Backward Search
FM-Index
(Full-text index in Minute space)

Motivation

- Combine Text compression with indexing (discard original text).
- Count and locate P by looking at only a small portion of the compressed text.
- Do it efficiently:
  - Time: $O(p)$
  - Space: $O(n\, H_k(T)) + o(n)$

HOW DOES IT WORK?

- Exploit the relationship between the *Burrows-Wheeler Transform* and the Suffix Array data structure.

- Compressed suffix array that encapsulates both the ***compressed text*** and the ***full-text indexing information.***

- Supports two basic operations:
  - **Count** – return number of occurrences of P in T.
  - **Locate** – find all positions of P in T.

Burrows-Wheeler Transform

- Every column is a permutation of T.
- Given row i, char L[i] precedes F[i] in original T.
- Consecutive char's in L are adjacent to similar strings in T.
- Therefore – L usually contains long runs of identical char's.

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

Burrows-Wheeler Transform

Reminder: Recovering T from L

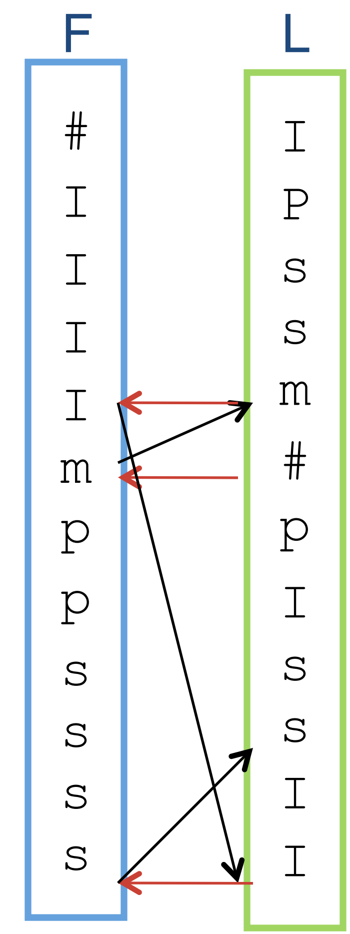
1. Find F by sorting L
2. First char of T?

   m

3. Find m in L
4. L[i] precedes F[i] in T. Therefore we get

   mi

5. How do we choose the correct i in L?
   - The i's are in the same order in L and F
   - As are the rest of the char's
6. i is followed by s: mis
7. And so on….



Next: Count P in T

- **Backward-search algorithm**
- Uses only L (output of BWT)
- Relies on 2 structures:
  - C[1,…,|Σ|] : C[c] contains the total number of text chars in T which are alphabetically smaller than c (including repetitions of chars)
  - Occ(c,q): number of occurrences of char c in prefix L[1,q]

Example

- C[ ] for T = mississippi#

1	5	6	8
i	m	p	s

- occ(s, 5) = 2
- occ(s,12) = 4

Occ ≡ Rank

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

BACKWARD-SEARCH

- Works in p iterations, from p down to 1

P = msi → (i = 3, c = 'i') → ms*i* → (i = 2, c = 's') → m*si* → (i = 1, c = 'm') → ***msi***

- Remember that the BWT matrix rows = sorted suffixes of T
  - All suffixes prefixed by pattern P, occupy a continuous set of rows
  - This set of rows has starting position ***First***
  - and ending position ***Last***
  - So, (Last – First +1) gives total pattern occurrences

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

- At the end of the i-th phase, ***First*** points to the first row prefixed by P[i,p], and ***Last*** points to the last row prefiex by P[i,p].

Algorithm backward_search($P[1,p]$)

(1) $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;

(2) **while** ((First $\leq$ Last) **and** ($i \geq 2$)) **do**

(3) $\quad c \leftarrow P[i-1]$;

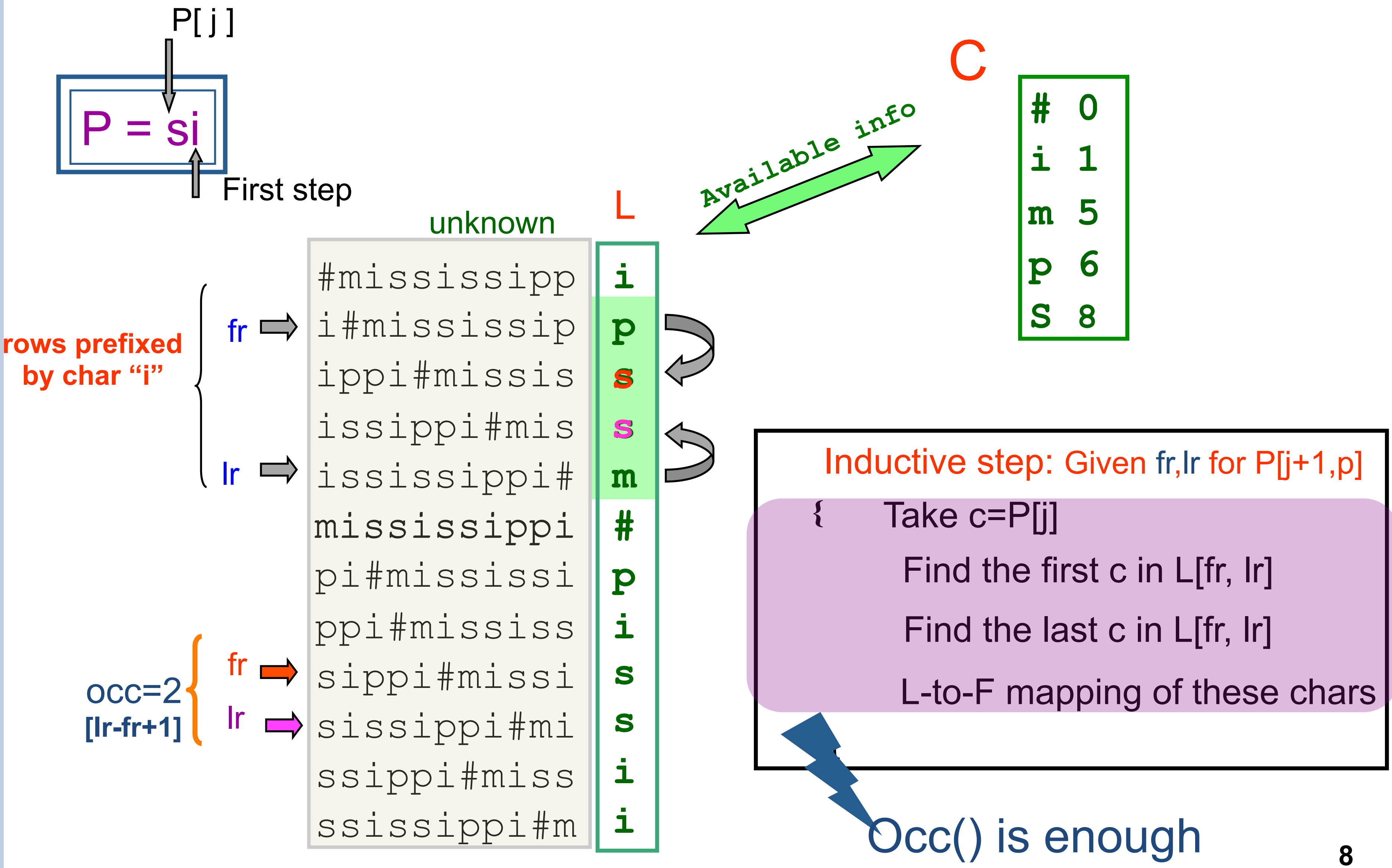
(4) $\quad$ First $\leftarrow C[c] + \text{Occ}(c, \text{First}-1)+1$;

