String Search using Compressed Indexes

Simon Gog and Andrew Turpin

Department of Computing and Information Systems
The University of Melbourne

Melbourne, May 7th 2013

Lecture objectives

At the end of this lecture you should:

- know what the string searching problem is
- know what the suffix array data structure is
- understand the connection between the *BWT* and suffix array
- know how to do backward search in the *BWT*
- know what a wavelet tree is and how it implements *rank*()

Problem Definition

Given a string T and a pattern P over an alphabet Σ of constant size σ. Let $n = |T|$ be the length of T, and $m = |P|$ be the length of P and $n \gg m$.

Example

$T = abracadabrabarbara\$$
$P = bar$
$\Sigma = \{\$, a, b, c, d, r\}, \sigma = 6, n = 18, m = 3$

Problem: String search

- Does P occur in T? (Existence query)
- How often does P occur in T? (Count query)
- Where does P occur in T? (Locate query)

Simple Solutions

- Check for each i in $i \in \{0, ..., n-m-1\}$ if $T[i..i+m-1] = P$.

```
1   for (size_t i=0; i<n; ++i){
2     bool match=true;
3     for (size_t j=0; j<m && match; ++j)
4       match = (i+j < n && T[i+j] == P[j]);
5     if (match)
6       return true;
7   }
8   return false;
```

- Time complexity: $\mathcal{O}(n \cdot m)$ comparisons

Improved solution

- Knuth, Morris, and Pratt precomputed a table of size m which allows to shift the pattern by possibly more than one position in case of a mismatch and get complexity: $\mathcal{O}(n+m)$
- This solution is optimal in the online scenario, in which we are not allowed to pre-process T (online scenario), but not in ...

our scenario

We are allowed to pre-compute an index structure I for T and use I for the string search.

- I should be small
- Time complexity of matching independent of n

First attempt: Suffix Arrays (1)

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
18	18	$abracadabrabarbara
17	17	a$abracadabrabarbar
10	10	abarbara$abracadabr
7	7	abrabarbara$abracad
0	0	abracadabrabarbara$
3	3	acadabrabarbara$abr
5	5	adabrabarbara$abrac
15	15	ara$abracadabrabarb
12	12	arbara$abracadabrab
14	14	bara$abracadabrabar
11	11	barbara$abracadabra
8	8	brabarbara$abracada
1	1	bracadabrabarbara$a
4	4	cadabrabarbara$abra
6	6	dabrabarbara$abraca
16	16	ra$abracadabrabarba
9	9	rabarbara$abracadab
2	2	racadabrabarbara$ab
13	13	rbara$abracadabraba

- First sort suffixes of T. (quicksort: $\mathcal{O}(n^2 \log n)$, best algorithms: $\mathcal{O}(n)$)
- Storing all suffixes takes $n^2 \log \sigma$ bits space. Only store starting positions of suffixes in SA ($n \log n$ bits).
- Question: How fast can we search using T and SA?

First attempt: Suffix Arrays (2)

- The suffixes are *ordered* in SA. We can use *binary search*!
- Start with the empty string ϵ which matches all prefixes (i.e. the interval $[sp_0..ep_0] = [0..n-1]$) of suffixes in SA.
- Then use binary search to determine the interval $SA[sp_j..ep_j]$ in $SA[sp_{j-1}..ep_{j-1}]$ so that all suffixes start with $P[0..j-1]$ for all $j \in [1..m]$.
- P occurs in T if $[sp_m..ep_m]$ is not empty.
- If P occurs the count query can be answered by $ep_m - sp_m + 1$.
- Time complexity: $\mathcal{O}(m \cdot \log n)$, space $\mathcal{O}(n \log n + n \log \sigma)$

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]
- Step 2: *ba* interval [9..10]
- Step 2: *bar* interval [9..10]

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]
- Step 2: *ba* interval [9..10]
- Step 2: *bar* interval [9..10]

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]
- Step 2: *ba* interval [9..10]
- Step 2: *bar* interval [9..10]

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]
- Step 2: *ba* interval [9..10]
- Step 2: *bar* interval [9..10]

