairs (t_i, i); and
- for a pattern p returns all those i, where p is a *prefix* of t_i

we have a solution to our problem.

From text to suffixes – example

Assume we have a text T:

```
0         1         2
12345678901234567890
TAACCCTAACCCTAACCCTA
```

and we consider all suffixes t_i of T`$`, where `$` $\notin \Sigma$:

(TAACCCTAACCCTAACCCTA\$, 1)	(AACCCTAACCCTA\$, 8)	(ACCCTA\$, 15)
(AACCCTAACCCTAACCCTA\$, 2)	(ACCCTAACCCTA\$, 9)	(CCCTA\$, 16)
(ACCCTAACCCTAACCCTA\$, 3)	(CCCTAACCCTA\$, 10)	(CCTA\$, 17)
(CCCTAACCCTAACCCTA\$, 4)	(CCTAACCCTA\$, 11)	(CTA\$, 18)
(CCTAACCCTAACCCTA\$, 5)	(CTAACCCTA\$, 12)	(TA\$, 19)
(CTAACCCTAACCCTA\$, 6)	(TAACCCTA\$, 13)	(A\$, 20)
(TAACCCTAACCCTA\$, 7)	(AACCCTA\$, 14)	(\$, 21)

`Positions(`I_T`, AAC) ->` $\{2, 8, 14\}$

Outline

The rest of the lecture:

- Trivial solution – sorted array
- Improved solution – suffix array (Manber & Myers 1990)
- Tries (de la Briandais 1959 and Fredkin 1960), PATRICIA (Morrison 1968) and LC-tries (Andersson & Nilsson 1993); Suffix trees or PAT tree (Weiner 1973)
- vEB structure (van Emde Boas 1975)
- COSD (Brodal & Fagerberg 2006)
- Luleå algorithm (Degermark et al. 1997)
- ERA (Mansour et al. 2011)
- String B trees (Ferragina & Grossi 1998)
- Some empirical results (Lipnik 2014, Mesić 2014, Jekovec 2014)

General framework

Quantities:

- We have a text T of length $|T| = n$.
- The length of a query pattern p is $|p| = m$.

- We preprocess the text T in time $T_p(n)$ to produce an index structure I_T.
- The space complexity accounts for T and I_T.
- We ran the query

  $$\texttt{Positions(}I_T\texttt{, }p\texttt{) ->} \{x_1, x_2, \ldots, x_r\}$$

  in time $T_q(n, m, r)$, which will (hopefully) be just $T_q(m, r)$.

Trivial solution – sorted array

Assume we have a text T:

```
0         1         2
12345678901234567890
TAACCCTAACCCTAACCCTA
```

The data structure I_T:

(AACCCTAACCCTAACCCTA$, 2)
(ACCCTAACCCTAACCCTA$, 3)
(A$, 20)
(CCCTA$, 16)
(CCTA$, 17)
(CTA$, 18)
(TAACCCTA$, 13)
(AACCCTAACCCTA$, 8)
(ACCCTAACCCTA$, 9)
(CCCTAACCCTAACCCTA$, 4)
(CCTAACCCTAACCCTA$, 5)
(CTAACCCTAACCCTA$, 6)
(TAACCCTAACCCTAACCCTA$, 1)
(TA$, 19) & ($, 21)
(AACCCTA$, 14)
(ACCCTA$, 15)
(CCCTAACCCTA$, 10)
(CCTAACCCTA$, 11)
(CTAACCCTA$, 12)
(TAACCCTAACCCTA$, 7)

`Positions(`I_T`, AAC) ->` $\{2, 8, 14\}$

In general order of solution sequence needs not to be increasing.

Sorted array – analysis

Preprocessing:

- Sort n strings into array of strings.
- Time complexity $\Omega(n \log n)$.
- Space complexity: $O(n + n^2)$, in fact we do not need T any more, so $O(n^2)$.

Query:

- Use bisection to find the entry and then lineary search arround to get all matching entries.
- Time complexity $O(m \log n + r)$.

Suffix array

We want to reduce the space complexity. For example with a genome of $3\ 10^9$ base pairs, this explodes to more than 10^{18} memory locations.

```
0         1          2
12345678901234567890
TAACCCTAACCCTAACCCTA
```

(AACCCTAACCCTAACCCTA$, 2)
(ACCCTAACCCTAACCCTA$, 3)
(A$, 20)
(CCCTA$, 16)
(CCTA$, 17)
(CTA$, 18)
(TAACCCTA$, 13)
(AACCCTAACCCTA$, 8)
(ACCCTAACCCTA$, 9)
(CCCTAACCCTAACCCTA$, 4)
(CCTAACCCTAACCCTA$, 5)
(CTAACCCTAACCCTA$, 6)
(TAACCCTAACCCTAACCCTA$, 1)
(TA$, 19) & ($, 21)
(AACCCTA$, 14)
(ACCCTA$, 15)
(CCCTAACCCTA$, 10)
(CCTAACCCTA$, 11)
(CTAACCCTA$, 12)
(TAACCCTAACCCTA$, 7)

We can replace I_T with a simple array of pointers into a text:

$$[2, 8, 14, 3, 9, 15, 20, 4, 10, 16, 5, 11, 17, 6, 12, 18, 1, 7, 13, 19, 21]$$

Suffix array – example

```
0         1          2
12345678901234567890
TAACCCTAACCCTAACCCTA
```

$$[2, 8, 14, 3, 9, 15, 20, 4, 10, 16, 5, 11, 17, 6, 12, 18, 1, 7, 13, 19, 21]$$

and a query

Positions(I_T, AAC): 16, 9, 14, 3, 8, 2 -> $\{14, 8, 2\}$

Suffix array – analysis

Preprocessing:

- Sort n indices into T.
- Time complexity $\Omega(n \log n)$.
- Space complexity: $O(n + n) = O(n)$.
- However, there are n characters from Σ and n references into T. In reality this is n bytes for T (or less – DNA) and n 64-bit references.

Query:

- Use bisection to find the entry and then lineary search arround to get all matching entries.
- Time complexity $O(m \log n + r)$.