(5) $\quad$ Last $\leftarrow C[c] + \text{Occ}(c, \text{Last})$;

(6) $\quad i \leftarrow i-1$;

(7) **if** (Last < First) **then return** "no rows prefixed by $P[1,p]$" **else return** $\langle$First, Last$\rangle$.

Substring search in T (Count the pattern occurrences)



Backward-search example

- P = pssi
  - i = 3
  - c = ‘s’
  - First = C[‘s’] + Occ(‘s’,1) +1 = 8+0+1 = 9
  - Last = C[‘s’] + Occ(‘s’,5) = 8+2 = 10
  - (Last – First + 1) = 2

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

First → row 2; Last → row 5

C[ ] =

i	m	p	s
1	5	6	8

Algorithm backward_search($P[1,p]$)

$i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;
while ((First $\leq$ Last) **and** ($i \geq 2$)) **do**
    $c \leftarrow P[i-1]$;
    First $\leftarrow C[c]$ + Occ(c, First -1) $+1$;
    Last $\leftarrow C[c]$ + Occ(c, Last);
    $i \leftarrow i-1$;
if (Last $<$ First) **then return** “no rows prefixed by $P[1,p]$” **else return** ⟨First, Last⟩.

Backward-Search Example

- P = pssi
  - i = 2
  - c = 's'
  - First = C['s'] + Occ('s',8) +1 = 8+2+1 = 11
  - Last = C['s'] + Occ('s',10) = 8+4 = 12
  - (Last – First + 1) = 2

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

First → row 11; Last → row 12

C[ ] =

i	m	p	s
1	5	6	8

Algorithm backward_search($P[1,p]$)

(1) $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;

(2) **while** ((First $\leq$ Last) **and** $(i \geq 2)$) **do**

(3) $\quad c \leftarrow P[i-1]$;

(4) $\quad$ First $\leftarrow C[c] +$ Occ(c, First $-1) + 1$;

(5) $\quad$ Last $\leftarrow C[c] +$ Occ(c, Last);

(6) $\quad i \leftarrow i - 1$;

(7) **if** (Last < First) **then return** "no rows prefixed by $P[1,p]$" **else return** $\langle$First, Last$\rangle$.

BACKWARD-SEARCH EXAMPLE

- P = pssi
  - i = 1
  - c = 'p'
  - First = C['p'] + Occ('p',10) +1 = 6+2+1 = 9
  - Last = C['p'] + Occ('p',12) = 6+2 = 8
  - (Last – First + 1) = 0

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

First → row 11, Last → row 12

C[ ] =

i	m	p	s
1	5	6	8

Algorithm backward_search($P[1,p]$)

(1) $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow C[c]+1$, Last $\leftarrow C[c+1]$;

(2) **while** ((First $\leq$ Last) **and** $(i \geq 2)$) **do**

(3) $\quad c \leftarrow P[i-1]$;

(4) $\quad$ First $\leftarrow C[c]$ + Occ(c, First -1) $+1$;

(5) $\quad$ Last $\leftarrow C[c]$ + Occ(c, Last);

(6) $\quad i \leftarrow i-1$;

(7) **if** (Last < First) **then return** "no rows prefixed by $P[1,p]$" **else return** ⟨First, Last⟩.

ASSIGNMENT 2

- Create a simple search program that implements BWT backward search, which can efficiently search a BWT encoded file.
- The program also has the capability to encode a text file to a BWT-coded file
- The program also has the capability to decode the BWT encoded file back to its original file in a lossless manner.
- Text is delimited by new lines.

ASSIGNMENT 2

- Your C/C++ program, called **bwtsearch**
  - **Bwtsearch -e fileToBeEncoded outputFile**
  - **Bwtsearch -d fileToBeDecoded**
    - **standard output**
  - **Bwtsearch -s fileEncoded “queryString”**
    - **Output all the lines contain “queryString”**
    - **Highlight “queryString” if capable**
    - **The search results need to be sorted according to their line numbers.**

ASSIGNMENT 2

- The first four bytes (an int) of each given BWT encoded file are reserved for storing the position (zero-based) of the BWT array that contains the last character. As a result, a given BWT encoded file in this assignment is 4 bytes larger than its original text file.
- For example, if the original text file contains only banana$, then the BWT encoded file will be 11 bytes long. The first four bytes contain the integer 4 and the rest of the bytes contain annb$aa. i.e., The last character is at position 4 (= the fifth character since it is zero -based).

Assignment 2

- Since each line is delimited by a newline character, your output will naturally be displayed as one line (ending with a '\n') for each match. No line will be output more than once, i.e., if there are multiple matches in one line, that line will only be output once.

Assignment 2

- Your solution can write out **one** external index file.
- You may assume that the index file will not be deleted during all the tests for a given BWT file, and all the test BWT files are uniquely named. Therefore, to save time, you only need to generate the index file when it does not exist yet.

