Space-Efficient Data Structures for Strings and Sequences

Roberto Grossi

Università di Pisa, Italy

Joint works with

P. Ferragina, A. Gupta, G. Ottaviano,

K. Sadakane, R. Shah, J.S. Vitter, B. Xu

Why string data?

Ubiquitous and massive data:

- Digital libraries and product catalogues
- Electronic white and yellow pages
- Specialized information sources (e.g., genome or patents)
- Web pages repositories
- XML format for structured data
- Newsgroups
- ...

What is a String Dictionary?

- Persistent and space-efficient data structure.
- Quickly answer to lots of seach queries.
- Accessing a small portion of the string collection.

- Basic building block of any IR system
  (along with modeling, ranking, query languages, security and access control)

  For example...

Text indexing (TI): **world-level** vs full-text

"This rare goods [the information]

will be prepared under malleable or eatable form,

will be delivered to more and more customers;

it will be sold, exported, duplicated and reproduced

everywhere…"

Paul Valéry, Poet, 1871-1945

Text indexing (TI): world-level vs **full-text**

“This rare goods [the information]

will be prepared under malleable or eatable form,

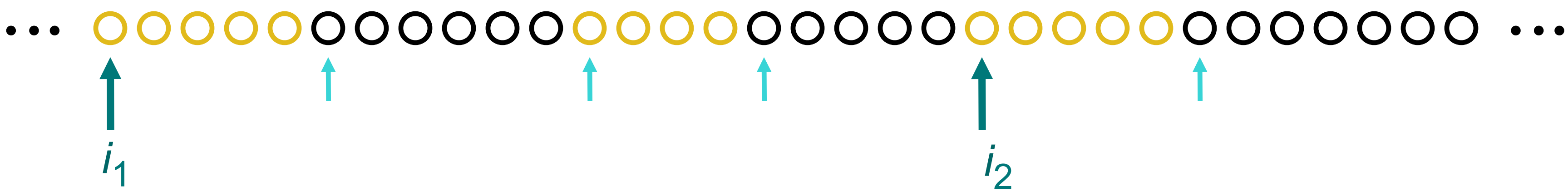
will be delivered to more and more customers;

it will be sold, exported, duplicated and reproduced

everywhere…”

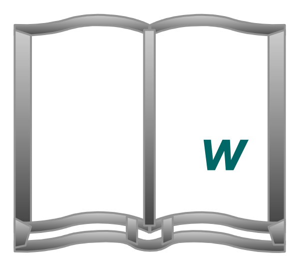
Paul Valéry, Poet, 1871-1945

Word-level TI: Inverted lists/files



- Split the text into words

- Collect all distinct words in a string dictionary:
  for each word *w*, store the
  (inverted or position) list
  of its locations in the text



i_1, i_2, …

- Support Boolean/ranked queries

Search “will” + “and”

“This rare goods [the information]

will be prepared under malleable or eatable form,

will be delivered to more and more customers;

it will be sold, exported, duplicated and reproduced

everywhere, and …”

Paul Valéry, Poet, 1871-1945

Basic property of full-text indexing: **Prefix searching** in a string dictionary

Pattern P occurs at position *i* of text T
iff
P is prefix of a suffix of T

`mississippi`

Basic property of full-text indexing: **Prefix searching** in a string dictionary

Pattern P occurs at position *i* of text T

iff

P is prefix of a suffix of T

`issippi`

Example of string dictionaries
supporting prefix search,
you can make succinct
using the techniques seen in previous talks...

Trie T for a set S of strings

1. S empty → T empty
2. $S = S_A \cup S_B \ldots \cup S_Z$ (partition):

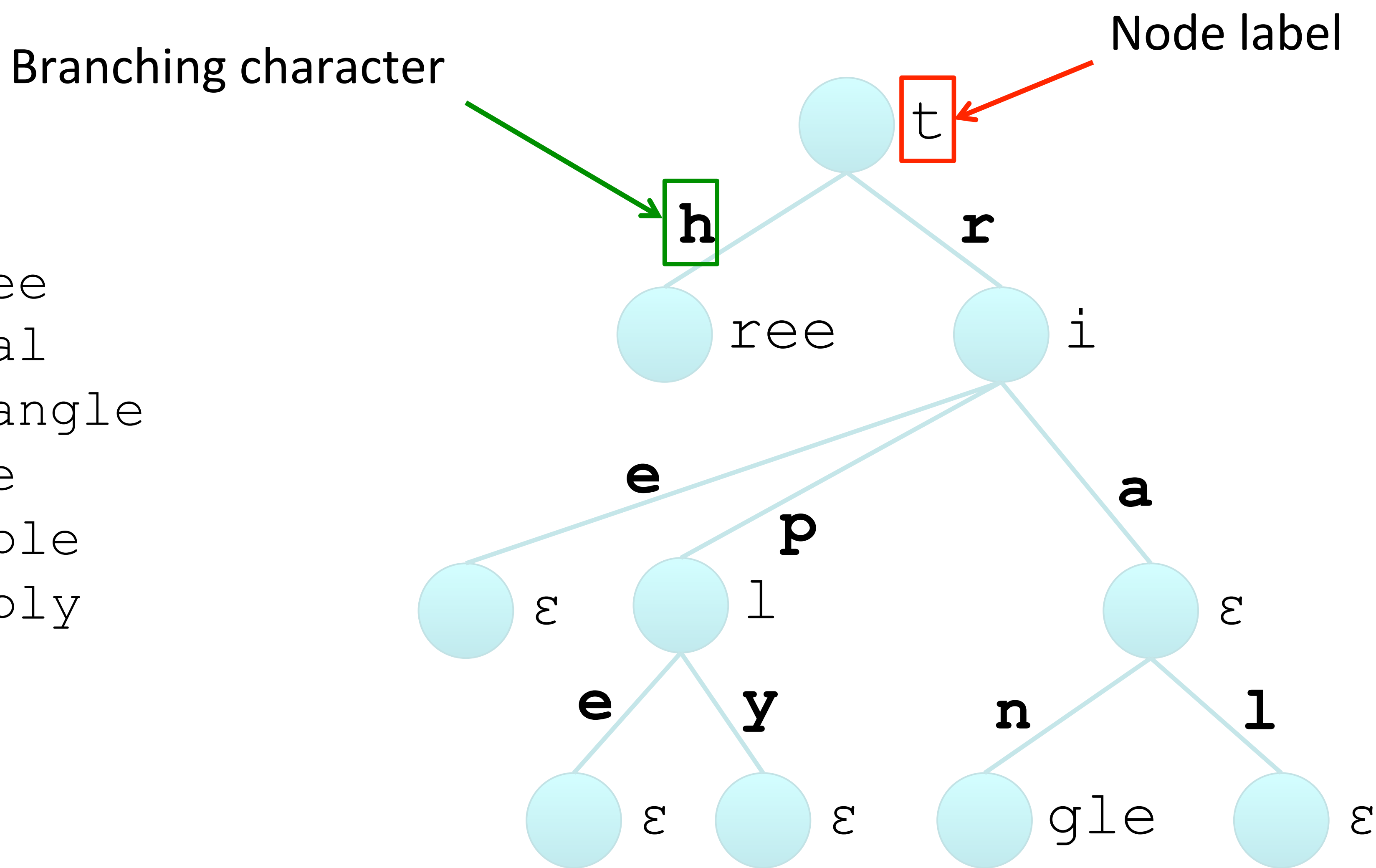
   T is the root with children

   $T_A, T_B, \ldots, T_Z$

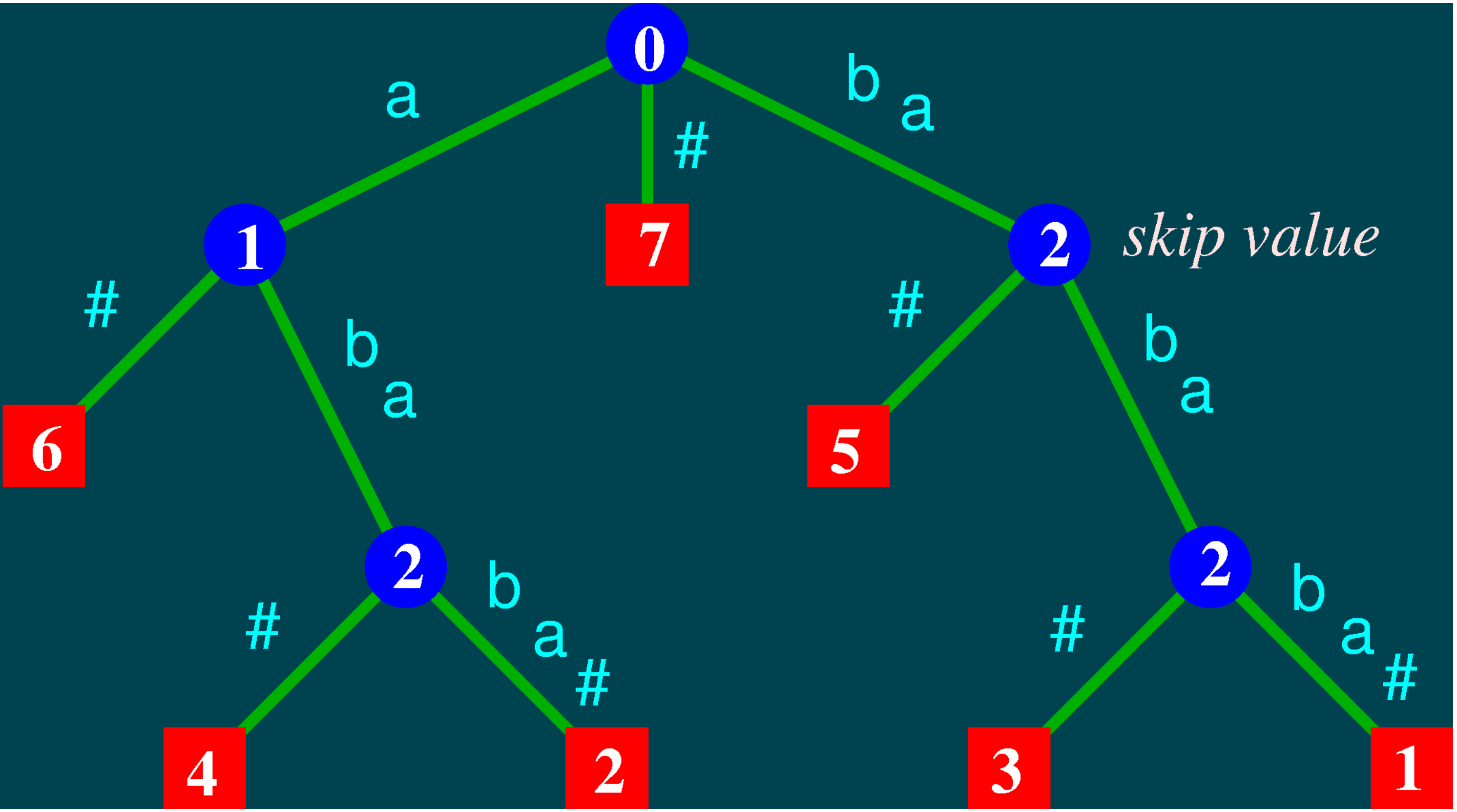
   with T_x = trie(S_x) with initial *x* removed