Suffix array – RAM model notes

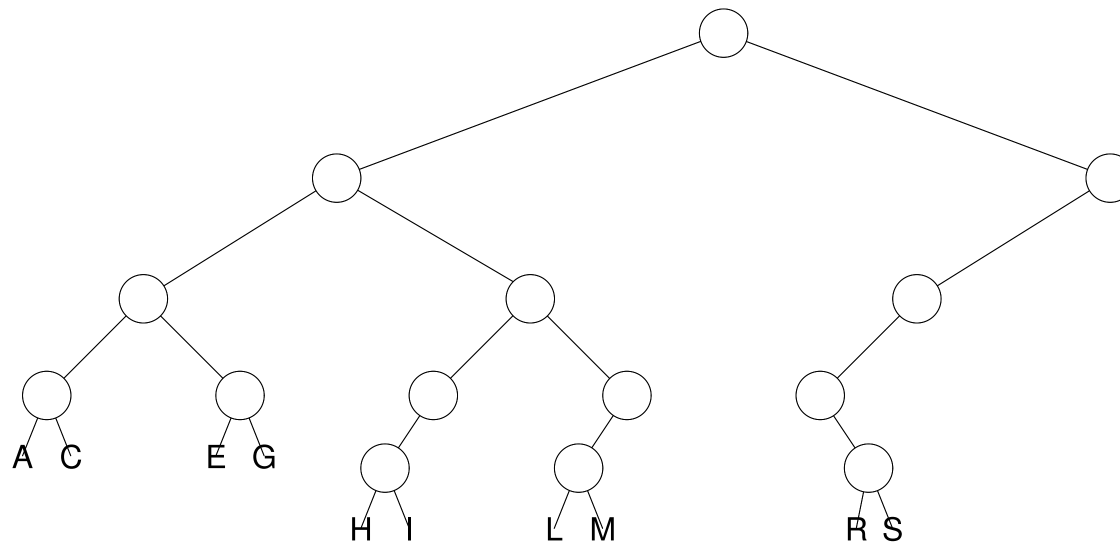
- Sorting can be done in
$$O(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B})$$
memory accesses (Arge 2001, Vitter 2001, Brodal, Fagerberg, Vinther 2007).
- However, the binary search requires a lot of jumps in the data structure. We would always want to divide n by B. That is query in
$$O(\frac{m}{B}\log\frac{n}{B}+\frac{r}{B})$$
and not in
$$O(m\log n+\frac{r}{B})$$
memory accesses.
- Issue of locality in a data structure.

Tries – example

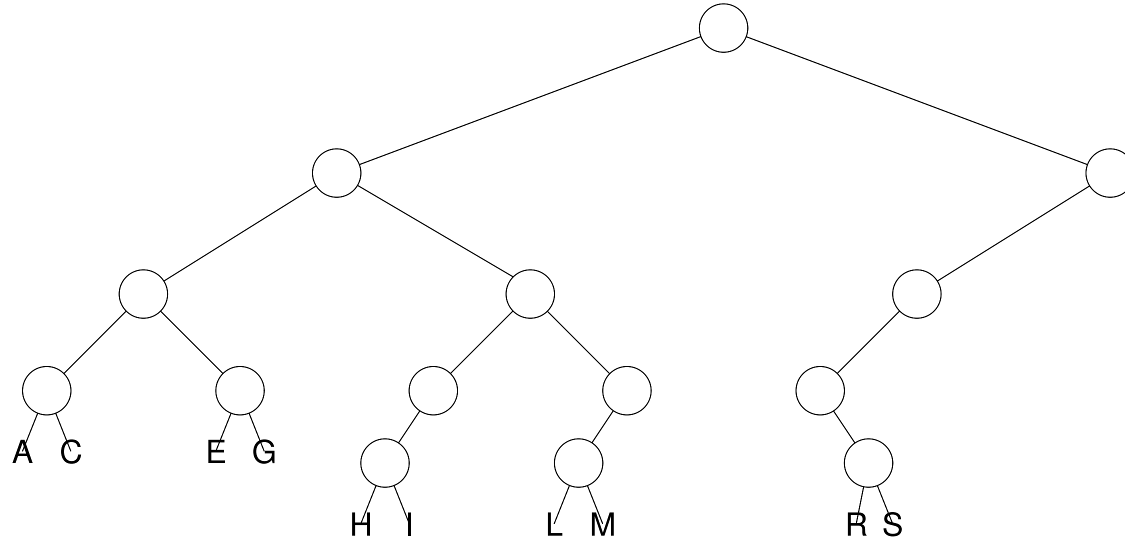
- Let $\Sigma = \{0, 1\}$.
- Assume we have values (t_i, i):

$$(\texttt{00001}, A), (\texttt{00011}, C), (\texttt{00101}, E), (\texttt{00111}, G), (\texttt{01000}, H),$$
$$(\texttt{01001}, I), (\texttt{01100}, L), (\texttt{01101}, M), (\texttt{10010}, R), (\texttt{10011}, S)$$

Then we construct a trie:



Trie – query



Positions(I_T, 001): L, L, R -> $\{E, G\}$

Trie – analysis

Preprocessing:

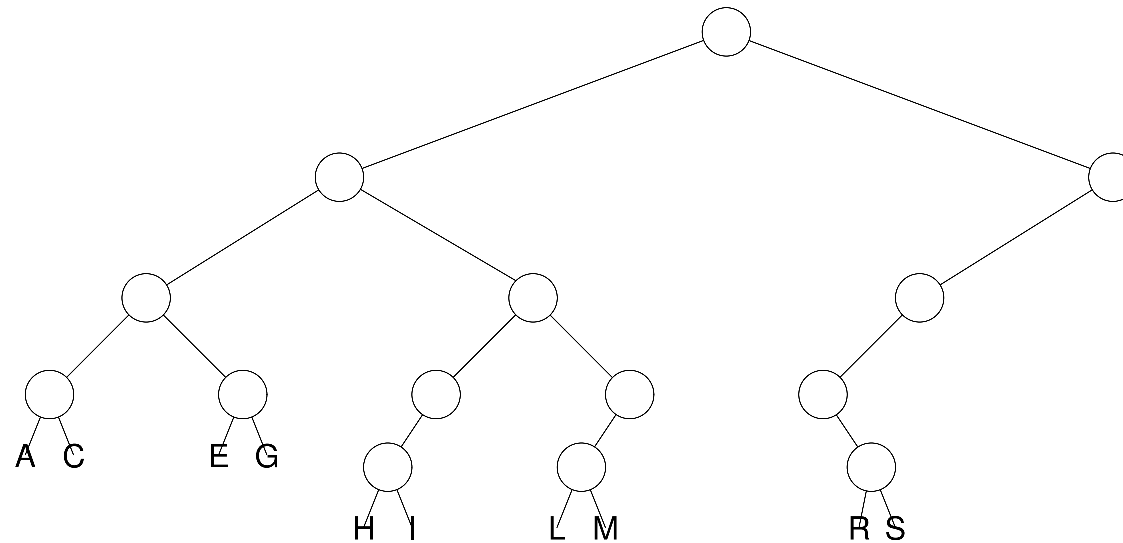
- Insert n strings of T in trie.
- Time complexity $O(n^2)$.
- Space complexity: $O(n + n^2)$.
- However, there are also associated 2 64-bit references with each node.

Query:

- Follow path from the root – $O(m)$ ...
- ... and than visit complete subtree – expensive.
- Solution:
  - Link all leaves in a linked list – additional n references.
  - have a references from a node to the left most and right most child of a subtree – 2 references per node.
- Consequently, time complexity $O(m + r)$.

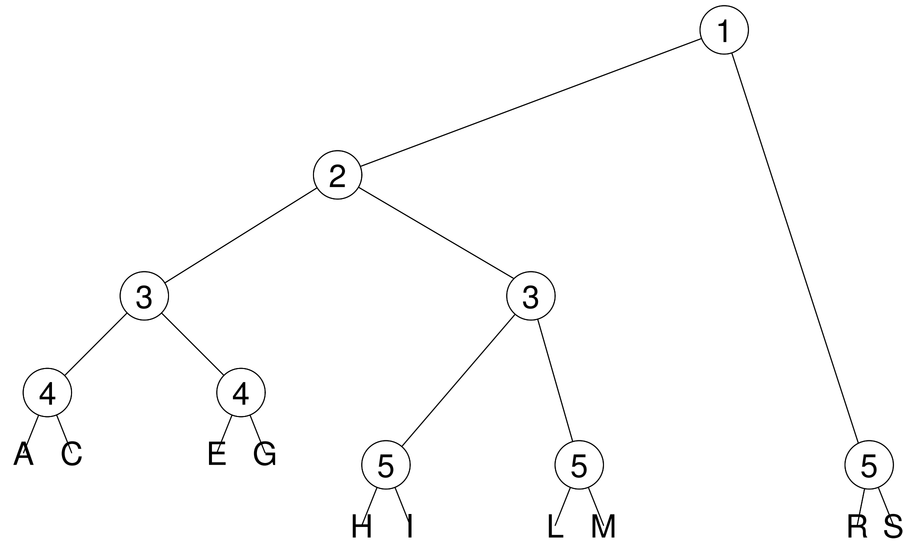
PATRICIA

We want to reduce the space complexity. For example with a genome of $3\ 10^9$ base pairs, this explodes to more than 10^{18} memory locations.



We do not need internal nodes with a single child – path compression.

PATRICIA



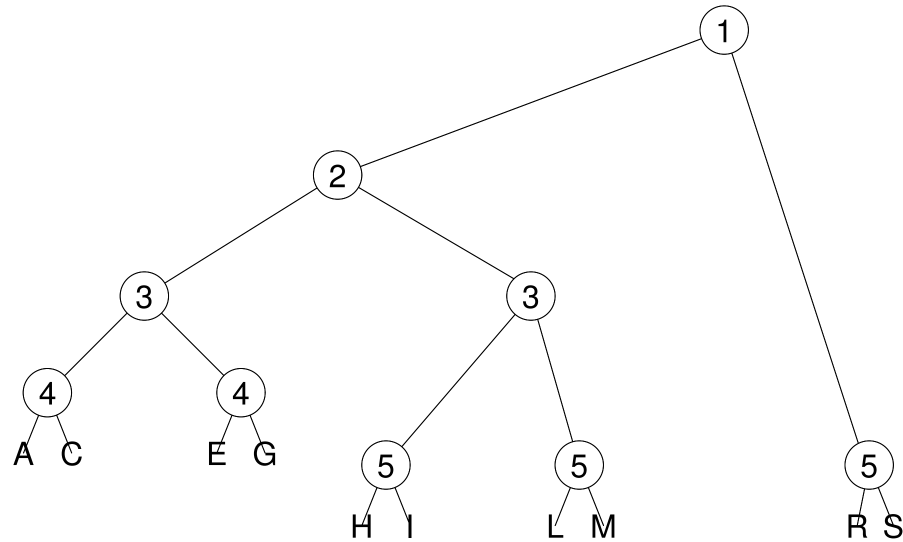
In nodes we have the following information:

1. the index of the next character in p to consider (in figure), or
2. the number of charecters to be skipped in p, or
3. all characters, which were compressed.

In the first two cases we get just candidates we need to verify in the original text.

PATRICIA – example

$(\texttt{00001}, A)$, $(\texttt{00011}, C)$, $(\texttt{00101}, E)$, $(\texttt{00111}, G)$, $(\texttt{01000}, H)$,
$(\texttt{01001}, I)$, $(\texttt{01100}, L)$, $(\texttt{01101}, M)$, $(\texttt{10010}, R)$, $(\texttt{10011}, S)$



`Positions(`I_T`, 101):  R -> candidates` $\{R, S\}$ `--> {}`

PATRICIA – analysis

Preprocessing:

- Insert n strings of T in trie.
- Time complexity $O(n^2)$.
- Space complexity: $O(n + n)$.
- However, there are also associated 4 64-bit references with each internal node, and one reference with a leaf (linked list).
- Additional information if internal nodes store the skipped characters.