LECTURE 5

- Compressed suffix array / BWT

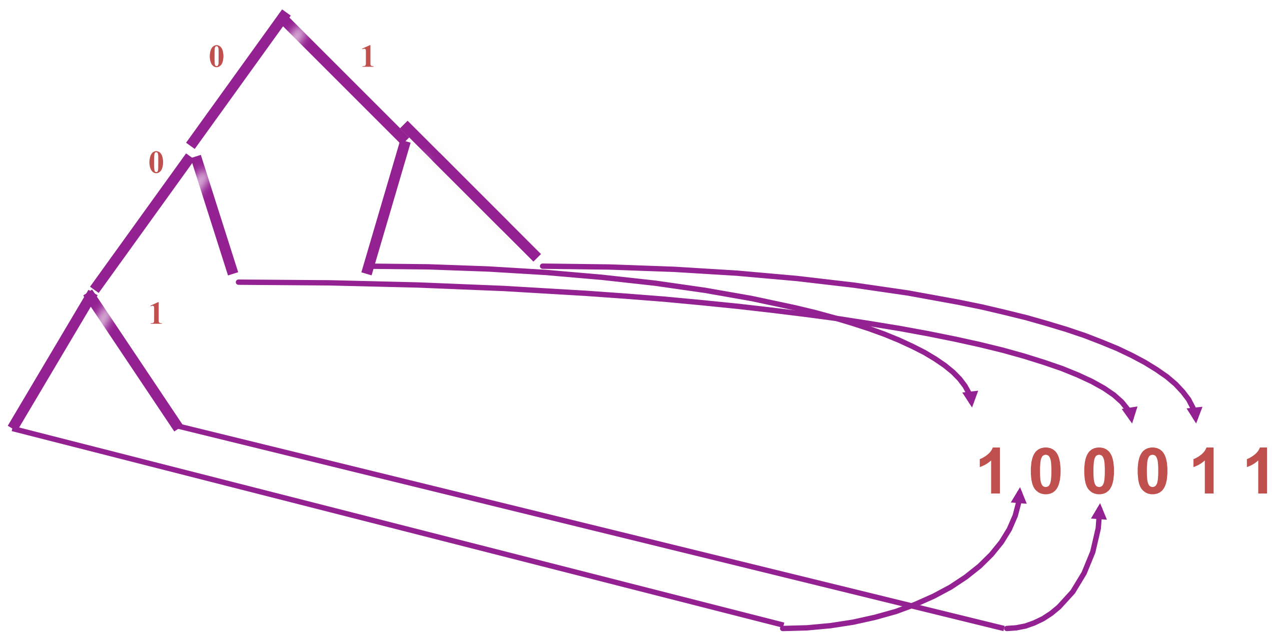
SUCCINCT SUFFIX ARRAYS BASED ON RUN-LENGTH ENCODING *

VELI MÄKINEN[†]
Dept. of Computer Science, University of Helsinki
Gustaf Hällströmin katu 2b, 00014 University of Helsinki, Finland
vmakinen@cs.helsinki.fi

GONZALO NAVARRO[‡]
Dept. of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Slides modified from the original Makinen & Navarro's

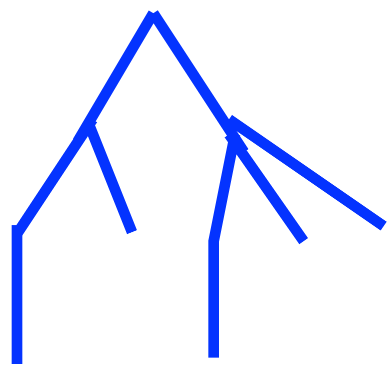
A Big Patricia Trie / Suffix Trie



- Given a large text file; treat it as bit vector
- Construct a trie with leaves pointing to unique locations in text that "match" path in trie (paths must start at character boundaries)
- Skip the nodes where there is no branching

ARBITRARY ORDERED TREES

- Use parenthesis notation
- Represent the tree



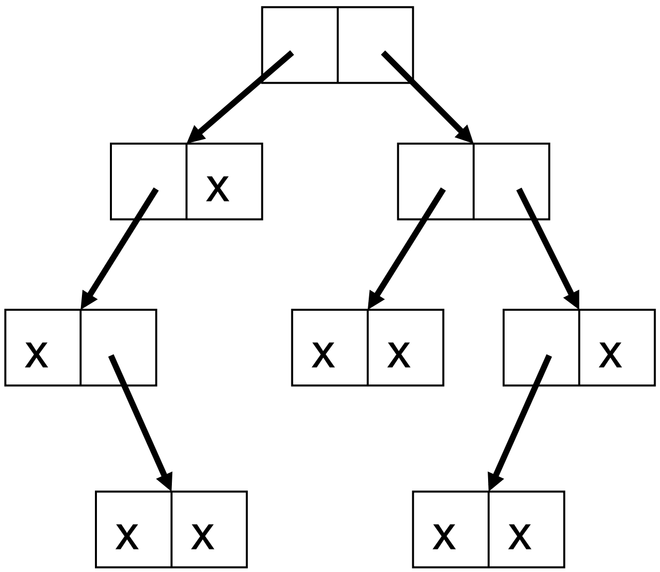
- As the binary string (((())())((())()())): traverse tree as "(" for node, then subtrees, then ")"
- 2 Bits per node

Space for trees

- The space used by the tree structure could be the dominating factor in some applications.
  - Eg. More than half of the space used by a standard suffix tree representation is used to store the tree structure.
- Standard representations of trees support very few operations. To support other useful queries, they require a large amount of extra space.

STANDARD REPRESENTATION

Binary tree: each node has two pointers to its left and right children



An n-node tree takes
$2n$ pointers or $2n \lg n$ bits

Supports finding left child or right child of a node (in constant time).

For each extra operation (eg. parent, subtree size) we have to pay, roughly, an additional $n \lg n$ bits.

CAN WE IMPROVE THE SPACE BOUND?

- There are less than 2^{2n} distinct binary trees on n nodes.
- $2n$ bits are enough to distinguish between any two different binary trees.
- Can we represent an n node binary tree using $2n$ bits?