Second attempt: Using BWT

i	$SA[i]$	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	a	$abracadabrabarbara
1	17	r	a$abracadabrabarbar
2	10	r	abarbara$abracadabr
3	7	d	abrabarbara$abracad
4	0	$	abracadabrabarbara$
5	3	r	acadabrabarbara$abr
6	5	c	adabrabarbara$abrac
7	15	b	ara$abracadabrabarb
8	12	b	arbara$abracadabrab
9	14	r	bara$abracadabrabar
10	11	a	barbara$abracadabra
11	8	a	brabarbara$abracada
12	1	a	bracadabrabarbara$a
13	4	a	cadabrabarbara$abra
14	6	a	dabrabarbara$abraca
15	16	a	ra$abracadabrabarba
16	9	b	rabarbara$abracadab
17	2	b	racadabrabarbara$ab
18	13	a	rbara$abracadabraba

- $BWT[i] = T[SA[i]-1]$, for $SA[i] > 0$
- $BWT[i] = T[n-1]$, for $SA[i] = 0$
- I.e. $BWT[i]$ is the character preceding suffix $SA[i]$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

Add an array C containing the left border of each character interval:

$	a	b	c	d	r	r+1
0	1	9	13	14	15	19

- Operation $rank(i, X, BWT)$ returns how often character $X \in \Sigma$ occurs in the prefix $BWT[0..i-1]$.
- Now search backwards for *bar*.

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval:
  $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = C[r] + rank(sp_0, r, BWT)$
  $ep_1 = C[r] + rank(ep_0 + 1, r, BWT) - 1$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval:
  $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + rank(0, r, BWT)$
  $ep_1 = 15 + rank(19, r, BWT)$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval:
  $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + 0$
  $ep_1 = 15 + rank(19, r, BWT) - 1$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval:
  $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15+0 = 15$
  $ep_1 = 15+4-1 = 18$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]\,T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = C[a] + rank(sp_1, a, BWT)$
  $ep_2 = C[a] + rank(ep_1 + 1, a, BWT) - 1$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a, BWT)$
  $ep_2 = 1 + rank(ep_1, a, BWT)$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a, BWT)$
  $ep_2 = 1 + rank(ep_1, a, BWT)$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + 6$
  $ep_2 = 1 + rank(19, a, BWT) - 1$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1+6 = 7$
  $ep_2 = 1+8-1 = 8$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = C[b] + rank(sp_2 - 1, b, BWT)$
  $ep_3 = C[b] + rank(ep_2, b, BWT) - 1$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b, BWT)$
  $ep_3 = 9 + rank(ep_1, b, BWT)$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b, BWT)$
  $ep_3 = 9 + rank(ep_1, b, BWT)$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + 0$
  $ep_3 = 9 + rank(9, b, BWT) - 1$

Second attempt: Using BWT

i	BWT	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	a	$abracadabrabarbara
1	r	a$abracadabrabarbar
2	r	abarbara$abracadabr
3	d	abrabarbara$abracad
4	$	abracadabrabarbara$
5	r	acadabrabarbara$abr
6	c	adabrabarbara$abrac
7	b	ara$abracadabrabarb
8	b	arbara$abracadabrab
9	r	bara$abracadabrabar
10	a	barbara$abracadabra
11	a	brabarbara$abracada
12	a	bracadabrabarbara$a
13	a	cadabrabarbara$abra
14	a	dabrabarbara$abraca
15	a	ra$abracadabrabarba
16	b	rabarbara$abracadab
17	b	racadabrabarbara$ab
18	a	rbara$abracadabraba

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9+0 = 9$
  $ep_3 = 9+2-1 = 10$

Backward search summary

- Only C and a data structure R supporting the *rank* operation on *BWT* are required for existance and count queries.
- Space: $\sigma \log n$ bits for C + space for R
- Time: $\mathcal{O}(m \cdot t_{rank})$, where t_{rank} is time for one rank operation. Independent from n?
- Next: How can we implement *rank*?

Backward search summary

- Only C and a data structure R supporting the *rank* operation on *BWT* are required for existance and count queries.
- Space: $\sigma \log n$ bits for C + space for R
- Time: $\mathcal{O}(m \cdot t_{rank})$, where t_{rank} is time for one rank operation. Independent from n? If t_{rank} is independent from n
- Next: How can we implement *rank*?