Query:

- Follow path from the root – $O(m)$ ...
- ... and than verify candidates – expensive; or
- ... store additional information – $O(m + r)$.

LC-tries

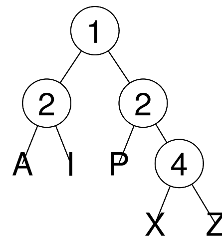
- Arrays are implicit data structures that do not need extra references for accessing its elements. On the other hand the trees require explicit references to access the elements – cf. sorted array and/or suffix array.
- Can we somehow replace and/or introduce arrays into our data structure?

- We replace l levels of trie with an array of size $|\Sigma|^l$ (in our case $|\Sigma| = 2$).

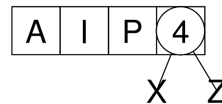
LC-tries – example

Assume we have:

$$(\texttt{00001}, A), (\texttt{01001}, I), (\texttt{10000}, P), (\texttt{11000}, X), (\texttt{11010}, Z)$$



Now we replace two top most layers with an array:



- Define density α and if there are at least $\alpha\,|\Sigma|^l$ in the array, we compress l layers.
- CHALLENGE: how to construct optimal LC-trie?

LC-tries – analysis

Preprocessing:

- Construct PATRICIA and then do level compression.
- Time complexity $O(n^2 + T_l)$, where T_l is time to compress levels.
- Space complexity: $O(n + n)$ and depending on data.

Query:

- At most the same as with PATRICIA – $O(m + r)$.
- Depending on data.

Suffix trees

We want to reduce the preprocessing time complexity. For example with a genome of $3\ 10^9$ base pairs, this explodes to more than 10^{18} – boils down to too long.

- Suffix trees are tries / PATRICIA / LC-tries, where the strings inserted in a trie are suffixes of some text T.
- However, the strings we are inserting into the data structure are not unrelated.
- Consider in

  ```
  0         1         2
  12345678901234567890
  TAACCCTAACCCTAACCCTA
  ```

  strings (t_i, i)

  $$(\texttt{AACCCTAACCCTA\$}, 8)\ (\texttt{ACCCTAACCCTA\$}, 9)$$

  then t_8 can written as $\texttt{A}t_9$.
- This was observed by (Ukkonen 1995) who presented $O(n)$ time construction algorithm.

Suffix trees – analysis

Preprocessing:

- Construct suffix tree using Ukkonen algorithm, perform path and/or level compression.
- Time complexity $O(n + T_l)$, where T_l is time to compress levels.
- Space complexity: $O(n + n)$ and depending on data.

Query:

- At most $O(m + r)$.
- Depending on data.

Suffix tree – RAM model notes

It is not clear if one can retain locality while constructing suffix tree:

- There is a single scan through T, but there is no obvious locality in tree construction.
- The tree can be embedded into a memory to reduce cache faults as much as possible – cache oblivious and/or cache aware data structures.

Ljubljana Algorithms and Data Structures Summer School 2014

Search in unstructured text

Andrej Brodnik
University of Ljubljana

Ljubljana, September 1-5, 2014

Outline

- Trivial solution – sorted array
- Improved solution – suffix array (Manber & Myers 1990)
- Tries (de la Briandais 1959 and Fredkin 1960), PATRICIA (Morrison 1968) and LC-tries (Andersson & Nilsson 1993); Suffix trees or PAT tree (Weiner 1973)

- vEB structure (van Emde Boas 1975)
- Luleå algorithm (Degermark et al. 1997)
- String B trees (Ferragina & Grossi 1998)
- COSD (Brodal & Fagerberg 2006)
- ERA (Mansour et al. 2011)
- Some empirical results (Lipnik 2014, Mesić 2014, Jekovec 2014)

General framework

Quantities:

- We have a text T of length $|T| = n$.
- The length of a query pattern p is $|p| = m$.

- We preprocess the text T in time $T_p(n)$ to produce an index structure I_T.
- The space complexity accounts for T and I_T.
- We ran the query

$$\texttt{Positions(}I_T\texttt{, }p\texttt{) -> } \{x_1, x_2, \ldots, x_r\}$$

in time $T_q(n, m, r)$, which will (hopefully) be just $T_q(m, r)$.

vEB – stratified trees

Given a set of numbers $\mathcal{N}$ from a universe $\mathcal{M}$, we want to perform efficiently:

- Insert x into $\mathcal{N}$.
- Delete x from $\mathcal{N}$.
- Find x in $\mathcal{N}$, that returns $y \in \mathcal{N}$ such that it is the closest to x.

To implement Find we will implement Left and Right efficiently.

We assume $|\mathcal{N}| = n$ and $|\mathcal{M}| = M$.

See also *Open Data Structures* – `http://opendatastructures.org/`.

vEB – stratified trees

General idea:

- Split $\mathcal{M}$ into k buckets (k to be defined later) each with a range k/M.
- For each bucket `b` know (in $O(1)$), which is the smalest `b.min` and which the largest element `b.max`.
- Apply the idea recursivley.

Find the left neighbour of x:

1. Let x belong to the bucket `b`.
2. If $x >$ `b.max`, `b.max` is the left neighbour.
3. If `b.min` $< x \leq$ `b.max` search recursively in bucket `b`.
4. Othervise find the first non-empty bucket `b1` to the left of `b` and then `b1.max` is the left neighbour of x.

How to implement efficiently the last step?

NOTE: k is not defined yet and the last step is not implemented!

vEB – the last step

1. Let x belong to the bucket `b`.
2. If $x >$ `b.max`, `b.max` is the left neighbour.
3. If `b.min` $< x \leq$ `b.max` search recursively in bucket `b`.
4. Othervise find the first non-empty bucket `b1` to the left of `b` and then `b1.max` is the left neighbour of x.