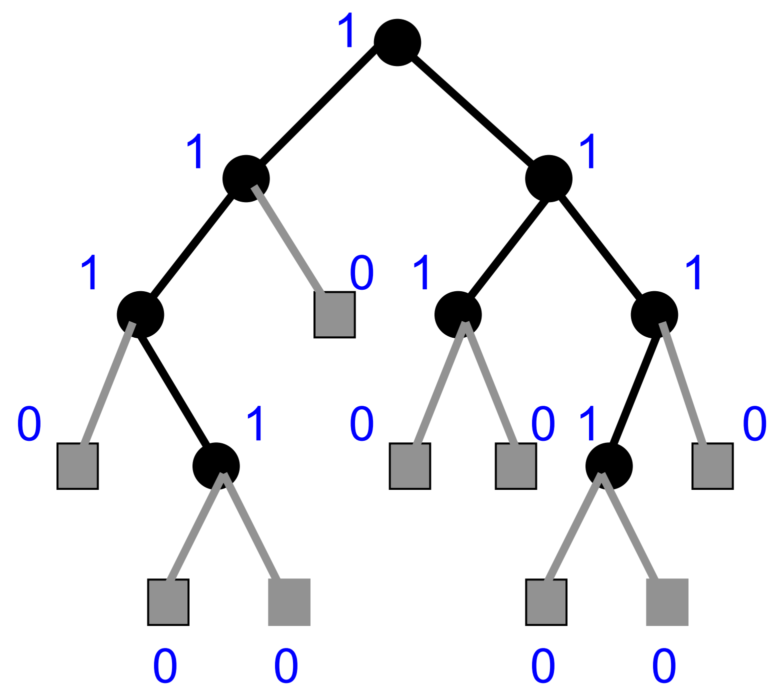
Heap-like notation for a binary tree

Add external nodes

Label internal nodes with a 1 and external nodes with a 0

Write the labels in level order

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



One can reconstruct the tree from this sequence

An n node binary tree can be represented in 2n+1 bits.

What about the operations?

HEAP-LIKE NOTATION FOR A BINARY TREE

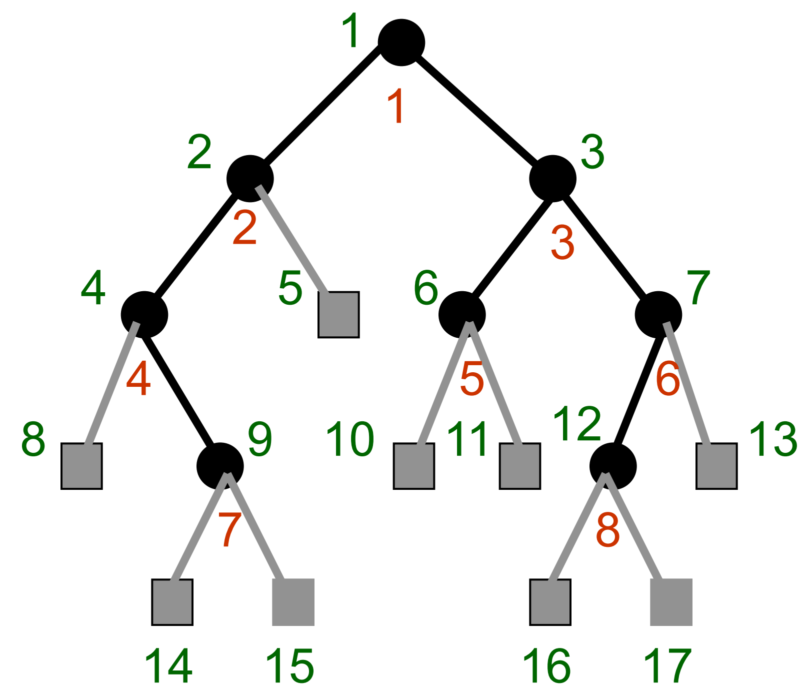
left child(x) = [2x]

right child(x) = [2x+1]

parent(x) = [⌊x/2⌋]

x → x: # 1's up to x

x → x: position of x-th 1



1	2	3	4		5	6		7			8					
1	1	1	1	0	1	1	0	1	0	0	1	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Rank/Select on a bit vector

Given a bit vector B

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B:	0	1	1	0	1	0	0	0	1	1	0	1	1	1	1

$rank_1(i)$ = # 1's up to position i in B

$select_1(i)$ = position of the i-th 1 in B

(similarly $rank_0$ and $select_0$)

$rank_1(5) = 3$
$select_1(4) = 9$
$rank_0(5) = 2$
$select_0(4) = 7$

Given a bit vector of length n, by storing an additional $o(n)$-bit structure, we can support all four operations in constant time.

An important substructure in most succinct data structures.

Have been implemented.

Binary tree representation

- A binary tree on n nodes can be represented using 2n+o(n) bits to support:

  - parent
  - left child
  - right child

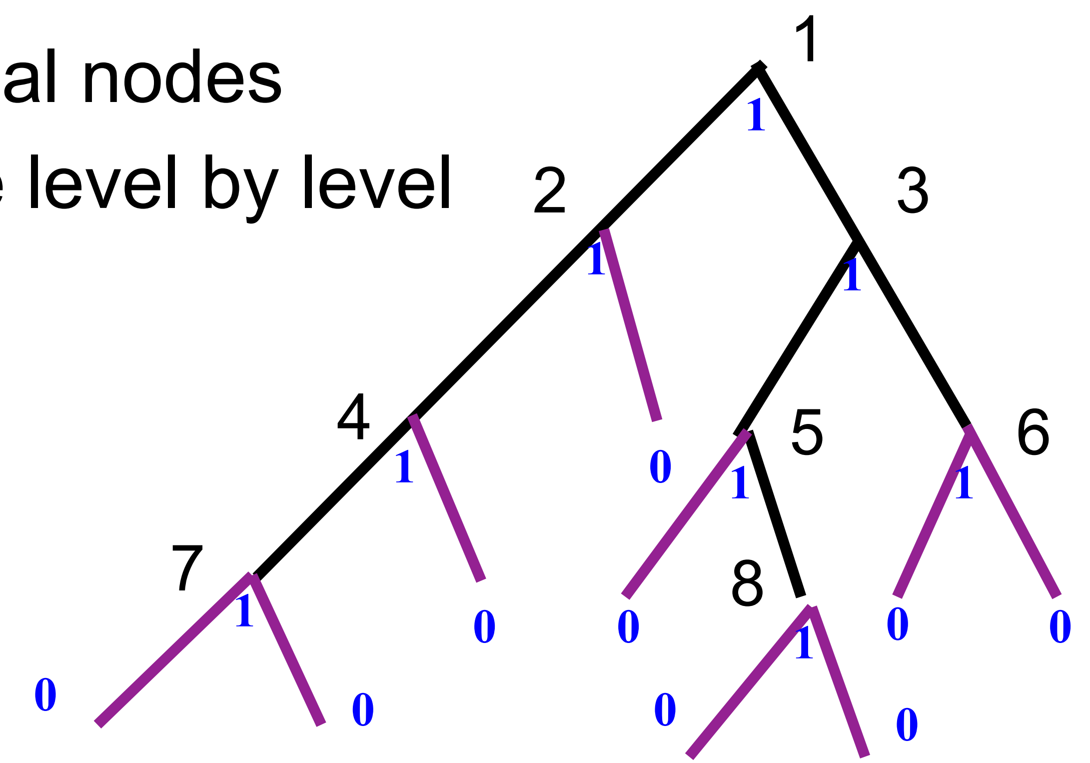
  in constant time.