Compacted tries

three
trial
triangle
trie
triple
triply

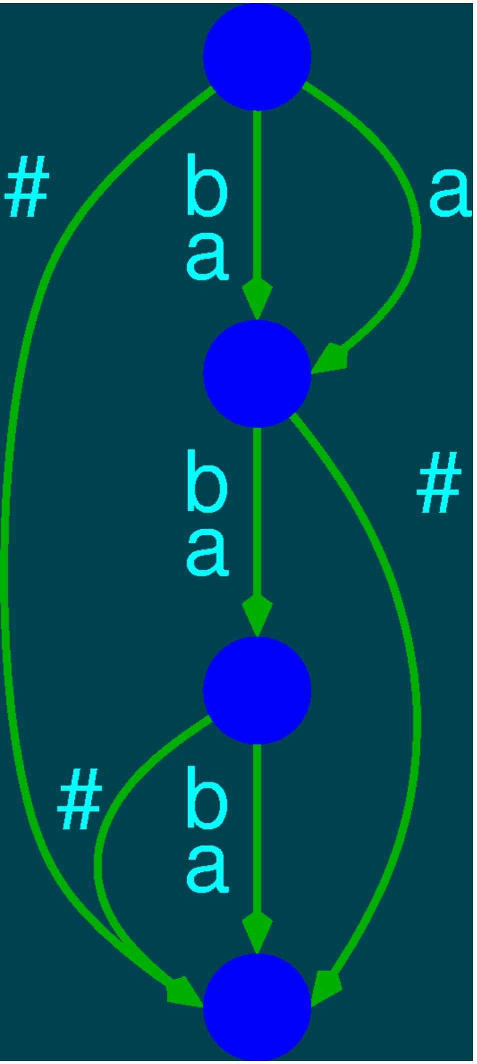


Full text index: Suffix tree



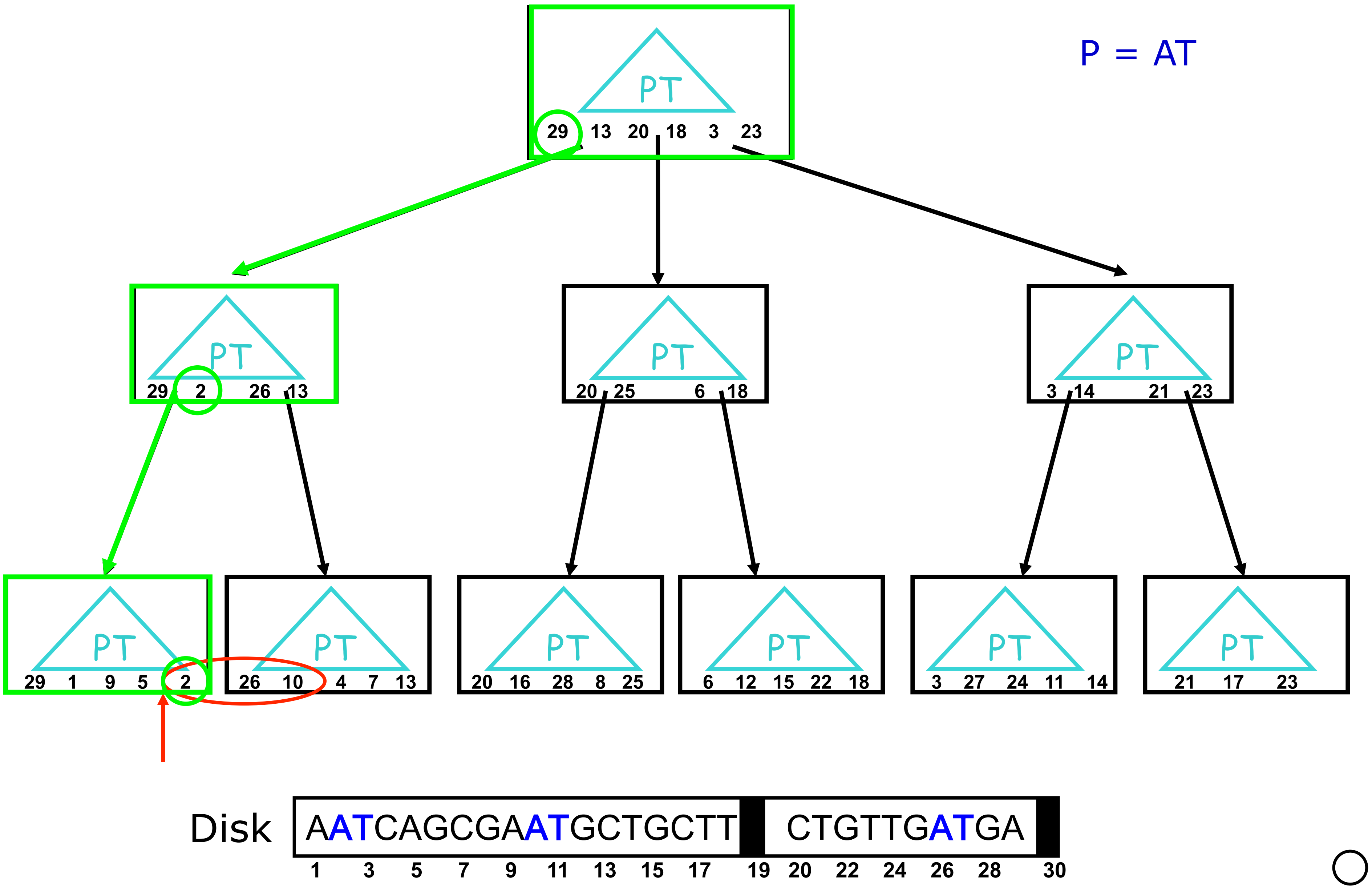
- Compact trie / Patricia storing the suffixes of bababa#
- Space is roughly 16*n* bytes [Manber,Myers]
- Online construction in linear time [Ukkonen]

Full text index: Automaton and DAWG



- Collapse isomorphic nodes to reduce their number
- Average size is about 1.3 nodes per character
- Nodes are of larger size
- Number of bytes does not reduce much

String B-trees



SA suffix array: `mississippi`

1	mississippi
2	ississippi
3	ssissippi
4	sissippi
5	issippi
6	ssippi
7	sippi
8	ippi
9	ppi
10	pi
11	i

11	i
8	ippi
5	issippi
2	ississippi
1	mississippi
10	pi
9	ppi
7	sippi
4	sissippi
6	ssippi
3	ssissippi

Let us focus on suffix arrays...

text T = S E N S E L E S S N E S S _

1	2	3	4	5	6	7	8	9	10	11	12	13	14
S	E	N	S	E	L	E	S	S	N	E	S	S	_

i	SA[i]	T[SA[i] ... n]
1	14	_
2	5	E L E S S N E S S _
3	2	E N S E L E S S N E S S _
4	11	E S S _
5	7	E S S N E S S _
6	6	L E S S N E S S _
7	10	N E S S _
8	3	N S E L E S S N E S S _
9	13	S _
10	4	S E L E S S N E S S _
11	1	S E N S E L E S S N E S S _
12	9	S N E S S _
13	12	S S _
14	8	S S N E S S _

Storing the SA permutation...

i	SA[i]	Φ(i)
1	14	0/11
2	5	6
3	2	8
4	11	13
5	7	14
6	6	5
7	10	4
8	3	10
9	13	1
10	4	2
11	1	3
12	9	7
13	12	9
14	8	12

"suffix link" for SA:
Φ **function** [G.&Vitter '00]

Φ(i) = k iff SA[k] = SA[i] + 1

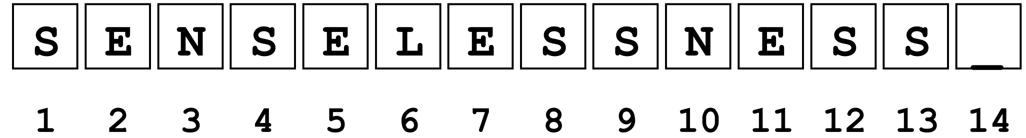
rank of the next suffix in T

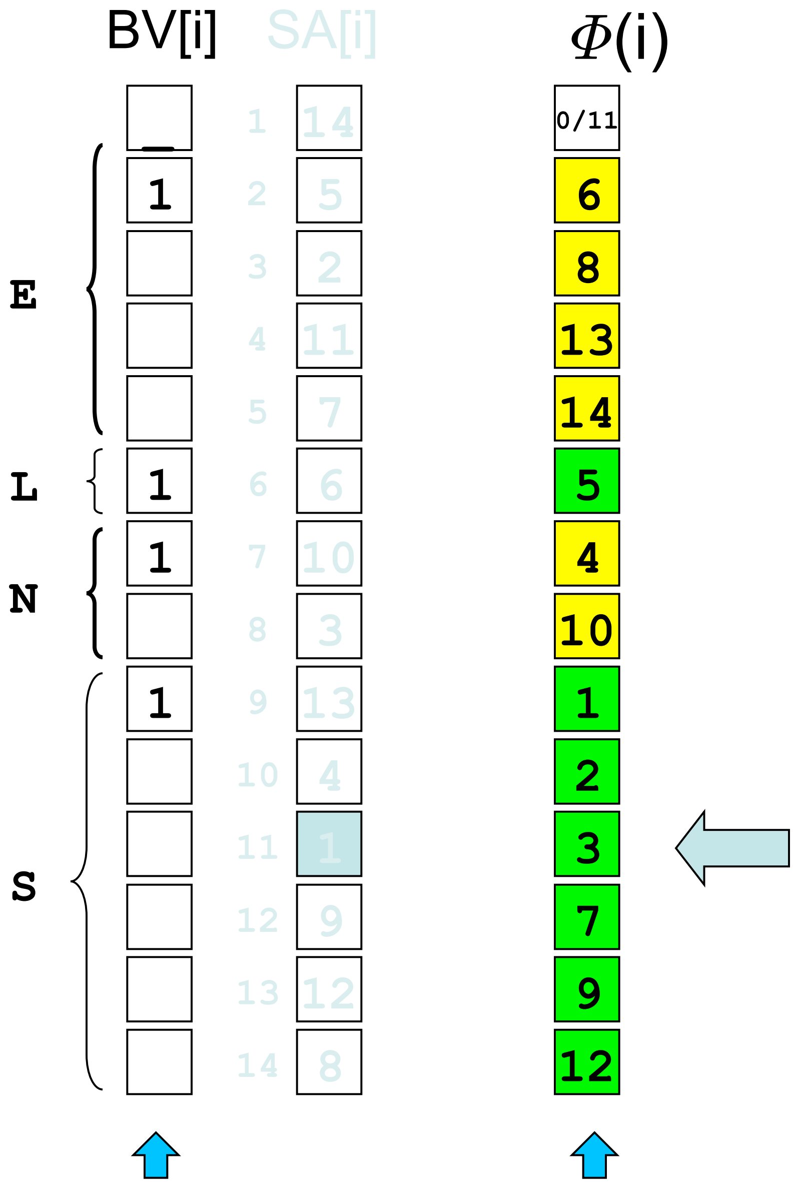
Φ(i)=0 when SA[i]=n or
Φ(i)=z where SA[z]=1

S	E	N	S	E	L	E	S	S	N	E	S	S	_
1	2	3	4	5	6	7	8	9	10	11	12	13	14

	BV[i]		SA[i]	T[SA[i]]	Φ(i)
	_	1	14	_	0/11
E	1	2	5	E	6
		3	2	E	8
		4	11	E	13
		5	7	E	14
L	1	6	6	L	5
N	1	7	10	N	4
		8	3	N	10
S	1	9	13	S	1
		10	4	S	2
		11	1	S	3
		12	9	S	7
		13	12	S	9
		14	8	S	12

Using Φ and a bitvector BV we can avoid storing SA





Using Φ and a bitvector BV we can avoid storing SA

S	E	N	S	E	L	E	S	S	N	E	S	S	_
1	2	3	4	5	6	7	8	9	10	11	12	13	14

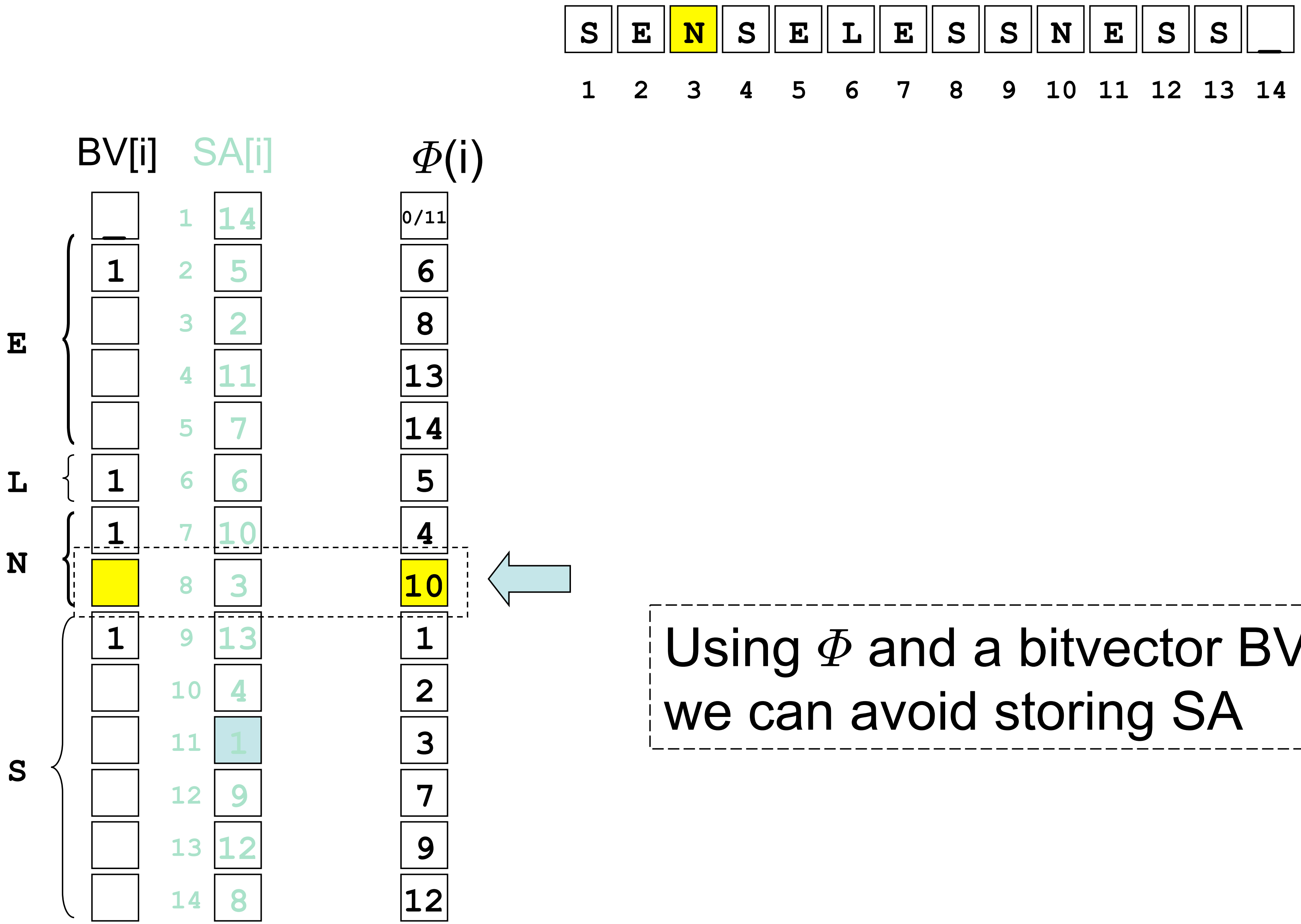
	BV[i]	i	SA[i]	Φ(i)
	_	1	14	0/11
E	1	2	5	6
		3	2	8
		4	11	13
		5	7	14
L	1	6	6	5
N	1	7	10	4
		8	3	10
S	1	9	13	1
		10	4	2
		11	1	3
		12	9	7
		13	12	9
		14	8	12