Simple solution for rank (first attempt)

- Remember $rank(i, X, A)$ is the count of how many times X occurs in $A[0..i-1]$
- Store for each $i \in [0..n-1]$ and for each $X \in \Sigma$ the answer for $rank(i, X, BWT)$.
- Pro: Constant time per rank. $\mathcal{O}(m)$ for string search!
- Con: Too much space ($\mathcal{O}(\sigma n \log n)$ bits).

Solution for rank (second attempt)

- We can solve *rank* on a bit vector in constant time using $n + o(n)$ bits as follows:
- Divide bit vector in superblocks of size $sbz = \log^2 n$ bits.
- (1) Store all values $rank(k \cdot sbz, 1)$.
- Divide superblocks in blocks of size $bz = \frac{\log n}{2}$ bits.
- (2) Store all values $rank(k \cdot sbz + k' \cdot bz, 1) - rank(k \cdot sbz, 1)$.
- (3) Pre-compute all answers of rank for bitvectors of size $\log n/2$.

Total space: n bits for bitvector +

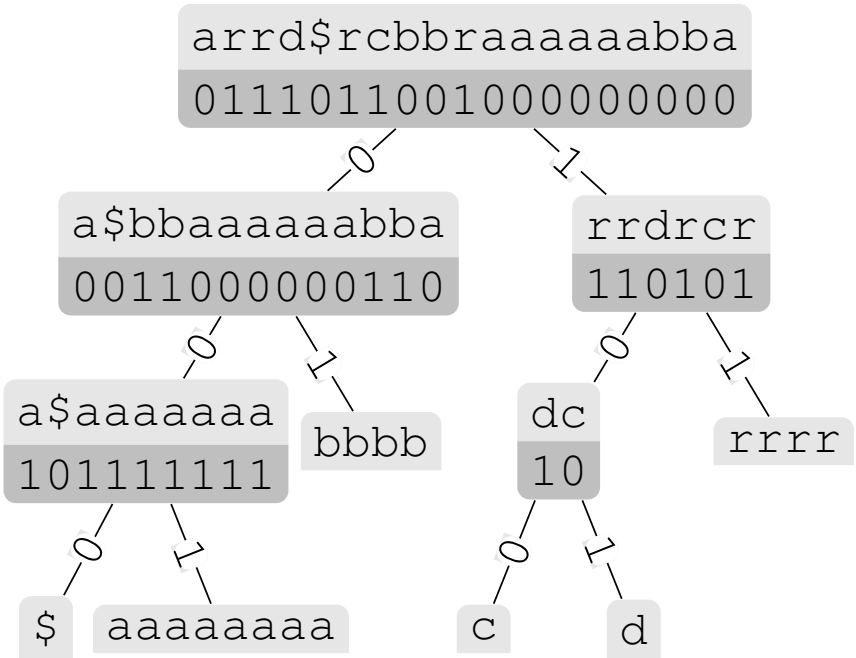
- (1) $n\lceil \frac{\log n}{\log^2 n} \rceil \in o(n)$ bits, (2) $n\lceil \frac{2 \log \log n}{\log n} \rceil \in o(n)$ bits
- (3) $2^{\log n/2} \log n/2 \log \log n/2 = \sqrt{n} \log n/2 \log \log n/2 \in o(n)$ bits

Solution for rank (second attempt)

Exercise: Implement the rank operation based on this schema.

Simple solution for rank (second attempt)