- 1 1 1 1 0 1 1 1 0 0 1 0 0 0 0 0 0

Heap-like Notation for a Binary Tree

Add external nodes

Enumerate level by level



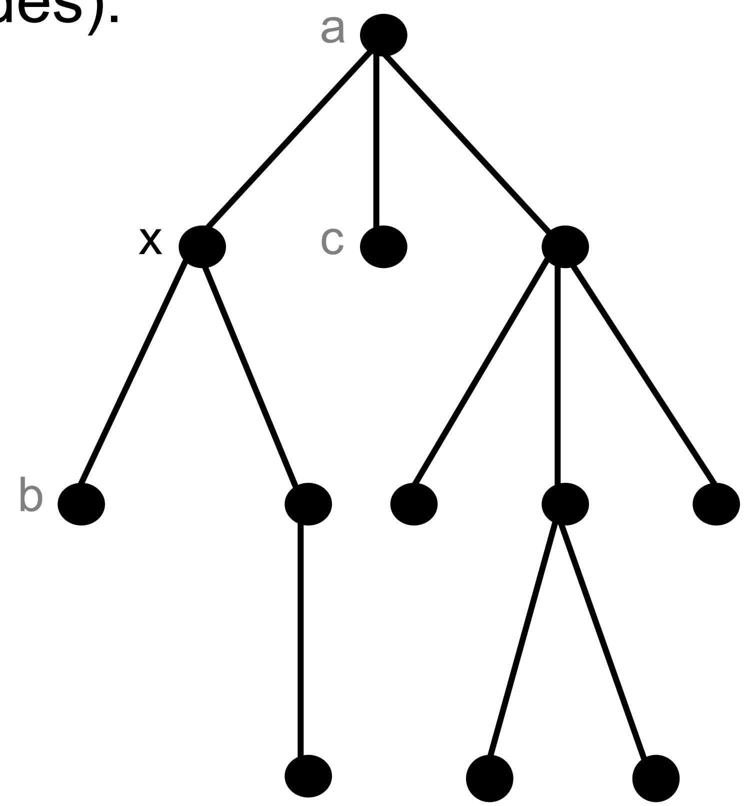
1 2 3 4 5 6 7 8

Store vector 1 1 1 1 0 1 1 1 0 0 1 0 00000 length 2n+1

1 2 3 4 5 6 7 8 9 0 1 2 34567

ORDERED TREES

A rooted ordered tree (on n nodes):



Navigational operations:

- parent(x) = a
- first child(x) = b
- next sibling(x) = c

Other useful operations:

- degree(x) = 2
- subtree size(x) = 4

Ordered trees

- A binary tree representation taking 2n+o(n) bits that supports parent, left child and right child operations in constant time.

- There is a one-to-one correspondence between binary trees (on n nodes) and rooted ordered trees (on n+1 nodes).

- Gives an ordered tree representation taking 2n+o(n) bits that supports first child, next sibling (but not parent) operations in constant time.

- We will now consider ordered tree representations that support more operations.

Level-order degree sequence

Write the degree sequence in level order

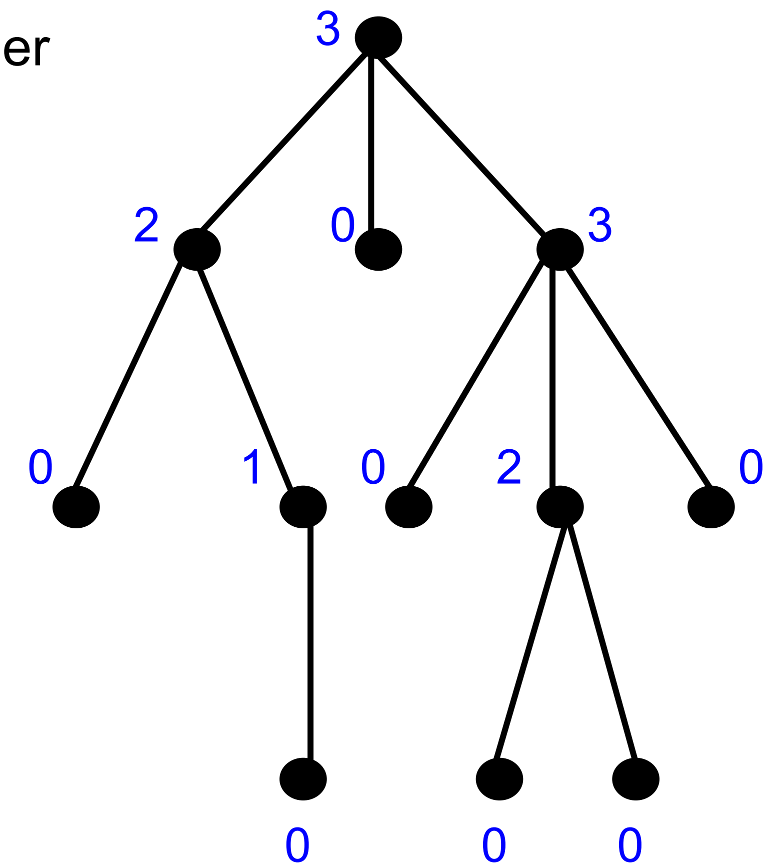
3 2 0 3 0 1 0 2 0 0 0 0

But, this still requires n lg n bits

Solution: write them in unary

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

Takes 2n-1 bits



A tree is uniquely determined by its degree sequence

SUPPORTING OPERATIONS

Add a dummy root so that each node has a corresponding 1

1 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1 0 0 1 1 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12