Using Φ and a bitvector BV we can avoid storing SA

S	E	N	S	E	L	E	S	S	N	E	S	S	_
1	2	3	4	5	6	7	8	9	10	11	12	13	14

	BV[i]		SA[i]	Φ(i)
	_	1	14	0/11
	1	2	5	6
E		3	2	8
		4	11	13
		5	7	14
L	1	6	6	5
N	1	7	10	4
		8	3	10
	1	9	13	1
		10	4	2
		11	1	3
S		12	9	7
		13	12	9
		14	8	12

Using Φ and a bitvector BV we can avoid storing SA



Using Φ and a bitvector BV we can avoid storing SA

S	E	N	S	E	L	E	S	S	N	E	S	S	_
1	2	3	4	5	6	7	8	9	10	11	12	13	14

	BV[i]	i	SA[i]	Φ(i)
	_	1	14	0/11
E	1	2	5	6
		3	2	8
		4	11	13
		5	7	14
L	1	6	6	5
N	1	7	10	4
		8	3	10
S	1	9	13	1
		10	4	2
		11	1	3
		12	9	7
		13	12	9
		14	8	12

Using Φ and a bitvector BV we can avoid storing SA

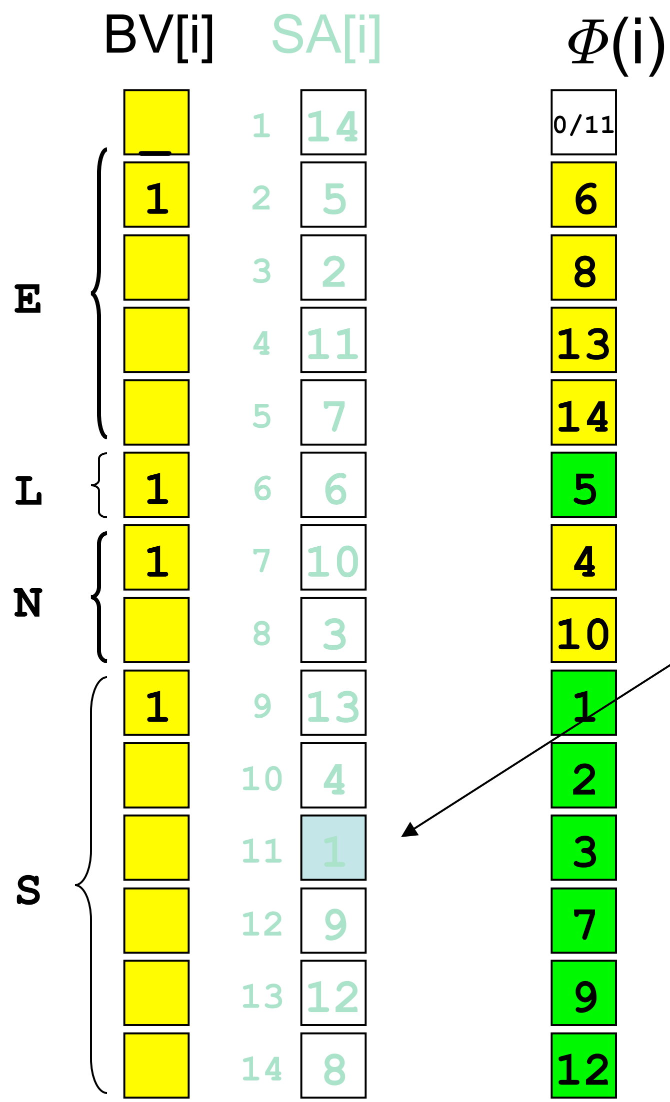
1	2	3	4	5	6	7	8	9	10	11	12	13	14
S	E	N	S	E	L	E	S	S	N	E	S	S	_

	BV[i]	i	SA[i]	Φ(i)
_		1	14	0/11
E	1	2	5	6
E		3	2	8
E		4	11	13
E		5	7	14
L	1	6	6	5
N	1	7	10	4
N		8	3	10
S	1	9	13	1
S		10	4	2
S		11	1	3
S		12	9	7
S		13	12	9
S		14	8	12

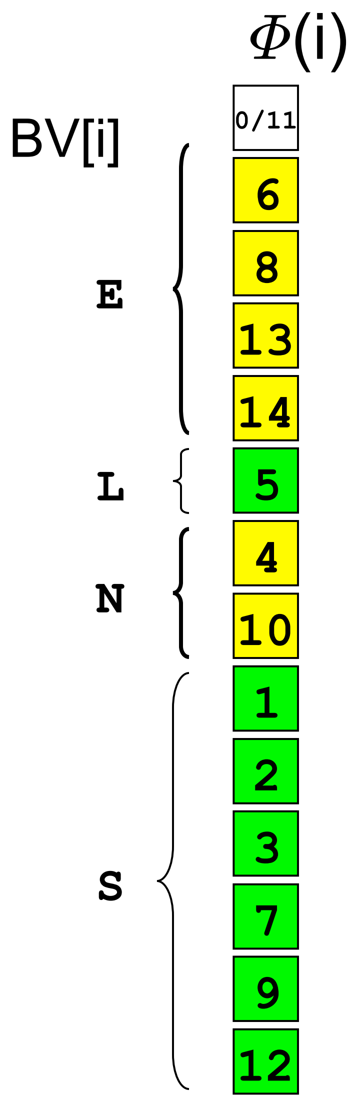
Using Φ and a bitvector BV we can avoid storing SA

S	E	N	S	E	L	E	S	S	N	E	S	S	_
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Replacing SA by Φ and BV



- Record which SA[j] = 1
- T[i] = c iff $\Phi^{(i)}$(j) $\in$ Interval$_c$



This is what we need to store:

- bitvector BV
- Φ function

Also Φ is represented by bitvectors!

Exploit Φ's properties

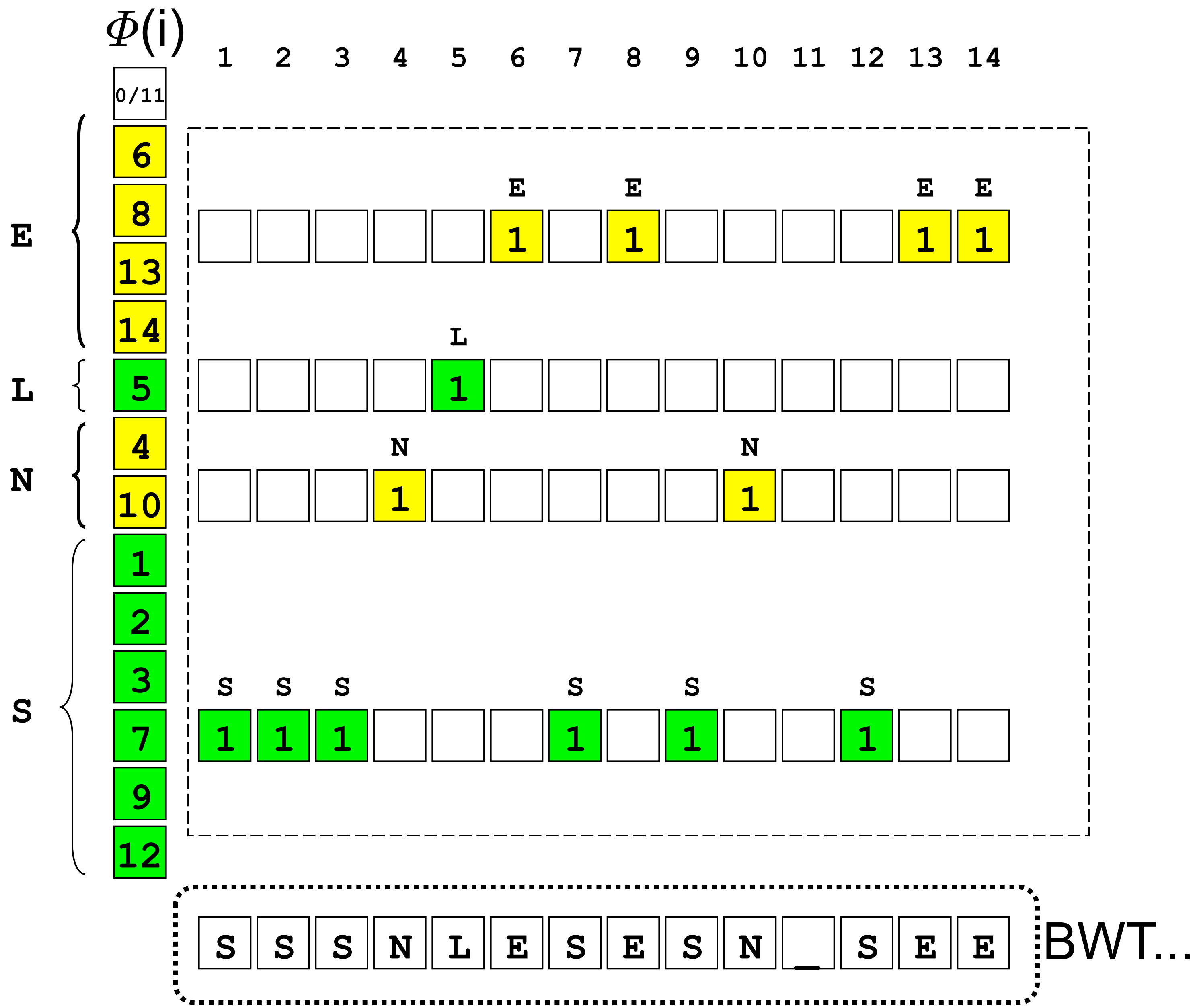
- Observation made on SA perms:

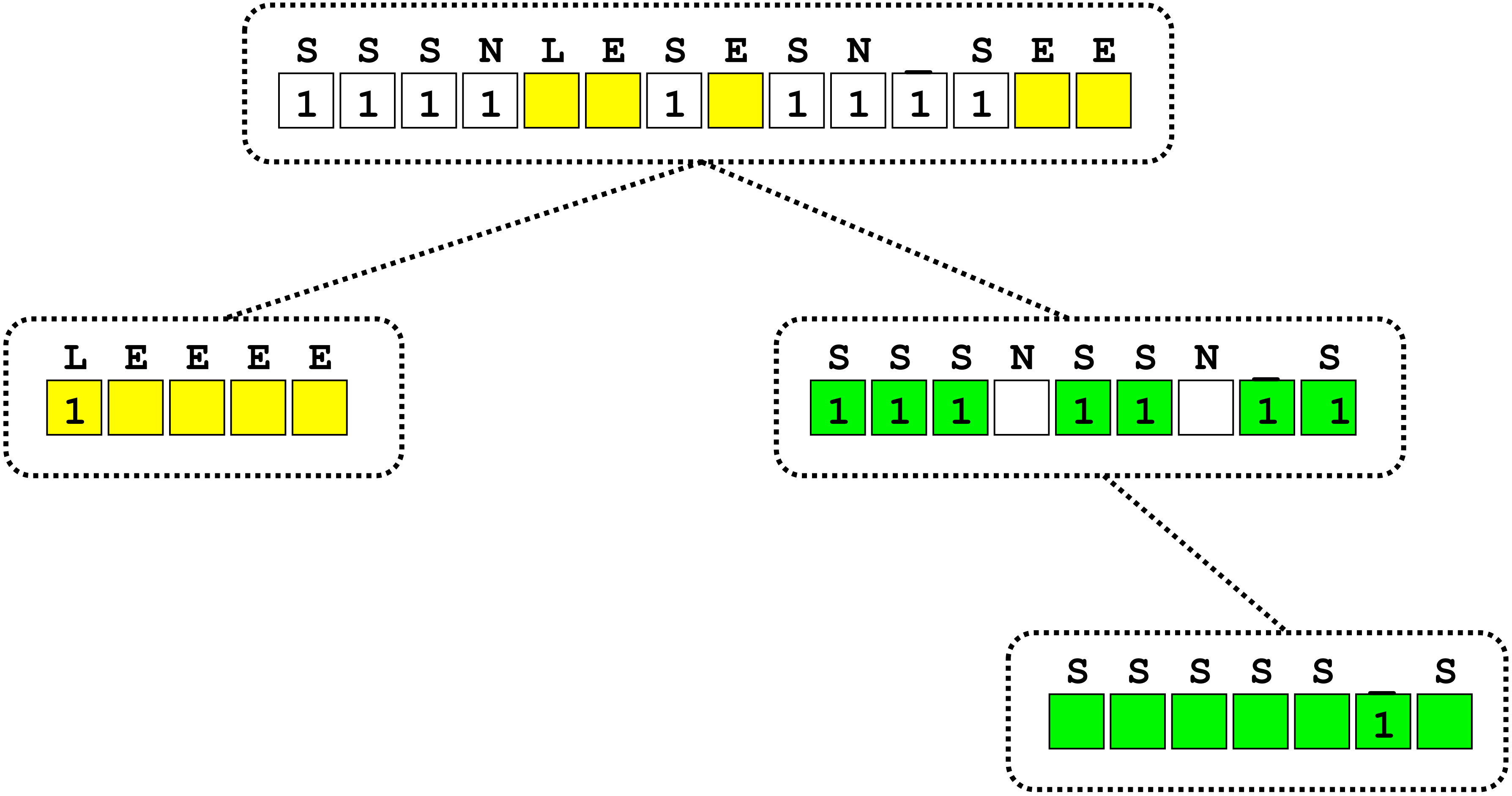
  T[SA[i]] = T[SA[i+1]] $\Rightarrow \Phi$(i) < Φ(i+1)

- Monotonicity: a maximal run of equal symbols

  T[SA[i]] = ⋯ = T[SA[j]] $\Rightarrow$ 1 $\leq \Phi$(i) < ⋯ < Φ(j) $\leq$ n

- At most σ runs, called Σ lists


Φ(i)
1 2 3 4 5 6 7 8 9 10 11 12 13 14
0/11
6
8
13
14
5
4
10
1
2
3
7
9
12
E
L
N
S
E E E E
1 1 1 1
L
1
N N
1 1
S S S S S S
1 1 1 1 1 1
S S S N L E S E S N _ S E E
BWT...



Wavelet tree on BWT

Entering compression stuff...

A first simple approach

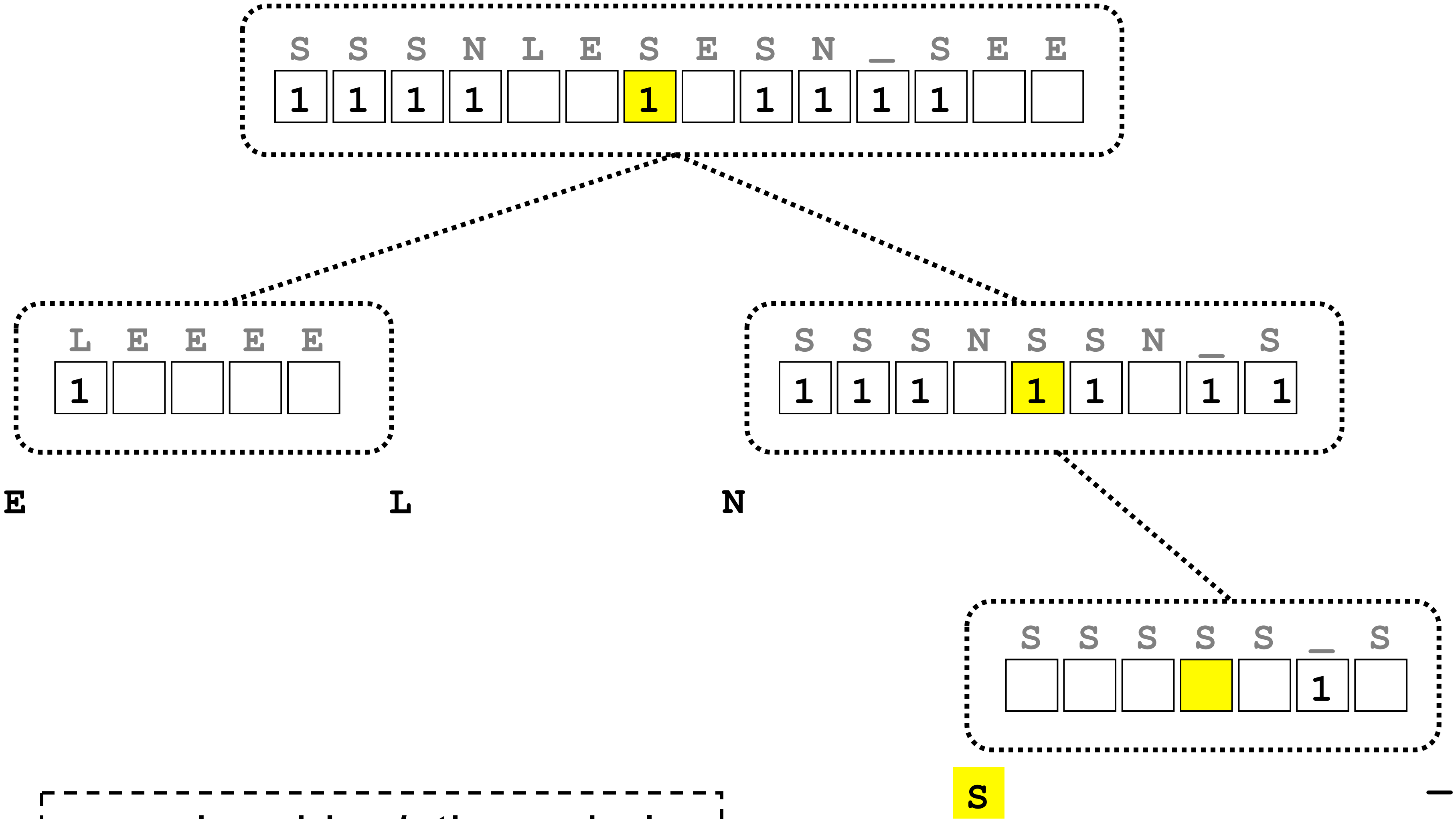
- Encode bitvector BV in B(σ,n) + o(n) bits
- Encode Φ function in B(n,nσ) + o(n log σ) bits

Using Wavelet Trees, compressed representation in

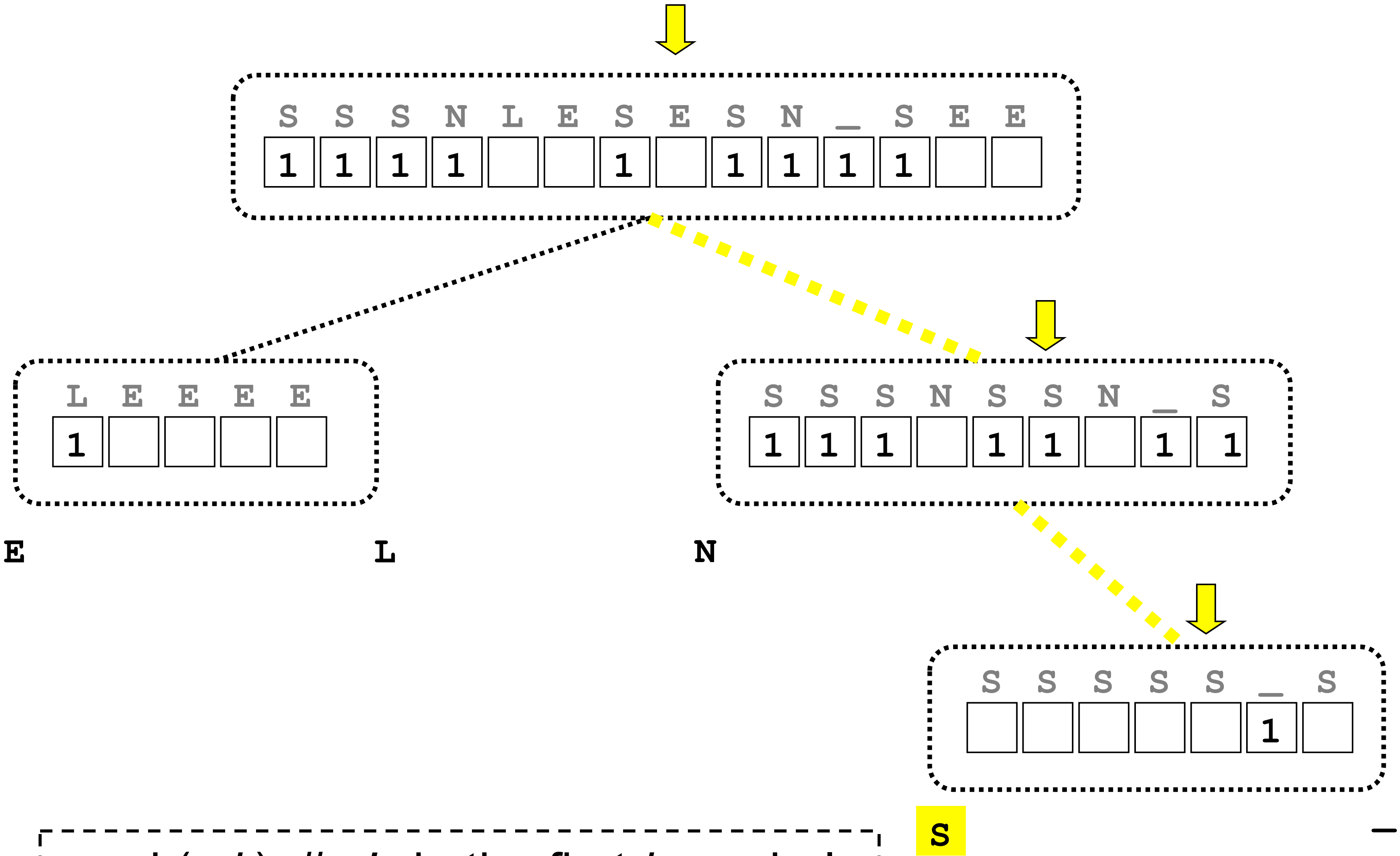
$\boxed{\text{n log } \sigma \text{ + lower order bits}}$