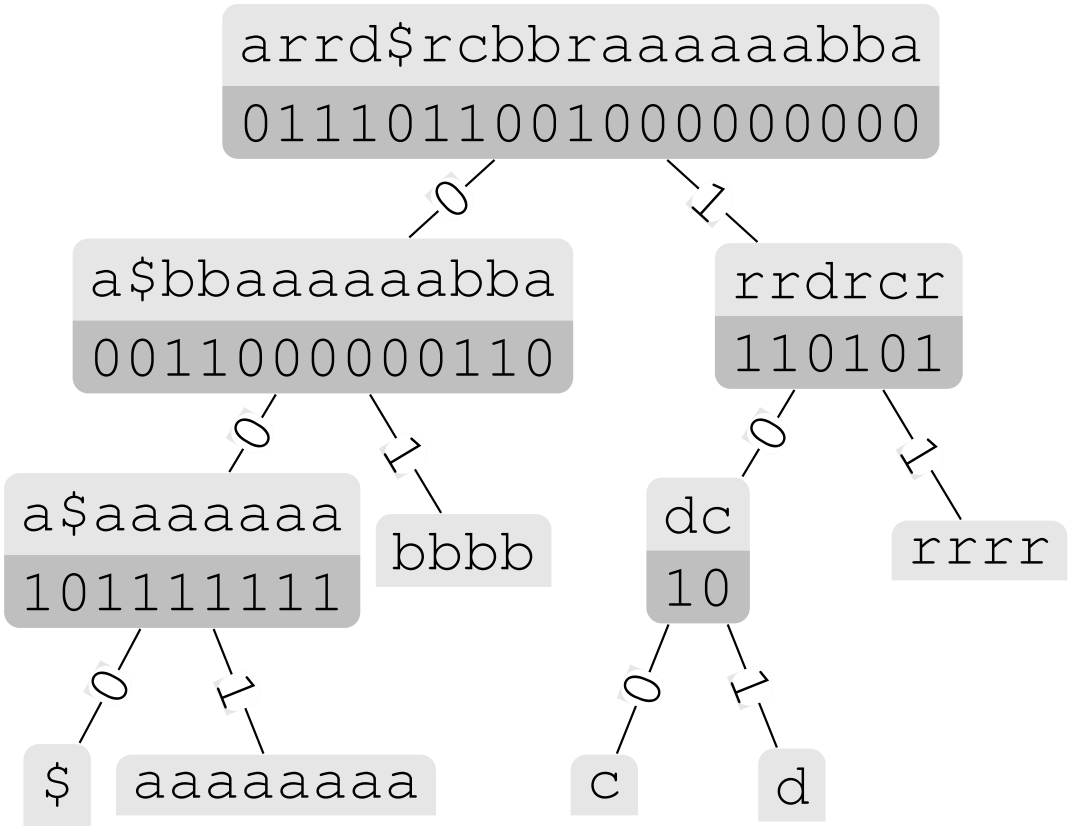
Use a wavelet tree to handle general alphabets:



Char	c	$codeword(c)$	freq
	$	000	1
	a	001	8
	b	01	4
	c	100	1
	d	101	1
	r	11	4

Depth: $\log \sigma$. Only bitvectors and pointers to bitvectors are stored.
Total space: $\approx n \log \sigma + 2\sigma \log n$