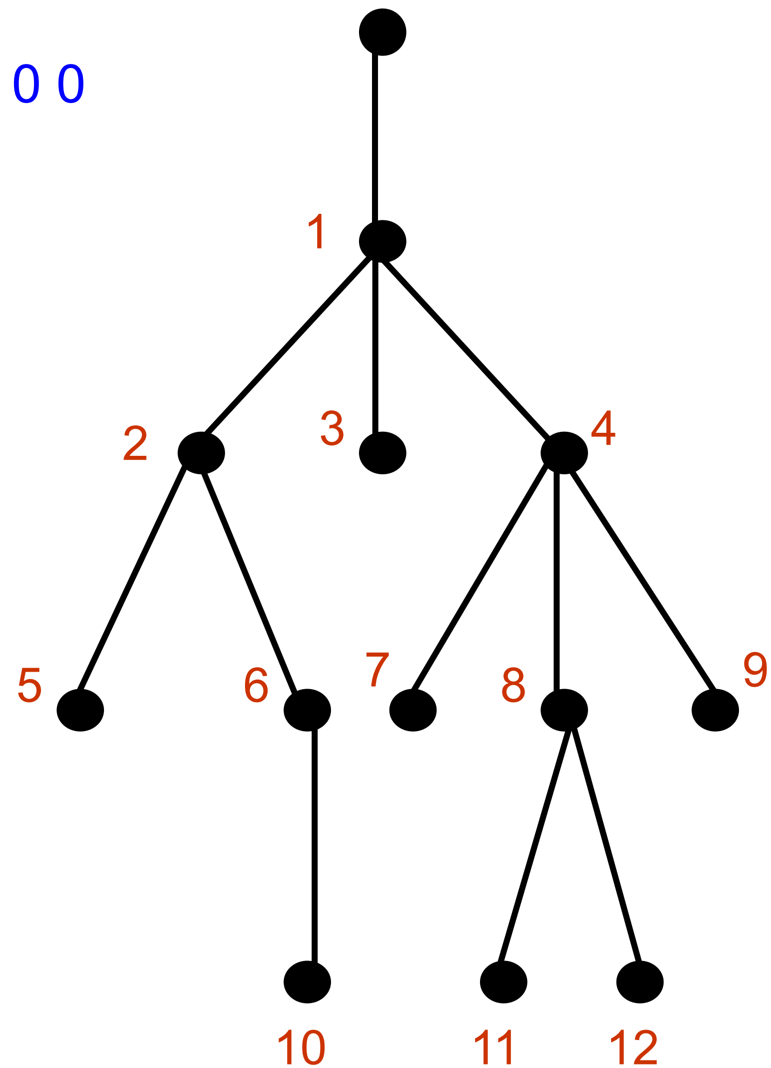
node k corresponds to the
k-th 1 in the bit sequence

parent(k) = # 0's up to the k-th 1

children of k are stored after the k-th 0

supports: parent, i-th child, degree

(using rank and select)



Simple FM-index

- Construct the *Burrows-Wheeler-transformed* text bwt(T) [BW94].
- From bwt(T) it is possible to construct the suffix array sa(T) of T in linear time.
- Instead of constructing the whole sa(T), one can add small data structures besides bwt(T) to simulate a search from sa(T).

Burrows-Wheeler transformation

- Construct a matrix M that contains as rows all rotations of T.
- Sort the rows in the lexicographic order.
- Let L be the last column and F be the first column.
- bwt(T)=L associated with the row number of T in the sorted M.

Example

```
pos  123456789
T  = kalevala#
```

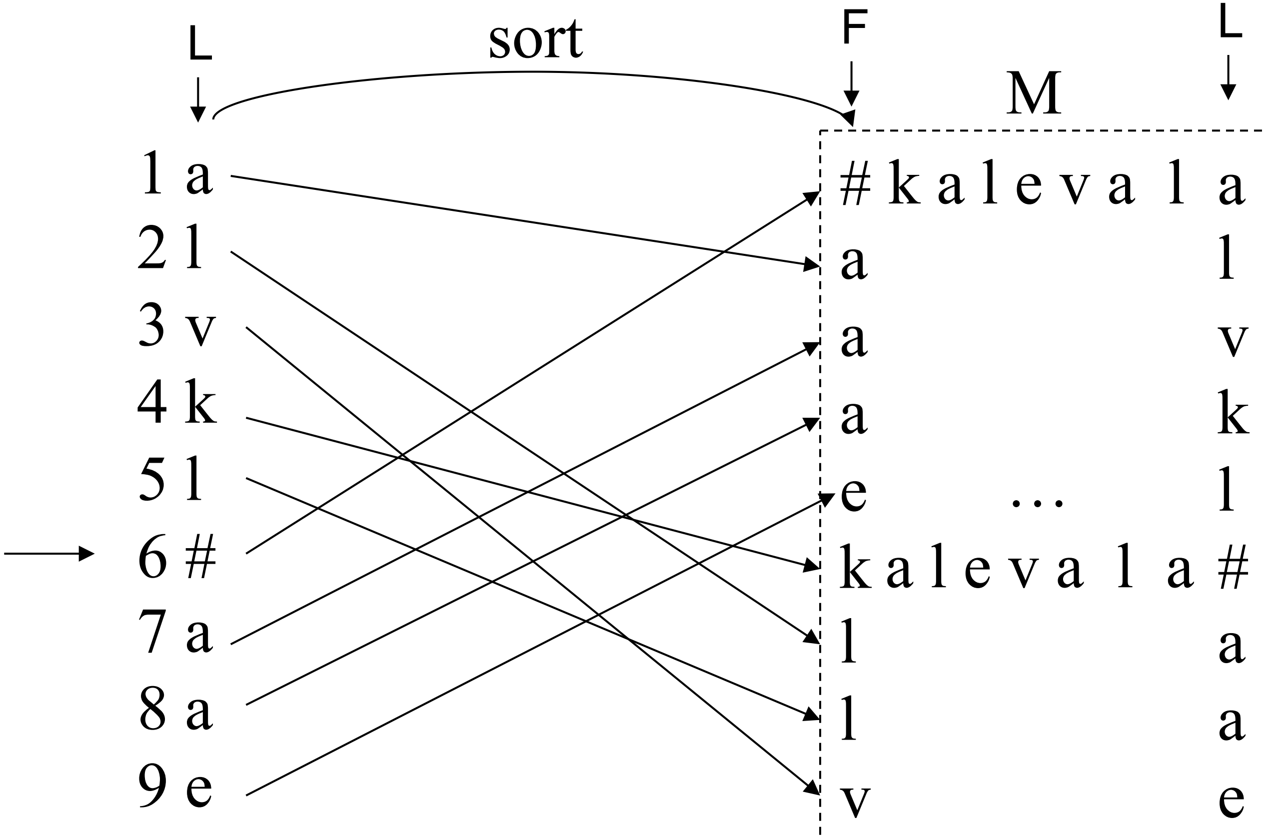
sa	F ↓ M L ↓
1:9	#kalevala
2:8	a#kaleval
3:6	ala#kalev
4:2	alevala#k
5:4	evala#kal
6:1	kalevala#
7:7	la#kaleva
8:3	levala#ka
9:5	vala#kale