- Search for the first non-empty bucket to the left is the same as search for the left element, only that the elements are now non-empty buckets!! Therefore use the same data structure – TOP.
- Since now we have the same (recursive) call as in step 3 of the algorithm, the number of buckets k should be the same as the range of one bucket. Therefore, $k = \sqrt{M}$.

vEB – time analysis

$$
\begin{aligned}
T(M) &= T(\sqrt{M}) + c \\
&= T(M^{\frac{1}{2}}) + c \\
&= T(M^{\frac{1}{2^2}}) + 2c \\
&\cdots \\
&= T(M^{\frac{1}{2^s}}) + sc = O(s)
\end{aligned}
$$

Let $M^{\frac{1}{2^s}} = 2$ and consequently

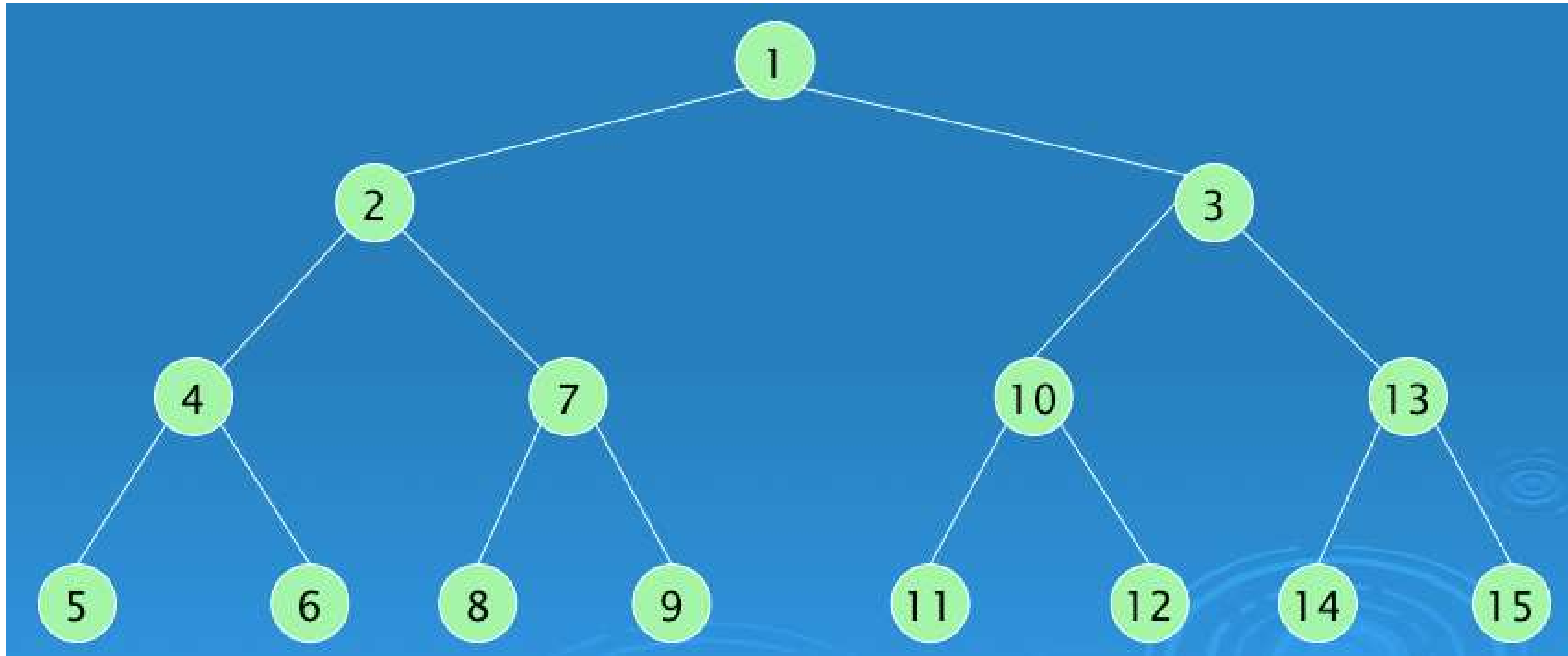
$$
\begin{aligned}
\frac{1}{2^s} \lg M &= 1 \\
\lg M &= 2^s \\
s &= \lg lgM
\end{aligned}
$$

vEB – space analysis

Similar recursion, and $O(M)$.

However, if we do not store buckets in an array, but in hash table, $O(n)$ (Willard 1983).

vEB – layout



Authors: Gerth Brodal, Rolf Fagerberg, Riko Jacob

Static vEB

Find the left neighbour of x:

1. Let x belong to the bucket $\texttt{b}$.
2. If $x > \texttt{b.max}$, $\texttt{b.max}$ is the left neighbour.
3. If $\texttt{b.min} < x \leq \texttt{b.max}$ search recursively in bucket $\texttt{b}$.
4. Othervise find the first non-empty bucket $\texttt{b1}$ to the left of $\texttt{b}$ and then $\texttt{b1.max}$ is the left neighbour of x.

How to implement efficiently the last step?

Static vEB

- We do not need TOP if most of the buckets are full – use just a precomputed reference to the left non-empty bucket. Consequently, $O(1)$ for this step.
- Now set $k = n$ – i.e. on average every bucket has one element. In particularly good with well spread data.
- Space use $O(dn)$, where d is depth – expected $O(1)$. Consequently, also time complexity $O(d) = O(1)$.
- Needs more detail analysis (cf. Brodnik & Munro 1999).

vEB and our problem

How is vEB related to search of positions in a text?

- Note, structure of vEB is identical to trie.
- Every character string can we treated as a (very long) number.
- We store such a number in vEB with a data that is position in the text.
- The search for a patern p is the same as:
  - expand p appropriately to $p.m$ and $p.M$;
  - search from an exapanded $p.m$ to the left and from $p.M$ to the right

  The prefix represents a node in vEB – the root of subtree we are looking for.