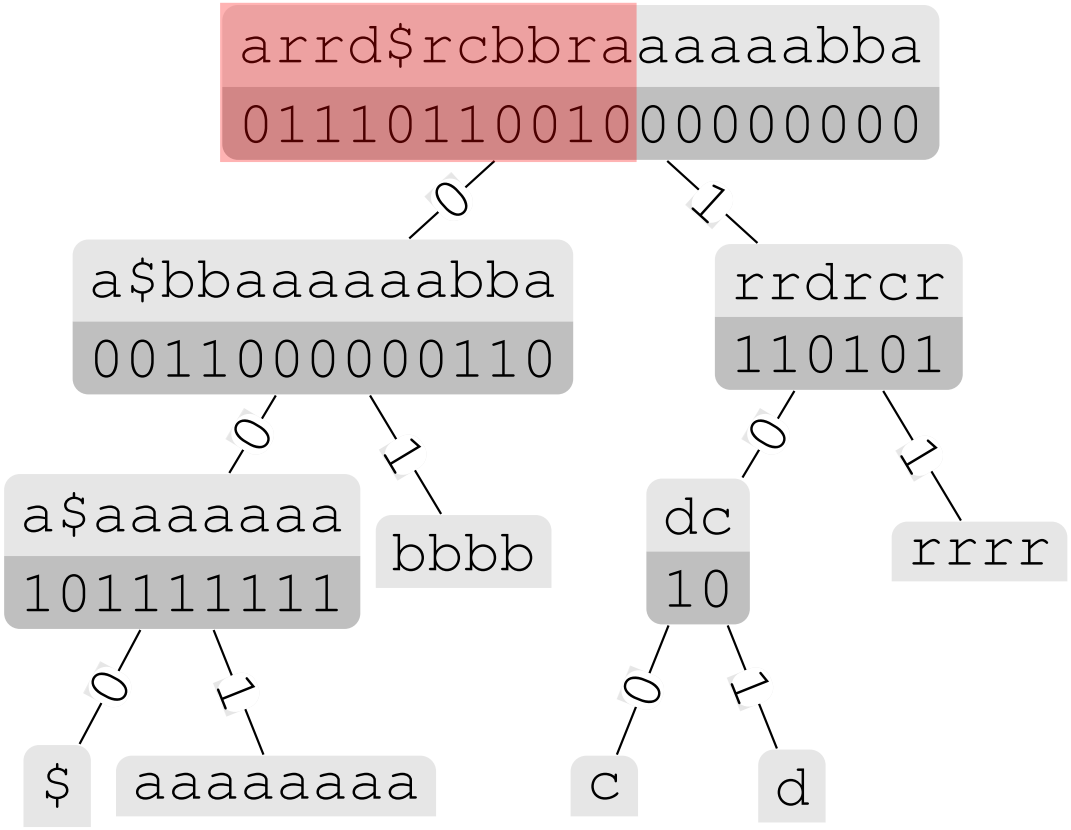
Wavelet Tree Example: Calculate Rank



a = 001

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

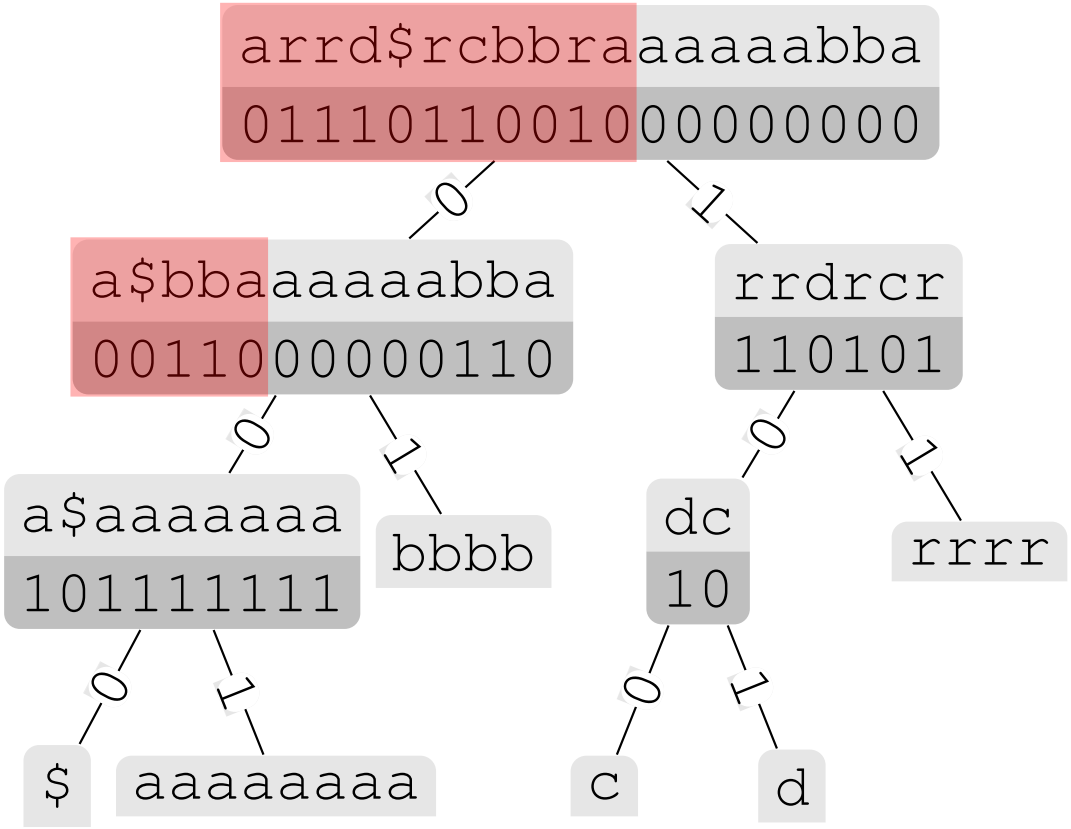
Wavelet Tree Example: Calculate Rank



a = 001

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