becomes high-order entropy **n H_k** by a refined analysis

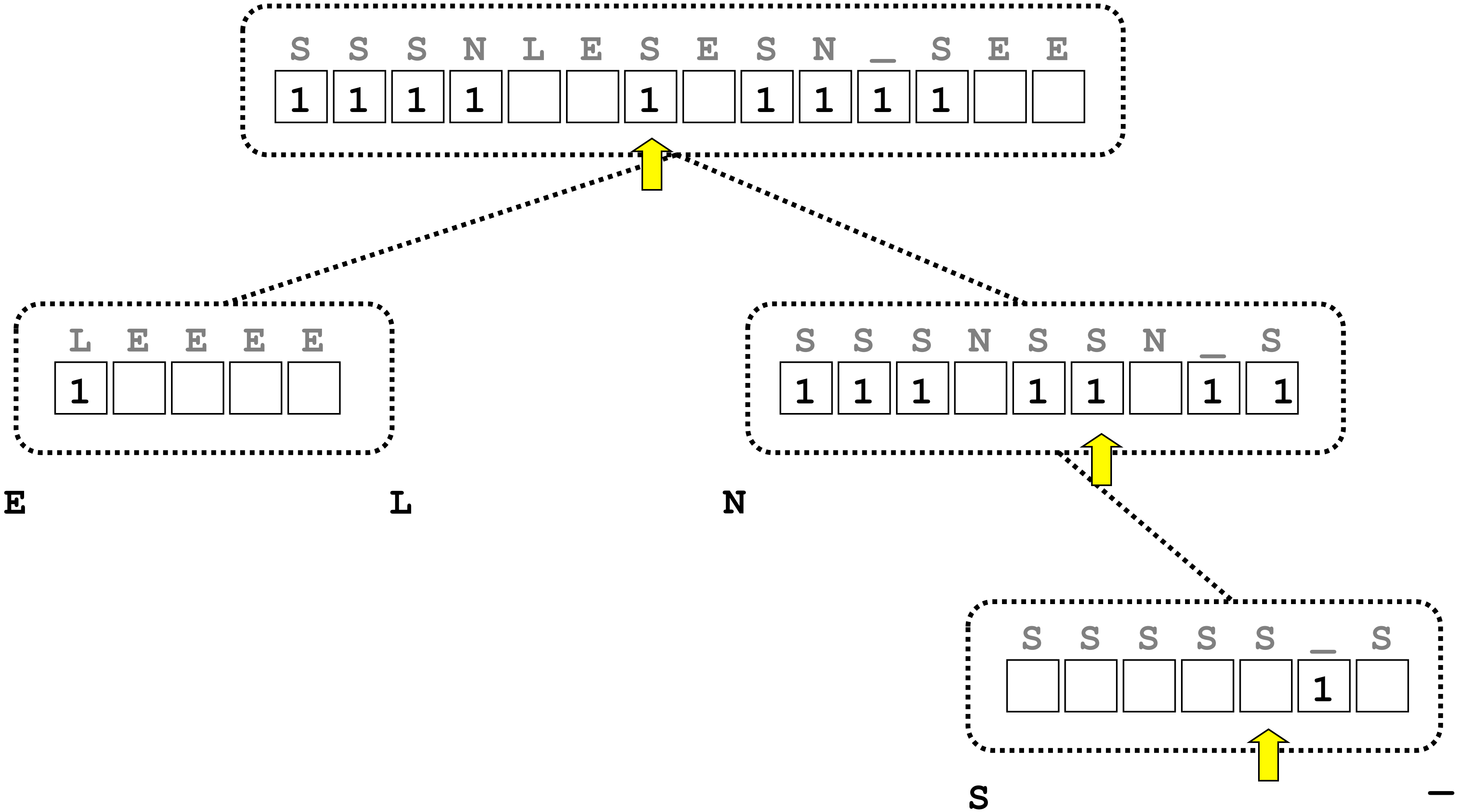
Let us focus on wavelet trees...



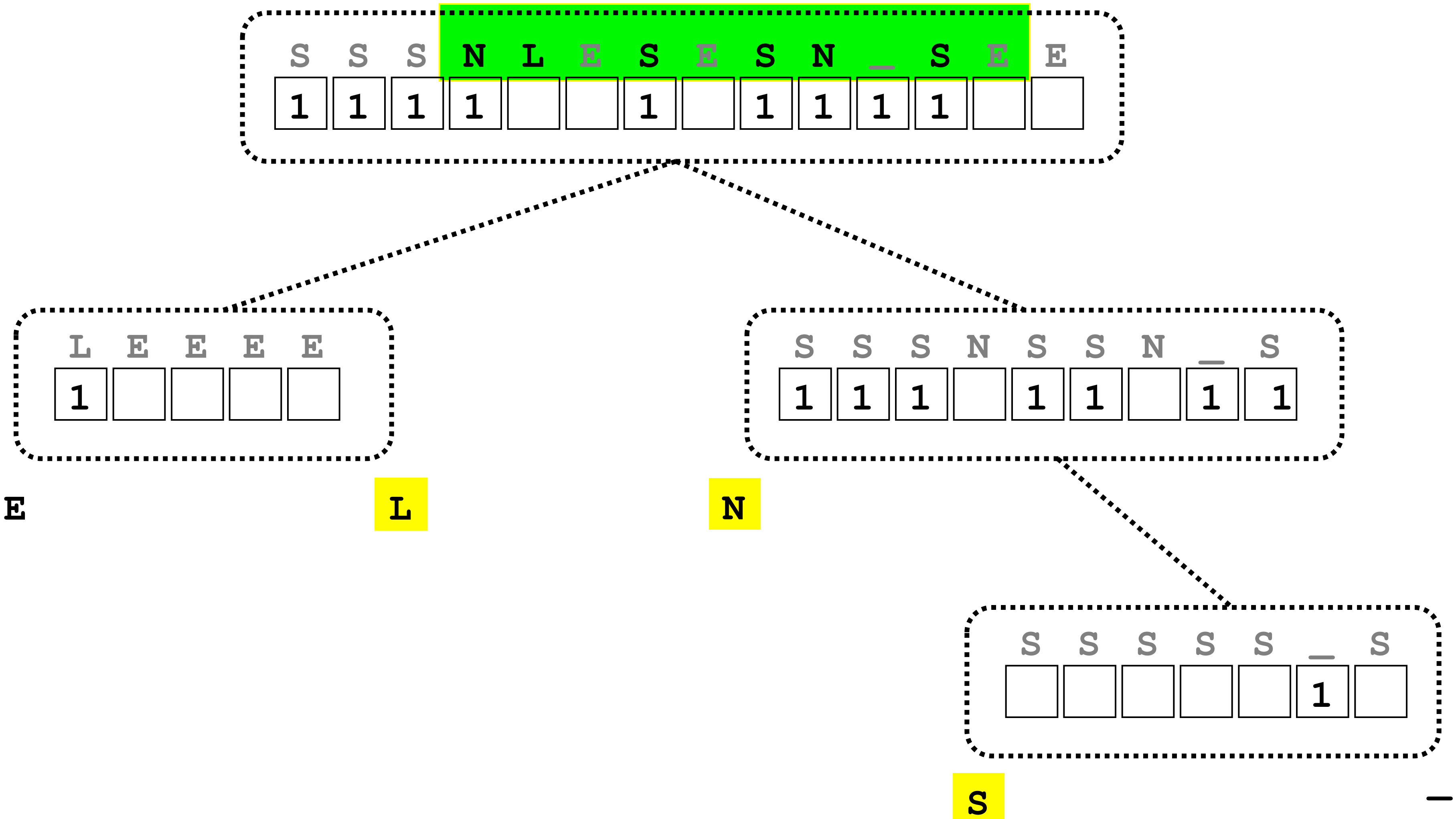
membership: *k*-th symbol



rank(c,k): # c's in the first k symbols



select(c,k): position of the k-th symbol c



2D-range($c..d$, $j..k$): like in Comp.Geom.

Wavelet tree represents…

- a **sequence of symbols** as we saw
- a **reordering** stored in the root of the multiset given by the elements in the leaves
- a **set of grid points**, where the x-coordinates are the positions and and the y-coordinates are the values

A list of applications [Navarro ’12]

- entropy encoding with direct access
- range queries
- positional inverted indexes
- graph representation
- permutations
- numeric sequences
- document retrieval
- binary relations
- data mining

… not always the best bounds, but easy to get them

Some theory

Quick Analysis of Space Occupancy

- B^j = j-th raw bit string in preorder traversal of the wavelet tree

- Fact 1. $\sum_j |B^j|\, H_0(B^j) = |string| \times H_0(string)$

- Fact 2 [Makinen-Navarro]. Wavelet on BWT(string)

$$\sum_j |B^j|\, H_0(B^j) = |string| \times H_k(string)$$

(order-k empirical entropy)

The rationale behind wavelet trees...

FOCUS on bit string $B \equiv B^j$

Fully indexable dictionaries (FID)

[Brodnik, Munro ‘99, Raman et al ’02]

- Bitvector V with n **1**s and m-n **0**s:
  - $rank_1(i)$ = #**1**s in V[1...i]
  - $select_1(j)$ = position in V of the jth **1**
  - same operations for **0**s
- Can also solve the precedessor problem

- space: B(n,m) + o(m) ~ n H_0 + o(m) bits
- time: O(1)

$$B(n,m) = \left\lceil \log \binom{m}{n} \right\rceil$$

Run Length Encoding (RLE) of a Bit String

- Bit string $B = B[1 \ldots n] = b_1^{l_1} b_2^{l_2} \ldots b_m^{l_m}$
  - $b_i \neq b_j$, if $i \neq j$
  - $l_i > 0$

- Prefix-free positive integer coding [Elias' 75]
  - $|\gamma(x)| = 2\lfloor \log x \rfloor + 1$ bits
  - $|\delta(x)| = \lfloor \log x \rfloor + 2\lfloor \log (\lfloor \log x \rfloor + 1) \rfloor + 1$ bits

$$\left.\begin{array}{l} \text{• RLE-}\gamma \text{ of B:} \quad b_1\, \gamma(l_1)\, \gamma(l_2) \ldots \gamma(l_m) \\ \text{• RLE-}\delta \text{ of B:} \quad b_1\, \delta(l_1)\, \delta(l_2) \ldots \delta(l_m) \end{array}\right\} \text{Uniquely decodable}$$

RLE of a Wavelet Tree

- T: an arbitrary text
  - Drawn from an alphabet of size σ
- $T_{rle,\gamma}$ = wavelet tree of T with each bit array RLE-γ encoded
- $T_{rle,\delta}$ = wavelet tree of T with each bit array RLE-δ encoded

Size of the RLE of a Wavelet Tree

- Total #bits in all the RLE encoded bit arrays

- $|B_{rle,\gamma}| \leq \mathbf{2}\ nH_0(B) + 2 \log n + 2$

  $|T_{rle,\gamma}| \leq \mathbf{2}\ nH_0(T)+(2\log n+2)(\sigma-1)$

- $|B_{rle,\delta}| \leq \mathbf{3}\ nH_0(B) + 2\log(\log n +1) + \log n + 2$

  $|T_{rle,\delta}| \leq \mathbf{3}\ nH_0(T)+(2\log(\log n + 1)+\log n + 2)(\sigma - 1)$

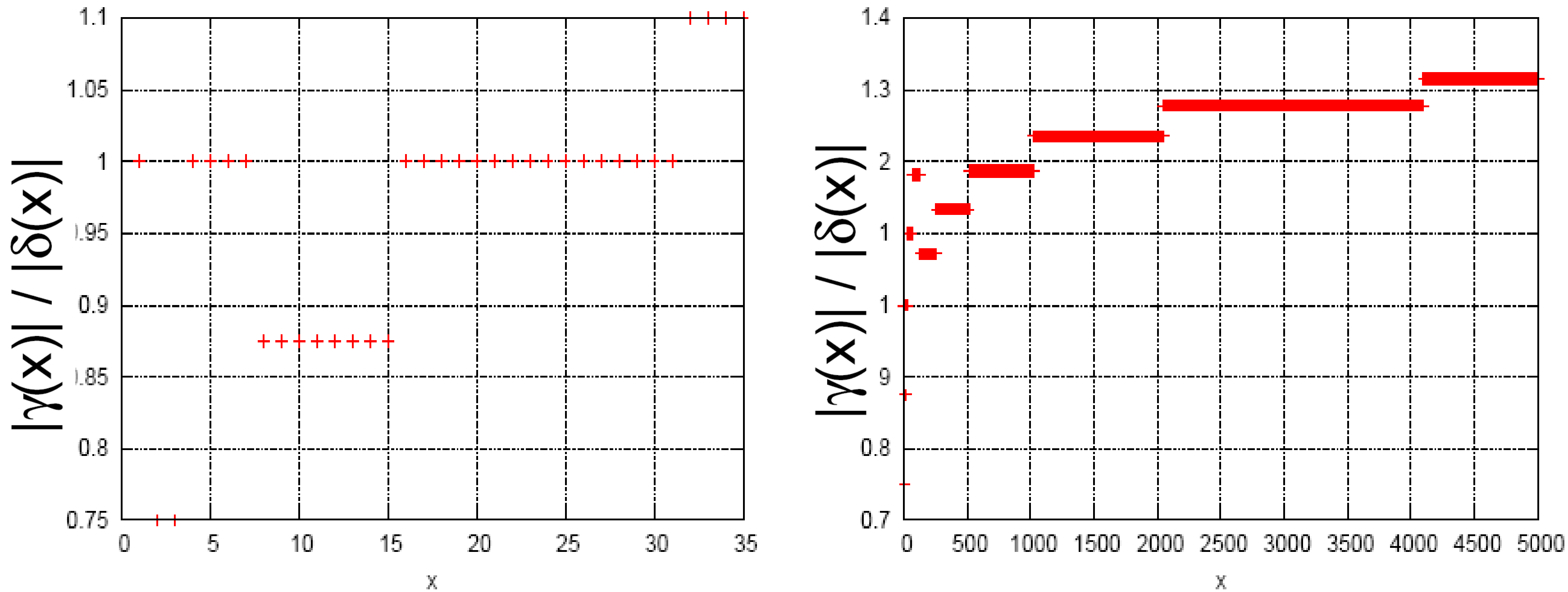
RLE-δ is less space efficient for an arbitrary text !!