==>

L = alvkl#aae, row 6

Exercise: Given L and the row number, we know how to compute T. What about sa(T)?

T^{-1}= # a l a v e l a k



sa(T)
1: 9
2: 8
3: 6
4: 2
5: 4
6: 1
7: 7
8: 3
9: 5

i	1	2	3	4	5	6	7	8	9
LF[i]	2	7	9	6	8	1	3	4	5

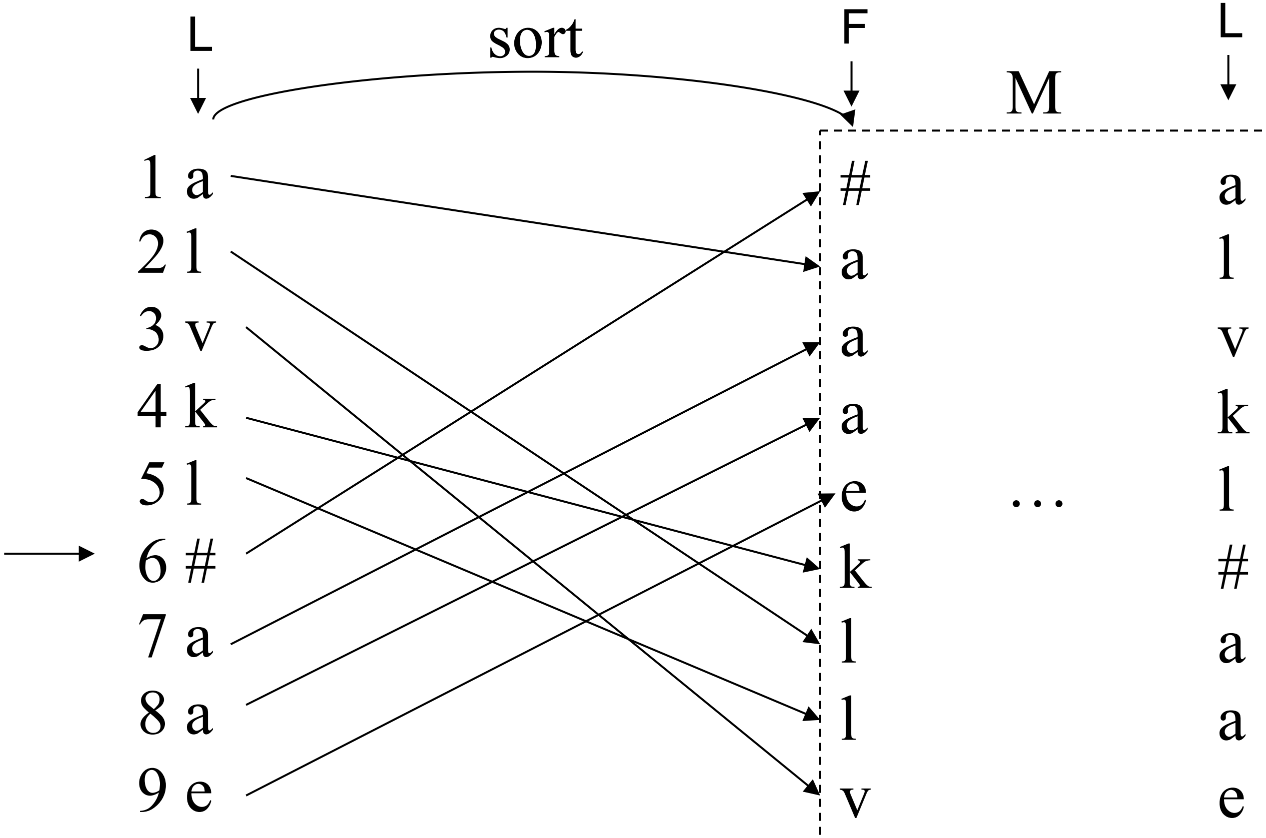
Implicit LF[i]

- Ferragina and Manzini (2000) noticed the following connection:
- $LF[i]=C_T[L[i]]+rank_{L[i]}(L,i)$
  - $C_T[c]$:
    - amount of letters $0,1,\ldots,c-1$ in $L=bwt(T)$
  - $rank_c(L,i)$:
    - amount of letters c in the prefix $L[1,i]$

Rank/Select

$select_1(L,j)$	3 6 9 10 12
L	001001001101
$rank_1(L,i)$	001112223445

T^{-1}= # a l a v e l a k



sa(T)
1: 9
2: 8
3: 6
4: 2
5: 4
6: 1
7: 7
8: 3
9: 5

i	1	2	3	4	5	6	7	8	9
LF[i]	2	7	9	6	8	1	3	4	5

$LF[7]=C_T[a]+rank_a(L,7)$
$=1+2=3$

Recall: Backward search on BWT(T)

- **Observation**: If [i,j] is the range of rows of M that start with string X, then the range [i',j'] containing cX can be computed as

$$i' := C_T[c]+\text{rank}_c(L,i-1)+1,$$
$$j' := C_T[c]+\text{rank}_c(L,j).$$

Backward search on BWT(T)...