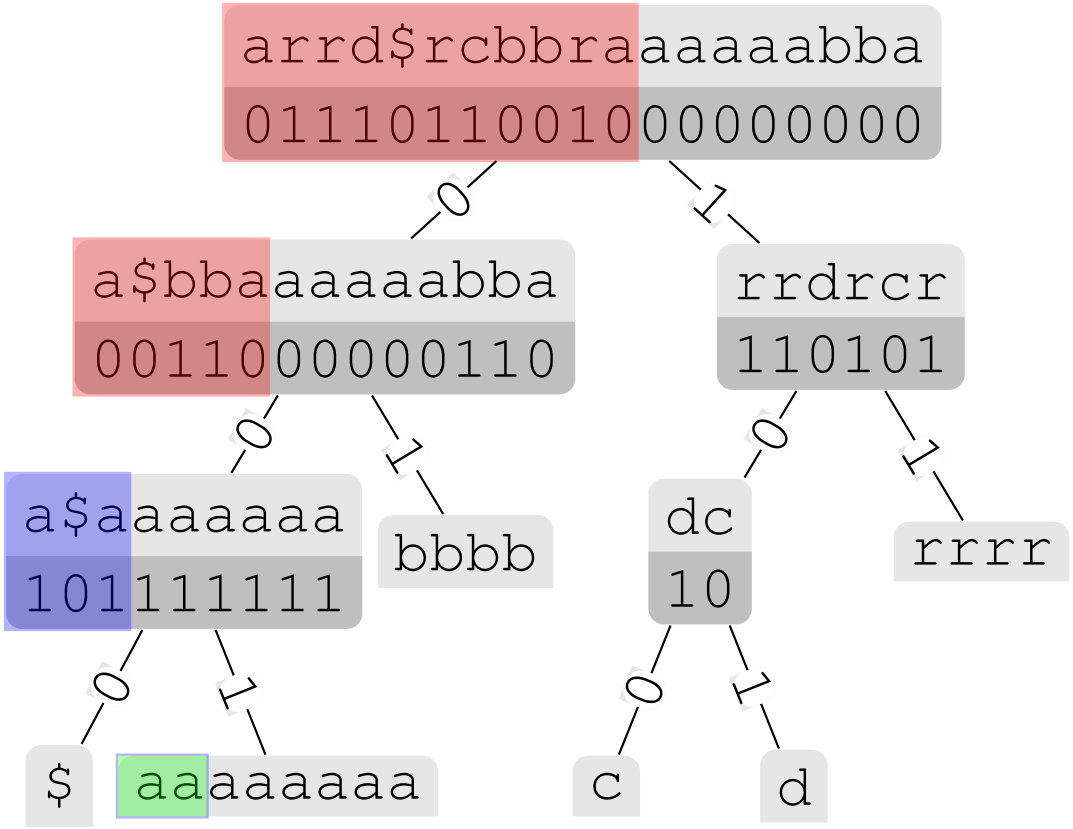
Wavelet Tree Example: Calculate Rank



a = 001

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

Wavelet Tree Example: Calculate Rank



a = 001

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

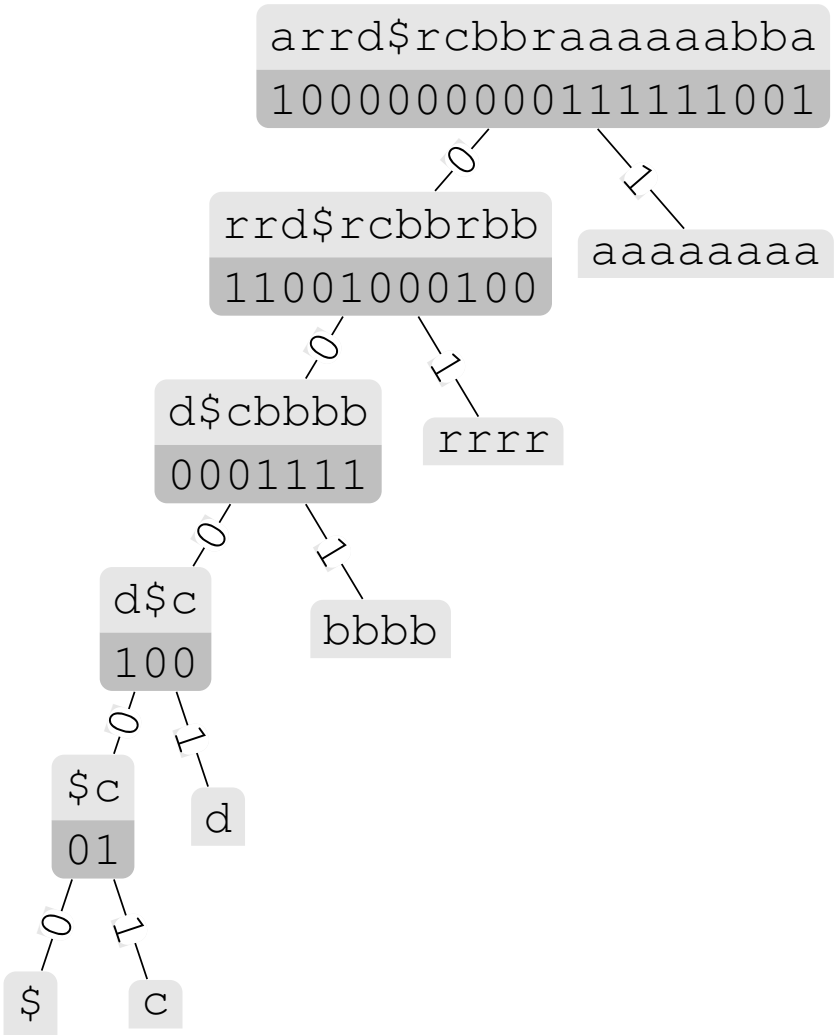
Pseudocode for rank on WT

$rank(i, c, WT)$

```
00  p ← b_ε
01  j ← 0
02  while not p! = codeword(c) do
03      if codeword(c)[j] = 0 then
04          i ← i − rank(i, 1, b_p)
05          p ← p0
06      else
07          i ← rank(i, 1, b_p)
08          p ← p1
09  return i
```

This code can also be used in a more space-efficient WT variant.