Why is RLE-δ Less Space Efficient ?

- Although:
  - $\gamma(x)$ uses $\mathbf{2}\lfloor \log x \rfloor + 1$ bits
  - $\delta(x)$ uses $\lfloor \log x \rfloor + 2\lfloor \log (\lfloor \log x \rfloor + 1) \rfloor + 1$ bits

- Actually:
  $\delta(x)$ uses more bits if x is small, which is often the case in the wavelet tree bit arrays.
  [well-known fact]

Why is RLE-δ Less Space Efficient ? (Cont’d)



(a) Over a narrow range of x (b) Over a wide range of x

$|\delta(x)| < |\gamma(x)|$ for $x > 31$; $|\delta(x)| = |\gamma(x)|$ for $x \in \{1, 4\text{–}7, 16\text{–}31\}$; $|\delta(x)| > |\gamma(x)|$ for $x \in \{2\text{–}3, 8\text{–}15\}$.

A comparison of |γ(x)| and |δ(x)|

Can RLE-δ become provably better ?

- **Yes, when the data is highly skewed.**
- **For a bit string B**
  - If $H_0(B) = o(1)$:

    $|B_{rle,\delta}| \leq \mathbf{nH_0(B)} + o(nH_0(B)) + 2\log(\log n+1) + \log n + 2$ (optimal)

    (while $|B_{rle,\gamma}| \leq \mathbf{2nH_0(B)} + 2 \log n + 2$)

- **For an arbitrary text T from alphabet of size σ**
  - If $\sigma = O(1)$ and $H_0(T) = o(\log \sigma)$: (optimal)

    $|T_{rle,\delta}| \leq nH_0(T) + o(nH_0(T)) + (2\log(\log n + 1) + \log n + 2)(\sigma - 1)$

    (while $|T_{rle,\gamma}| \leq 2nH_0(T) + (2\log n + 2)(\sigma - 1)$)

Experiments

Software tool for wavelet trees

3 tree shapes (balanced, Huffman, Hu-Tucker)

X

11 coding schemes for bit strings B^i

- Can experiment 33 possible incarnations...
- URL: http://penguin.ewu.edu/~bojianxu/publications

Experimental results

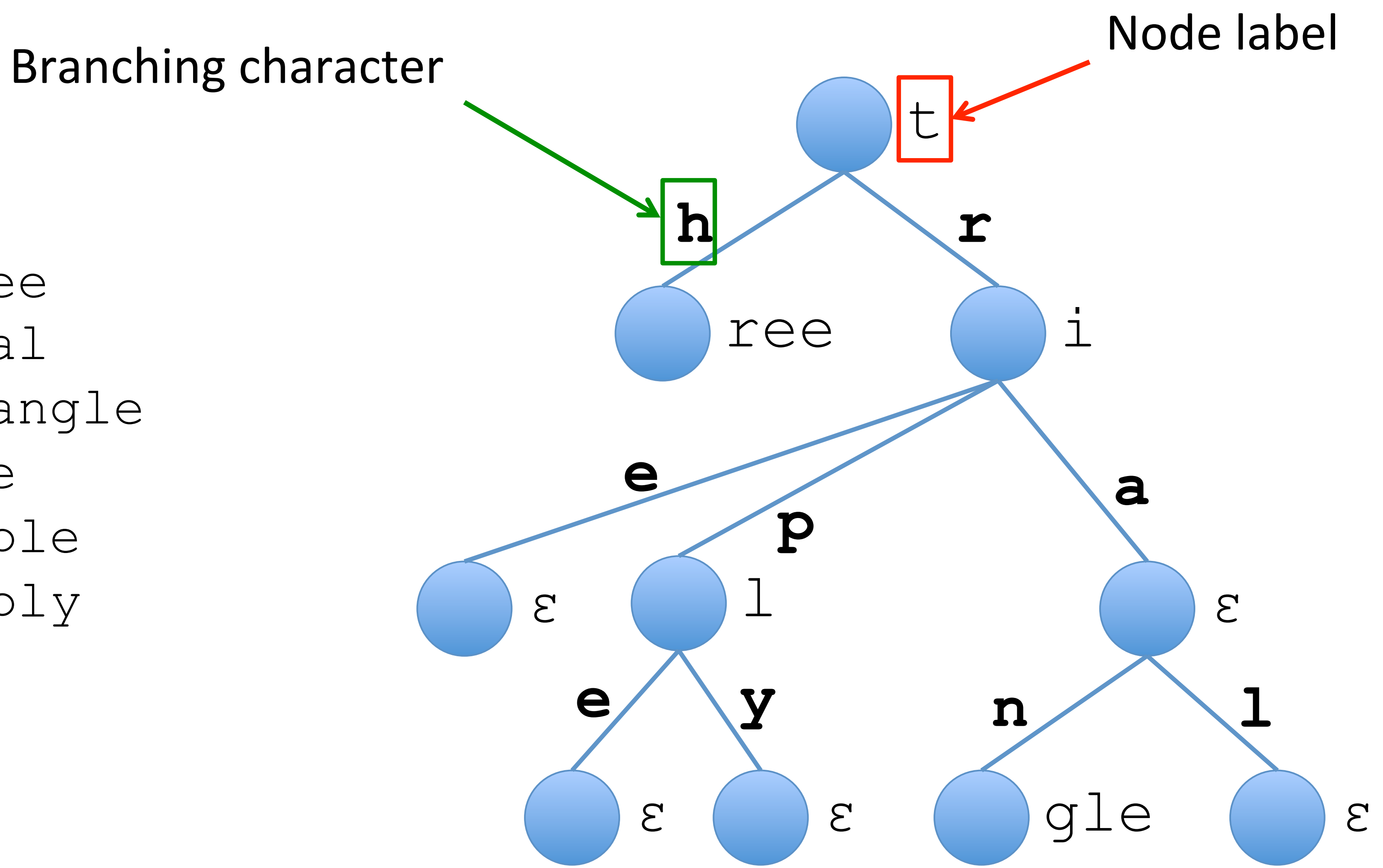
1. RLE+γ is good for low-entropy text
2. Hu-Tucker shape as good as Hufmann shape, except for low-entropy data
3. Queries are faster in Hu-Tucker shape

Part II

Fast Compressed Tries through Path Decompositions

Compacted tries

```
three
trial
triangle
trie
triple
triply
```



Applications

- String dictionaries
  - With prefix lookup, predecessor, …
  - Exploit prefix compression
- Monotone perfect hash functions
  - “Hollow” or “Blind” tries [ALENEX 09]
  - Binary tree (no need store branching chars)
  - No need to store node labels, just lengths (skips)

Height vs. performance

- Tries can be deep – no guarantee on height
- Bad with pointer-based trees
  - ~1 cache miss per *child* operation
- Worse with succinct tree encodings
  - Need to access several directories
  - *Many* cache misses per *child* operation
  - Large constants hidden in the O(1)

Problem

- Given a sorted set *S* of *K* prefix-free strings
  - $\{s_1, s_2, s_3, \ldots., s_K\}$
  - With total length $N = \sum |s_i|$
  - With each character drawn from $\sum = \{1,2,..,\sigma\}$

- Create a data structure that answers the following queries for a pattern p

  - **Lookup(p)** returns -1 if p is not in S or a unique ID in the interval [K]
  - **Access(*i*)** retrieves the string with ID = *i*

Linearized Trie Lower Bound: LT(S)

- Let T be the compacted trie of S
- Let E be # characters on the edges of T
- Let t be the total number of nodes of T

Theorem. Any succinct encoding of S requires at least

$$\boxed{LT(S) = E \log \sigma + B(t-1,E)}$$

bits in the worst case.

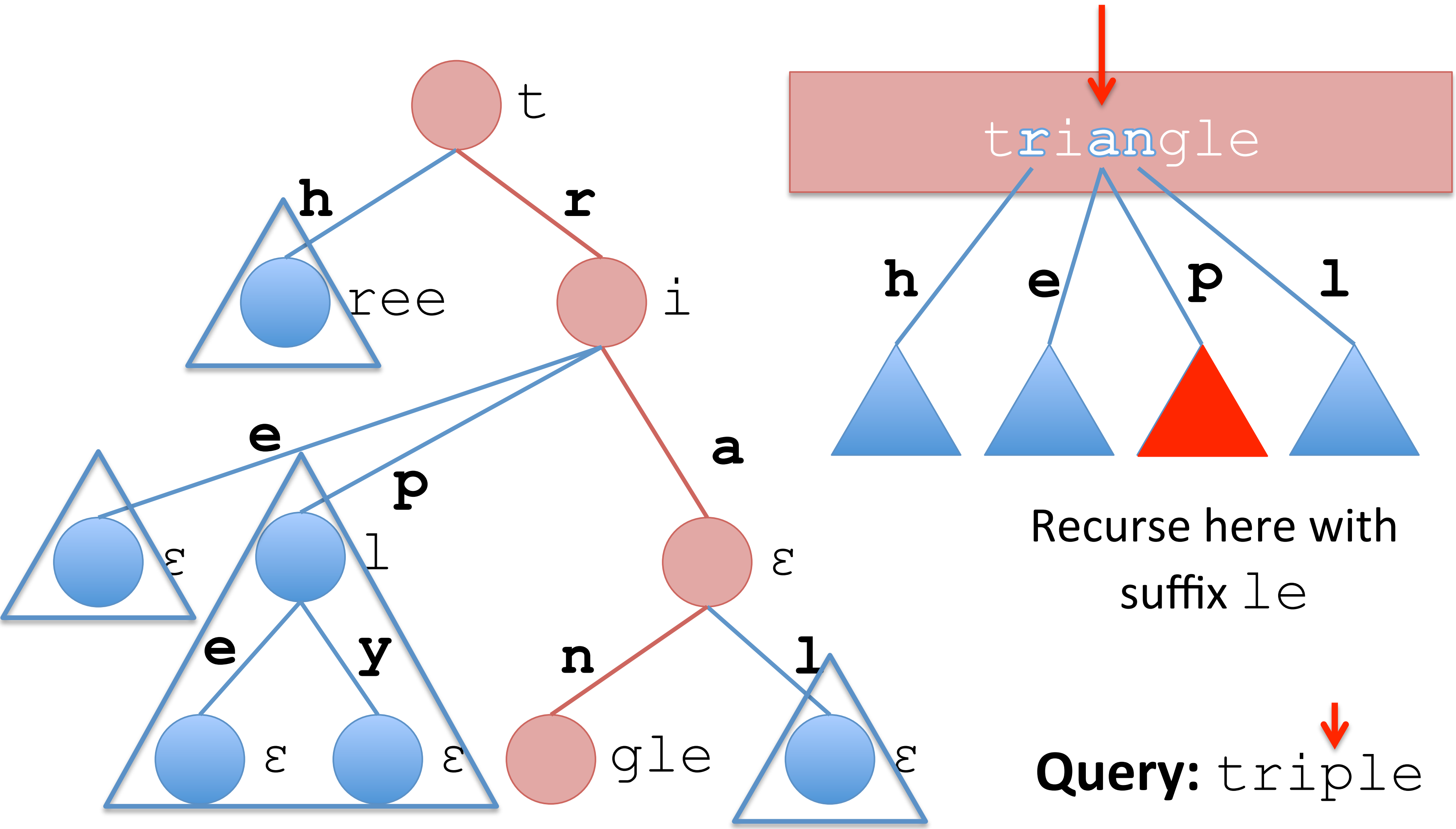
$$B(n,m) = \left\lceil \log \binom{m}{n} \right\rceil$$

Upper Bounds

Encoding for S that

- supports Lookup(p) and Access(i) = p in O(p) time
- uses (1 + o(1)) LT(S) + O(K) bits
- easy to implement and fast