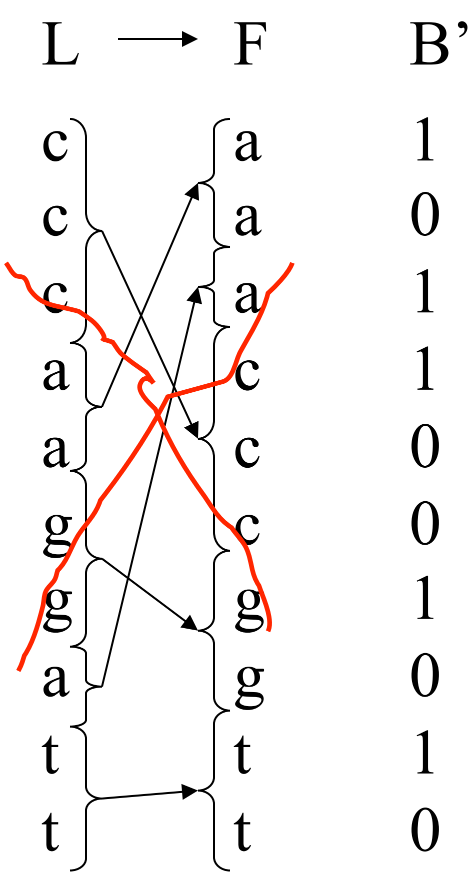
- Array $C_T[1,\sigma]$ takes $O(\sigma \log |T|)$ bits.
- $L=Bwt(T)$ takes $O(|T| \log \sigma)$ bits.
- Assuming $rank_c(L,i)$ can be computed in constant time for each (c,i), the algorithm takes $O(|P|)$ time to count the occurrences of P in T.

Run-Length FM-index

- We make the following changes to the previous FM-index variant:
  - $L=Bwt(T)$ is replaced by a sequence $S[1,n']$ and two bit-vectors $B[1,|T|]$ and $B'[1,|T|]$,
  - Cumulative array $C_T[1,c]$ is replaced by $C_S[1,c]$,
  - wavelet tree is build on S, and
  - some formulas are changed.

Run-Length FM-index…

L	B	S
c	1	c
c	0	a
c	0	g
a	1	a
a	0	t
g	1	
g	0	
a	1	
t	1	
t	0	



Changes to formulas

- Recall that we need to compute $C_T[c]+rank_c(L,i)$ in the backward search.
- **Theorem:** $C[c]+rank_c(L,i)$ is equivalent to
  - $select_1(B',C_S[c]+1+rank_c(S,rank_1(B,i)))-1$, when $L[i] \neq c$,
  - $select_1(B',C_S[c]+rank_c(S,rank_1(B,i)))+ i-select_1(B,rank_1(B,i))$, otherwise

EXAMPLE, L[I]=C

L	F	B	S	B'
c	a	1	c	1
c	a	0	a	0
c	a	0	g	1
a	c	1	a	1
a	c	0	t	0
g	c	1		0
g	g	0		1
a	g	1		0
t	t	1		1
t	t	0		0

$LF[8] = select_1(B', C_S[a] + rank_a(S, rank_1(B,8))) + 8 - select_1(B, rank_1(B,8))$

$= select_1(B', 0 + rank_a(S,4)) + 8 - select_1(B,4)$

$= select_1(B', 0+2) + 8 - 8$

$= 3$

- For more detail, read the original paper

Exercise

- ipsm$pisi
- 111011111010

WHAT IS B'

i	**B**	**S**
1	1	i
2	1	p
3	1	s
4	0	
5	1	m
6	1	$
7	1	p
8	1	i
9	1	s
10	0	
11	1	i
12	0	

USUALLY B' IS GIVEN TO SAVE COMPUTATIONS

i	**B**	**S**	**B'**
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

Reverse BWT from row 6

i	B	S	B'
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

Reverse BWT

i	**B**	**S**	**B'**
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

$S[rank_1(B, 6)] = \$$

Reverse BWT

i	B	S	B'
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

$S[rank_1(B, 6)] = \$$

LF[6]

$= select_1(B', C_S[\$] + rank_\$(S, rank_1(B, 6))) + 6 - select_1(B, rank_1(B, 6)))$

$= select_1(B', 0 + rank_\$(S, 5)) + 6 - select_1(B\ 5)$

$= 1 + 6 - 6 = 1$

Reverse BWT

i	B	S	B'
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

$S[\text{rank}_1(B, 1)] = i$

$LF[1]$

$= \text{select}_1(B', C_S[i] + \text{rank}_i(S, \text{rank}_1(B, 1))) + 1$

$- \text{select}_1(B, \text{rank}_1(B, 1)))$

$= \text{select}_1(B', 1 + \text{rank}_i(S, 1)) + 1 - \text{select}_1(B, 1)$

$= 2 + 1 - 1 = 2$

REVERSE BWT

i	B	S	B'
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

$S[rank_1(B, 1)] = i$

LF[1]

$= select_1(B', C_S[i] + rank_i(S, rank_1(B, 1))) + 1$

$- select_1(B, rank_1(B, 1)))$

$= select_1(B', 1 + rank_i(S, 1)) + 1 - select_1(B, 1)$

$= 2 + 1 - 1 = 2$

You can also construct the SA in this way:

12, 11, ….

12,11,8,5,2,1,10,9,7,4,6,3

BACKWARD SEARCH

i	B	S	B'
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

Suppose search for si:

c = i, First = 2, Last = 5

c = s

First = C[c] + Occ(c, First – 1) + 1

Last = C[c] + Occ(c, Last)

BACKWARD SEARCH

i	B	S	B'
1	1	i	1
2	1	p	1
3	1	s	1
4	0		1
5	1	m	0
6	1	$	1
7	1	p	1
8	1	i	1
9	1	s	1
10	0		0
11	1	i	1
12	0		0

c = i, First = 2, Last = 5

c = s

First = $select_1$(B', C_S[s]+1+$rank_s$(S, $rank_1$(B, 2-1))) -1 + 1

=$select_1$(B',7+1+$rank_s$(S,1))

=$select_1$(B', 8) = 9

Last = $select_1$(B', C_S[s]+1+$rank_s$(S, $rank_1$(B,5))) -1

=$select_1$(B',7+1+$rank_s$(S,4)) – 1

=$select_1$(B', 9) -1 = 11 – 1 = 10