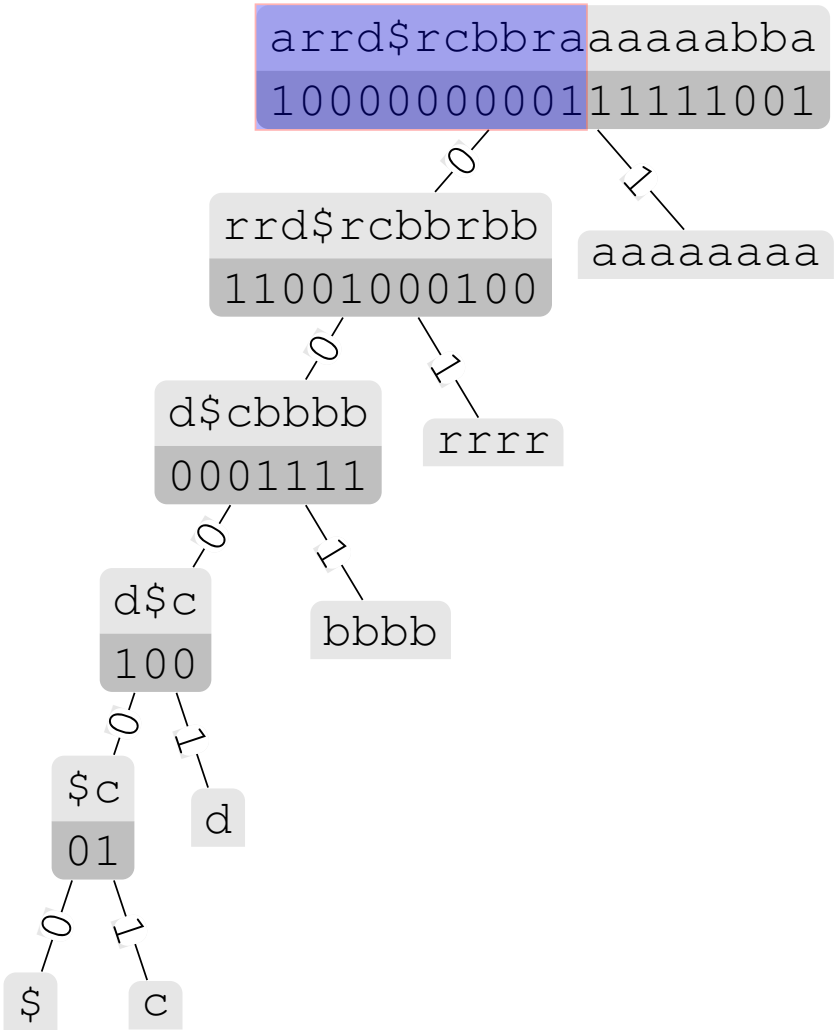
Huffman shaped wavelet tree



Char	c	$codeword(c)$
	$	00000
	a	1
	b	001
	c	00001
	d	0001
	r	01

Avg. depth: $H_0(BWT)$. Total space: $\approx nH_0 + 2\sigma \log n$

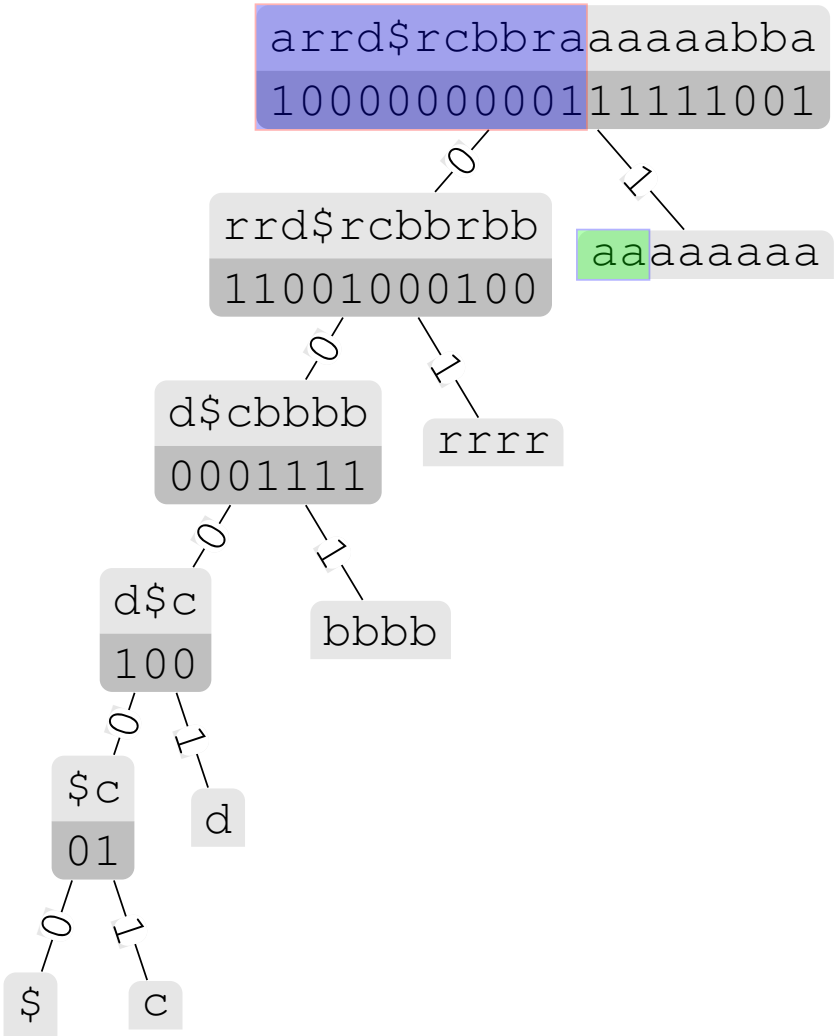
Huffman shaped wavelet tree



Char	c	$codeword(c)$
	$	00000
	a	1
	b	001
	c	00001
	d	0001
	r	01

$rank(11, a, WT) = rank(11, 1, b_\epsilon) = 2$

Huffman shaped wavelet tree



Char	c	$codeword(c)$
	$	00000
	a	1
	b	001
	c	00001
	d	0001
	r	01

$rank(11, a, WT) = rank(11, 1, b_\epsilon) = 2$

Summary

- Using a wavelet tree we can answer *rank*() in $\mathcal{O}(\log \sigma)$ time
- Using backward search in *BWT* we can count patterns in $\mathcal{O}(m \log \sigma)$ independent of n, the text size
- Space required for the index is just the wavelet tree and the C array: $\approx n \log \sigma + \sigma \log n$ bits or $\mathcal{O}(n + \sigma)$ words
- for locate queries we need the *SA* as well: same time, $n \log n$ additional bits.