Path decomposition



Recurse here with suffix le

Query: triple

Centroid path decomposition

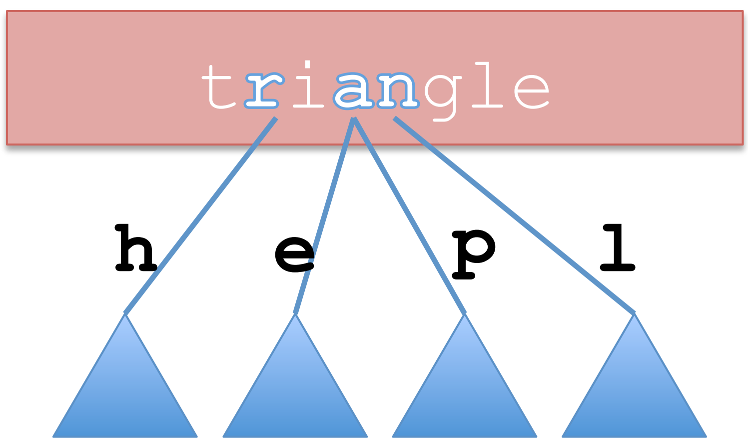
- Decompose along the **heavy paths**
  - choose the edge that has most descendants
- Height of the decomposed tree: O(log n)
  - Usually lower
- Average height

	Web Queries	URLs	Synthetic
Compacted trie	11.0	18.1	504.4
Centroid trie	**5.2**	**6.2**	**2.8**
Hollow trie	50.8	67.3	1005.3
Centroid hollow trie	**8.0**	**9.2**	**2.8**

Succinct encoding

- [PODS 08] presents a succinct data structure for centroid path-decomposed tries
- Not practical: need complex operations on succinct trees
- Simpler and practical encoding
- This encoding enables also simple compression of the labels

Succinct encoding




```
L :  t1ri2a1ngle
BP:   (  (((    )
B :   h  epl
```

(spaces added for clarity)

- Node label written literally, interleaved with number of *other* branching characters at that point in array **L**
- Corresponding branching characters in array **B**
- Tree encoded with DFUDS in bitvector **BP**
  - Variant of Range Min-Max tree [ALENEX 10] to support Find{Close,Open}, more space-efficient (Range Min tree)

Compression of **L**

```
...$...index.html$....html$....html$...index.html$
```

```
...
3 index
...
5 .html
...
```

Dictionary

```
...$...35$...5$...5$...35$
```

- Dictionary codewords shared among labels
- Codewords do not cross label boundaries (`$`)
- Use vbyte to compress the codeword ids

Compression of **L**

- Node labels (`t`<u>**1**</u>`ri`<u>**2**</u>`a`<u>**1**</u>`ngle`, `l`<u>**1**</u>`e`, …):
  - each label is suffix of a string in the set
  - interleaved with few "special characters" <u>**1**</u>, <u>**2**</u>, <u>**3**</u>,…
- Compressible if strings are compressible
- Dictionary and parsing computed with modified Re-Pair
  - Domain-specific compression can be used instead
- Decompression overhead negligible

Experimental results (time)

- Experiments show gains in time comparable to the gains in height
- Confirm that bottleneck is traversal operations

	Web Queries	URLs	Synthetic
Trie	3.5	7.0	119.8
Centroid trie	**2.4**	**4.3**	**5.1**
Hollow trie [ALENEX 09]	16.6	22.4	462.7
Hollow trie	7.2	13.9	137.1
Centroid hollow trie	**2.8**	**4.4**	**11.1**

(microseconds, lower is better)

Code available at `https://github.com/ot/path_decomposed_tries`

Experimental results (space)

- For strings with many common prefixes, even non-compressed trie is space-efficient
- Labels compression considerably increases space-efficiency
- Decompression time overhead: ~10%

	Web Queries	URLs	Synthetic
Hu-Tucker Front Coding	40.9%	24.4%	19.1%
Centroid trie	55.6%	22.4%	17.9%
Centroid trie + compression	**31.5%**	**13.6%**	**0.4%**

(compression ratio, lower is better)

Code available at `https://github.com/ot/path_decomposed_tries`

Part III:

Wavelet Trie for a **Sequence** of Strings

Sequence of strings ≠ Set of strings

- **Time order** e.g. in data analytics:

  What has been the most accessed domain during winter vacation?

- **Binary relation** e.g. in web graphs and social networks:

  How did friendship links change during winter vacation?

Indexed String Sequences

(foo, bar, foobar, foo, bar, bar, foo)

0 1 2 3 4 5 6

- **Queries**
  - **Access(i):** access the *i*-th element
    - Access(2) = `foobar`
  - **Rank(s, pos):** count occurrences of *s* before *pos*
    - Rank(`bar`, 5) = 2
  - **Select(s, i):** find the *i*-th occurrence of a *s*
    - Select(`foo`, 2) = 6

Prefix operations

(foo,	bar,	foobar,	foo,	bar,	bar,	foo)
0	1	2	3	4	5	6

- **Queries**
  - **RankPrefix(p, pos):** count strings prefixed by *p* before *pos*
    - RankPrefix(`foo`, 5) = 3
  - **SelectPrefix(p, i):** find the *i*-th string prefixed by *p*
    - SelectPrefix(`foo`, 2) = 3

Example: storing relations

- Write the **columns** as string sequences
  - Store them separately
  - Reduce relational operations to **sequence queries**

	User			Likes URL
0	Leonard	battl	0	et/wow/
1	Penny	tmz.	1	ı
2	Sheldon	ba	2	et/wow/
3	Penny	thec	3	secakefactory.com
4	Leonard	wi	4	ia.org/Star_Trek
5	Sheldon	wi	5	ia.org/String_theory
6	Sheldon	ma	6	com

What does **Sheldon** like?

Who likes pages from domain **wikipedia.org**?

Other operations: range counting, …

Some notation

```
(foo, bar, foobar, foo, bar, bar, foo)
  0    1     2      3    4    5    6
```

- Sequence S, |S| = n
  - In the example n = 7
- String set S_{set} is *unordered set* of *distinct* strings appearing in S
  - In the example, `{foo, bar, foobar}`, $|S_{set}| = 3$
  - Also called *alphabet* but is LARGE and VARIABLE
- Sequence symbols can also be integers, characters, …
  - As long as they are binarized to strings

Dynamic sequences

We want to support the following operations:

- **Insert(s, pos)**: insert the string *s* immediately before position *pos*
- **Append(s)**: append the string *s* at end of the sequence (special case of **Insert**)
- **Delete(pos)**: delete the string at position *pos*

If data structure only supports **Append**, we call it *append-only*, otherwise *dynamic* (or *fully dynamic*)

All the operations can change S_{set}

The Wavelet Trie

- The Wavelet Trie is a Wavelet Tree on sequences of binary strings ($S_{set} \subset \{0, 1\}^*$)
- Supports Access/Rank(Prefix)/Select(Prefix)
- Fully dynamic…
- … or append only (with better bounds)
- The string set S_{set} *need not be known in advance*

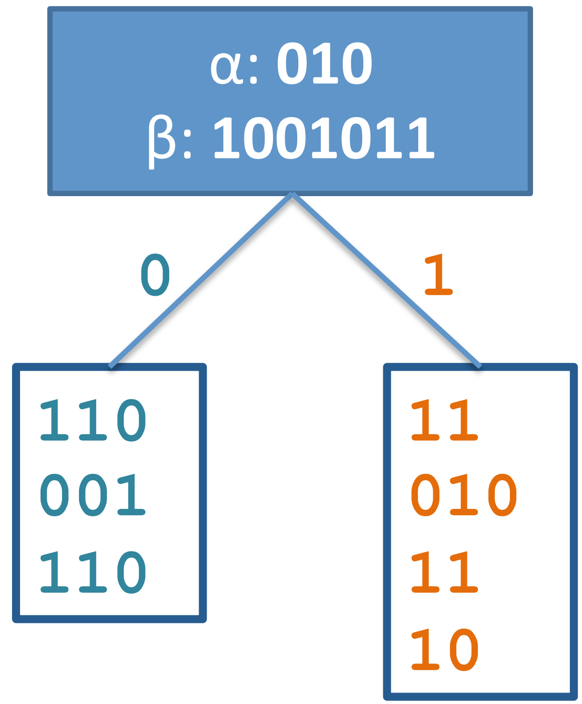
Wavelet Trie: Construction

Sequence of **binary** strings

Branching bit: β

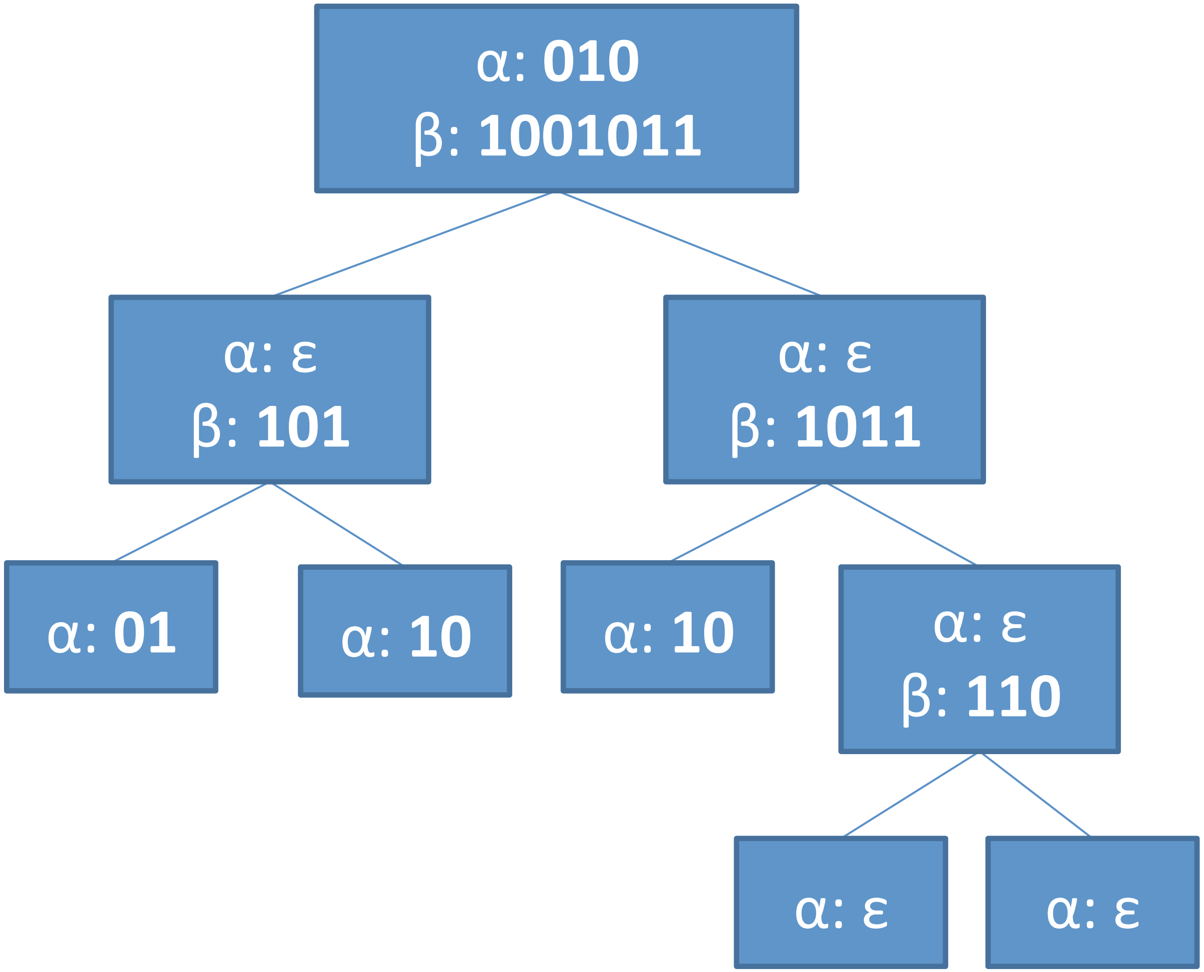
```
0101 11
0100 110
0100 001
0101 010
0100 110
0101 11
0101 10
```

Common prefix: α



Wavelet Trie: Construction

0 1 0 1 1 1
0 1 0 0 1 1 0
0 1 0 0 0 0 1
0 1 0 1 0 1 0
0 1 0 0 1 1 0
0 1 0 1 1 1
0 1 0 1 1 0



Space analysis

- Information-theoretic lower bound

$$LB(S) = LT(S_{set}) + nH_0(S)$$

  - LT is the information-theoretic lower bound for storing a *set* of strings (see previous slides)
  - $nH_0(S)$ is the 0-th order entropy of a sequence S of n atomic items

- $\hat{h}$ = avg height of the Wavele Trie (WT)

Space analysis

- Static WT:

$$LB(S) + o(\hat{h}n)$$

- Append-only WT:

$$LB(S) + PT(S_{set}) + o(\hat{h}n)$$

  – $PT(S_{set})$: space taken by the Patricia Trie

- Fully dynamic WT:

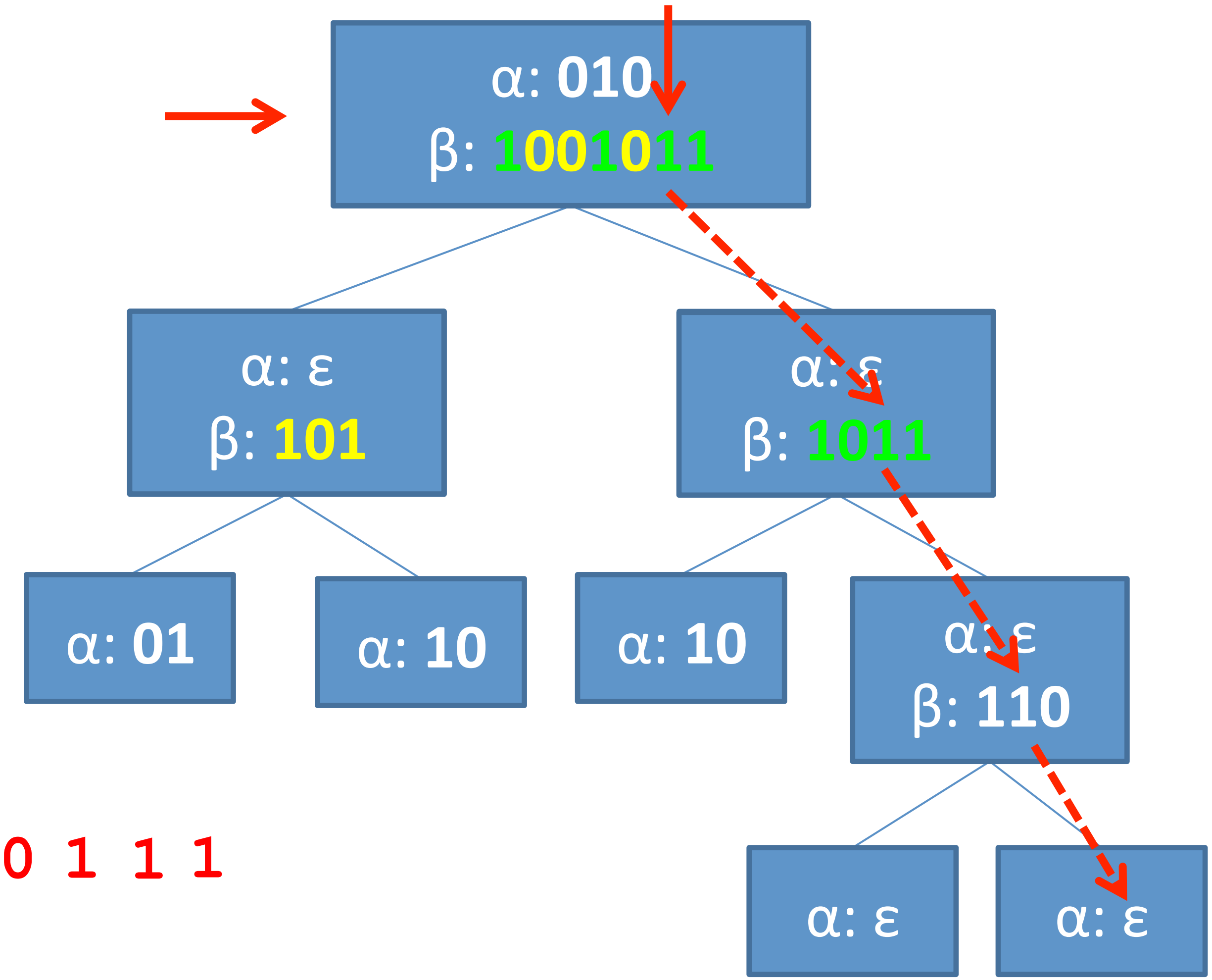
$$LB(S) + PT(S_{set}) + O(nH_0(S))$$

Operations time complexity

- Need new dynamic bitvectors to support *initialization* (create a bitvector $\mathtt{0}^n$ or $\mathtt{1}^n$)
- Static and Append-only Wavelet Trie
  - All supported operations on *s* in $O(|s| + h_s)$
  - h_s is number of nodes traversed by string *s*
- Fully dynamic Wavelet Trie
  - All supported operations in $O(|s| + h_s \log n)$
  - Deletion may take $O(|\hat{s}| + h_s \log n)$ where $\hat{s}$ is longest string in the trie

Wavelet Trie: Access

0 010111
1 0100110
2 0100001
3 0101010
4 0100110
5 010111
6 010110

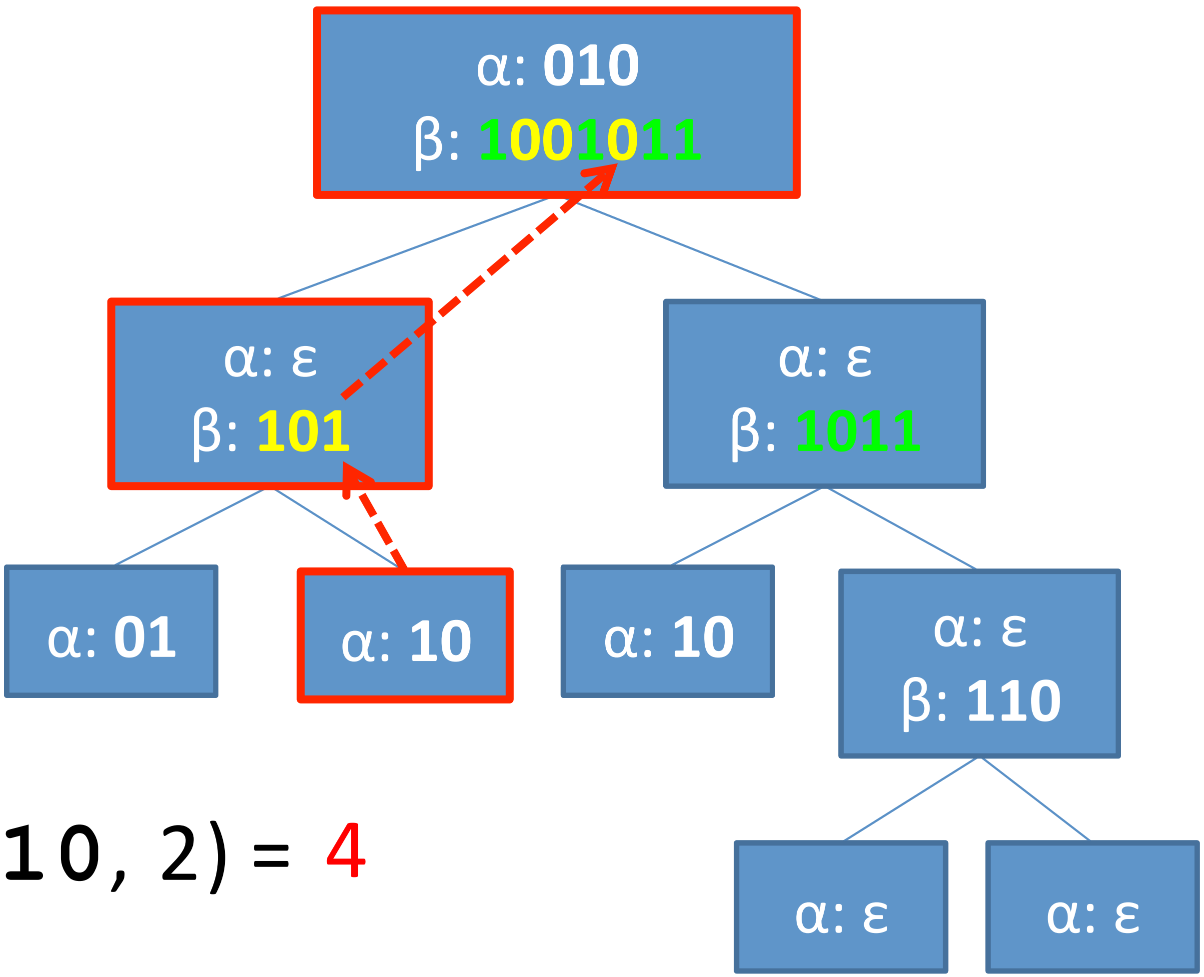


Access(5) = 010 1 1 1

Rank is similar

Wavelet Trie: Select

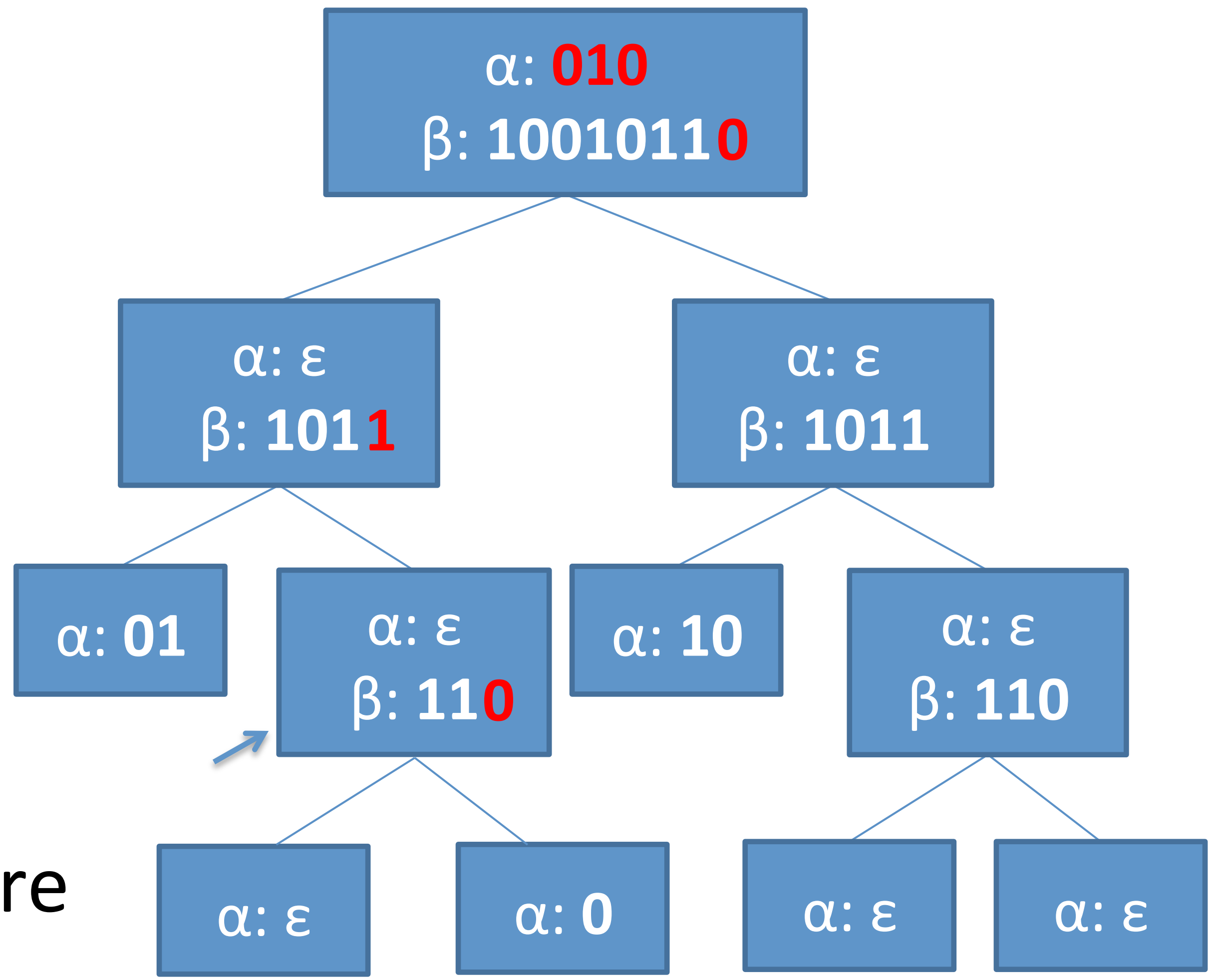
0 010111
1 0100110
2 0100001
3 0101010
4 0100110
5 010111
6 010110



Select(0100110, 2) = 4

Wavelet Trie: Append

010111
0100110
0100001
0101010
0100110
010111
010110
010010



Insert/Delete are similar

Summary

- Compressed suffix arrays for a **string** = **sequence of symbols**
- Wavelet trees for a string
- Succinct tries for a **set of strings**
- Wavelet tries for a **sequence of strings**

- Industrial applications: json processing, query log analysis, auto-completion
- Code being developed is available at https://github.com/ot/succinct

Thanks for your attention!

Questions?