Luleå algorithm

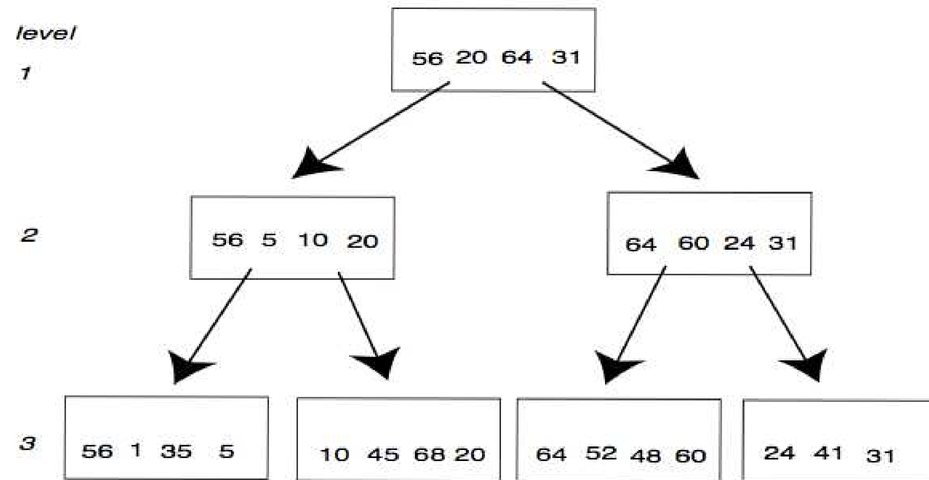
- A static vEB used for IP packet next hop calculation from a destination address.
- Buckets with a small number of elements are represented by a (sorted) list of contained elements.
- Depth for a routing table $d = 3$.
- Consequently d access to the memory.
- In the case of routing table it is small enough to fit into L2 cache.
- What about big unstructured text?
- Construction (preprocessing) of suffixes and not to hit $O(n^2)$ barrier?

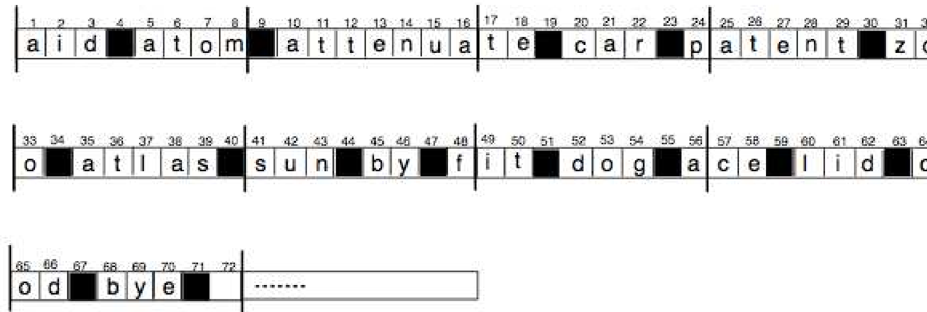
B tree of strings

B-tree:

- Tree, where internal nodes have between B and $B/2$ children.
- All leaves at the same depth – balanced.
- Consequently, height of tree $O(\log_B n)$.
- Cache aware as B can be tailored to size of B :-)
- However, B is defined by the system, while B in B-trees is calculated from the former; hence we want to have it as big as possible.
- B$^+$ trees have elements only at leaves, while internal nodes are just used as disciminators which makes B from B-tree as big as possible.
- Have a (sorted) set of words that are represented by the references and use in a B tree references.

B tree of strings

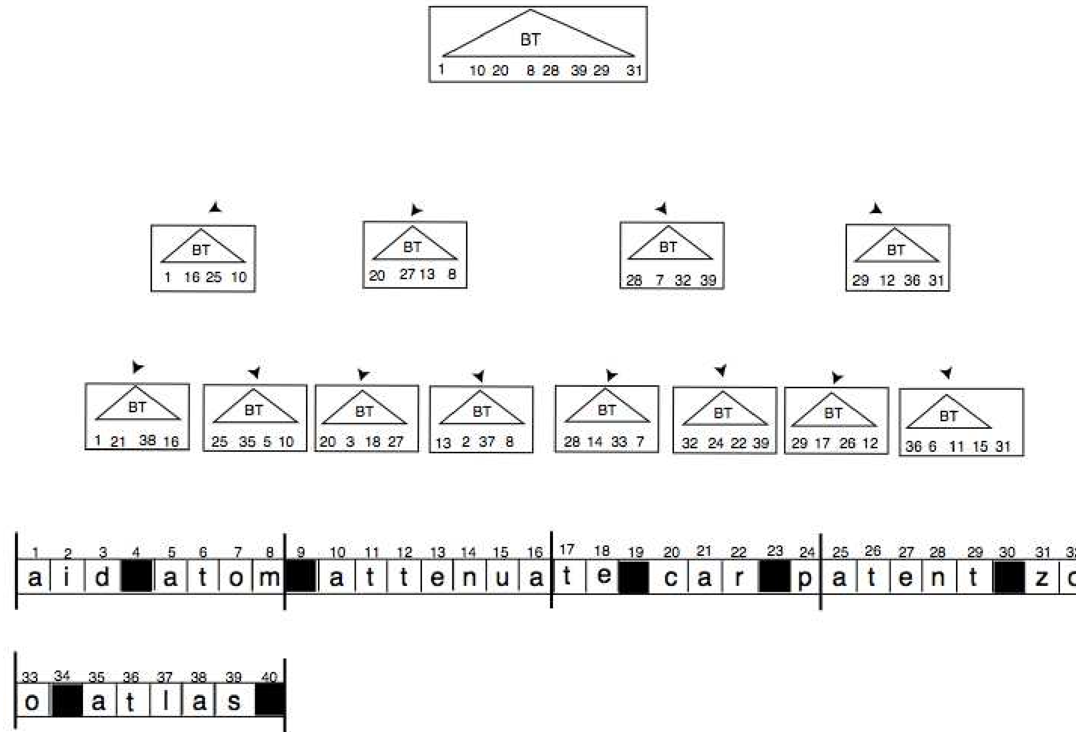




Authors: Paolo Ferragina and Roberto Grossi.

String B tree

Replace internal nodes by PATRICIA and also insert all suffixes:



Authors: Paolo Ferragina and Roberto Grossi.

String B tree – analysis

Preprocessing:

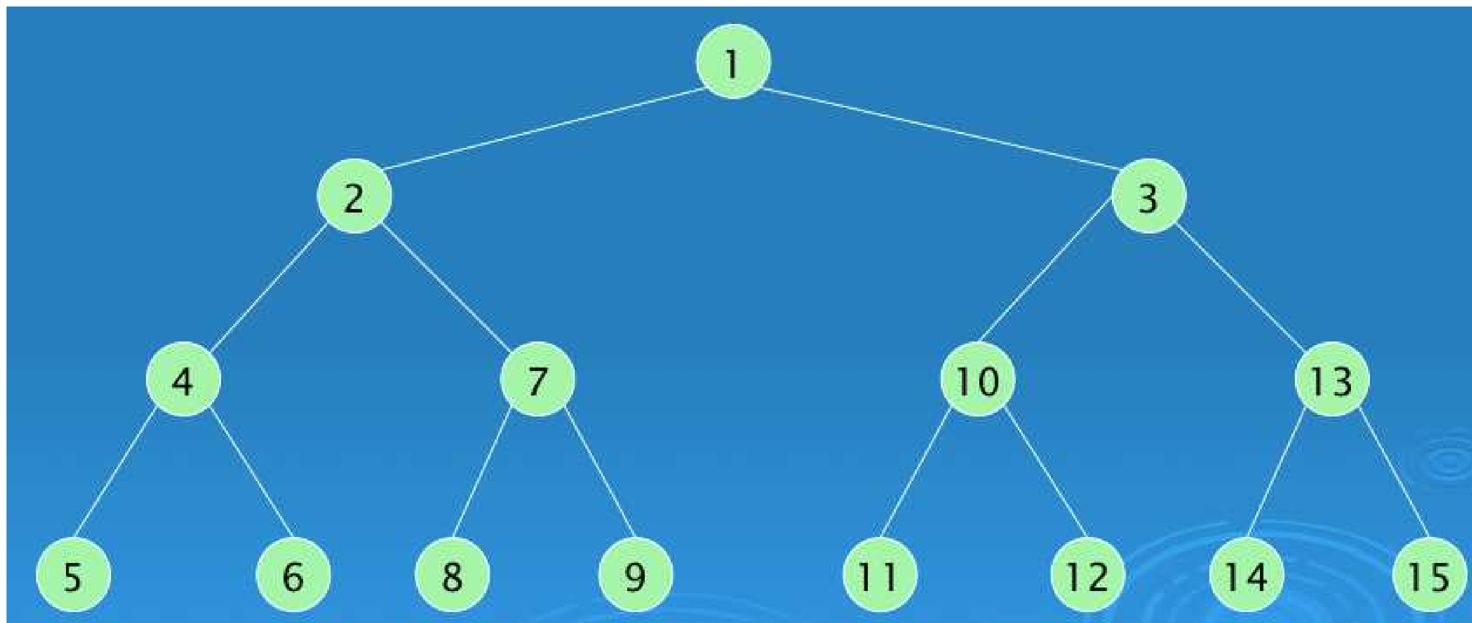
- Insert n strings of T in string B tree.
- Time complexity $O(n^2)$.
- Space complexity: $O(n + n)$.

Query:

- Traverse from the top to the leaves $O(d) = O(\log_B n)$.

String B trees anf RAM

- Cache aware through B.
- Cache oblivious B-trees (Bender, Demaine & Farach-Colton 2000) – arrange nodes in a tree in vEB layout.



Authors: Gerth Brodal, Rolf Fagerberg, Riko Jacob

- Cache oblivious String B-trees (Bender, Farach-Colton & Kuszmaul 2006).

COSD – Cache-Oblivious String Dictionaries

- Start with a trie and make it cache oblivious.

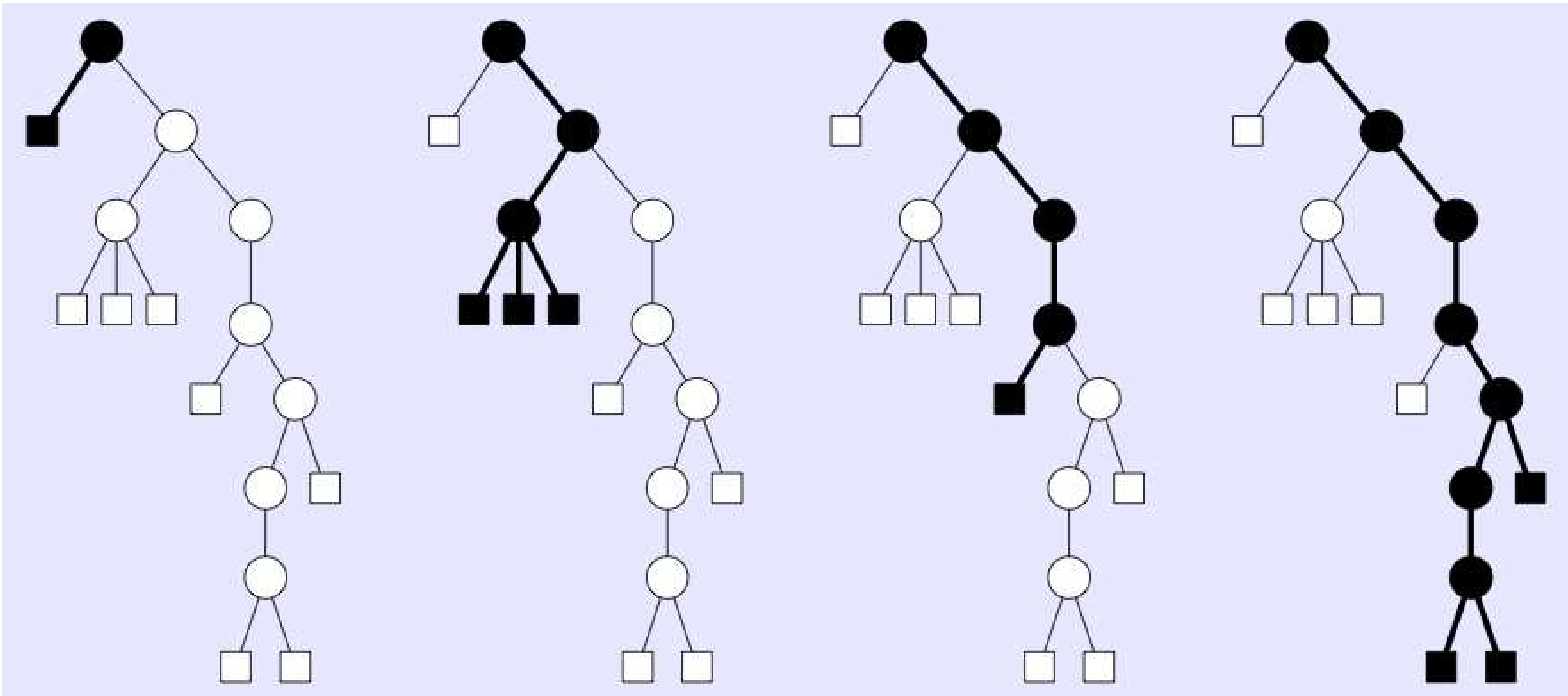
We know how to:

- Scanning $O(n/B)$.
- B-tree searching $O(\log_B n)$,
- Sorting $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$.

How to use this on tries? $\Rightarrow$ Transform them.

COSD – The Idea

- Make a trie where the “neck” is long and requires scanning and pay for the search in a “body” (cf. giraffe).
- Glue together giraffes using B-tree like structures.



Authors: Gerth Brodal, Rolf Fagerberg

COSD – analysis

Preprocessing:

- Insert all suffixes into a COSD: $O(n^2)$.
- Size $O(n)$.

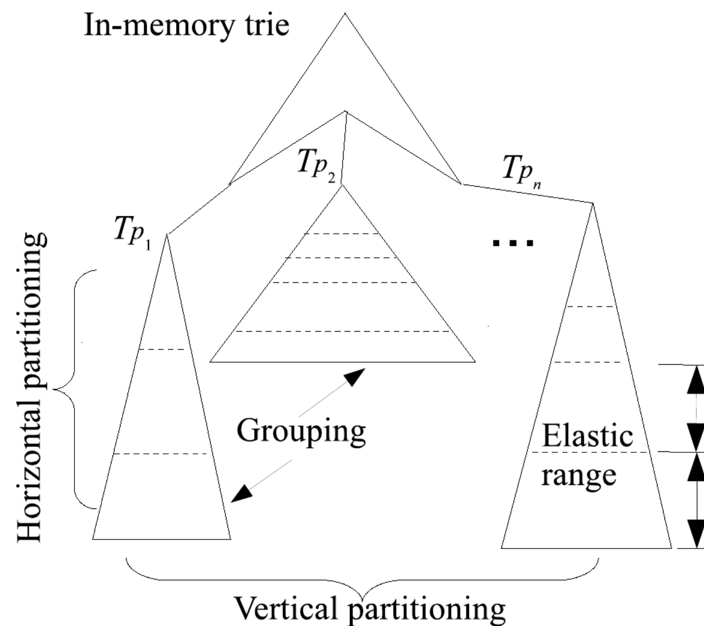
Query:

- From B-trees $O(\log_B n)$.

ERa – Elastic Range

- "Real world" need to construct suffix tree.
- Thereofre use real hardware, which is multicore, and ...
- ... it has to be done in our "life-time":-)

Carefully designed construction algorithm that is very much aware of real system it is computed on.



Measurements – query times

(Jekovec 2014, Lipnik 2014, Mesić 2014)

- $T = G[10^7 \ldots 10^7 + 12,000)$
- Query sets $Q_1, Q_2.Q_3, Q_4$:
  - Sizes 5,000, 10,000, 15,000 and 20,000 patterns p
  - Q_i contains half patterns in T and randomly interleaved
  - $|p|$ is random in range $[4, |T|)$

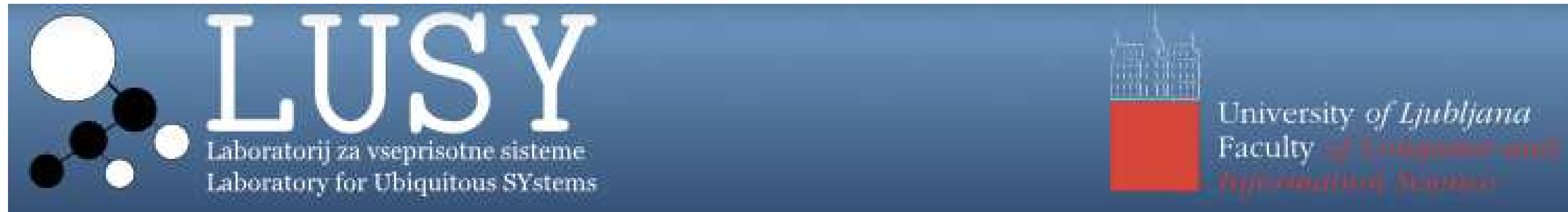
število poizvedb	ERA	COSD	B-drevo nizov	
			v1	v2
5.000	0.06	0.30	0.04	0.05
10.000	0.12	0.83	0.08	0.10
15.000	0.19	0.88	0.12	0.14
20.000	0.23	1.24	0.16	0.18

Measurements – memory accesses

(Jekovec 2014, Lipnik 2014, Mesić 2014)

Query set Q_1.

	ERA	COSD	B-drevo nizov	
			v1	v2
Ir	50.067	62.692	58.997	59.062
I1mr	2	0	0	10
ILmr	0	0	0	0
Dr	12.693	12.016	21.448	21.429
D1mr	130	**1.913**	131	148
DLmr	59	**1.886**	6	5
Dw	45	144	204	214
D1mw	0	0	3	4
DLmw	0	0	0	0
LL	132	**1.914**	134	161



Thanx for your attention!

URL: lads14.fri.uni-lj.si
URL: lusy.fri.uni-lj.si
e-mail: andrej.brodnik@fri.uni-lj.si