Laboratorio de Bases de Datos
Facultade de Informática
Universidade da Coruña

Departamento de Ciencias de
la Computación
Universidad de Chile

Huffman-Compressed Wavelet Trees for Large Alphabets

Gonzalo Navarro (DCC)

Alberto Ordóñez (LBD)

WCTA 2012: SPIRE 2012 Workshop on Compression, Text, and Algorithms

Outline

- Introduction
- Compressing-permutations
- Compressing the mapping of Canonical Huffman shaped Wavelet Tree
- Removing pointers from Canonical Huffman WT
- Results

Introduction

- We usually build self-indexes over texts (or sequences with large alphabets) using:
  - An encoding for the indexed symbols (Canonical Huffman encoding, Hu-Tucker encoding, ...)
  - A WT to support operations

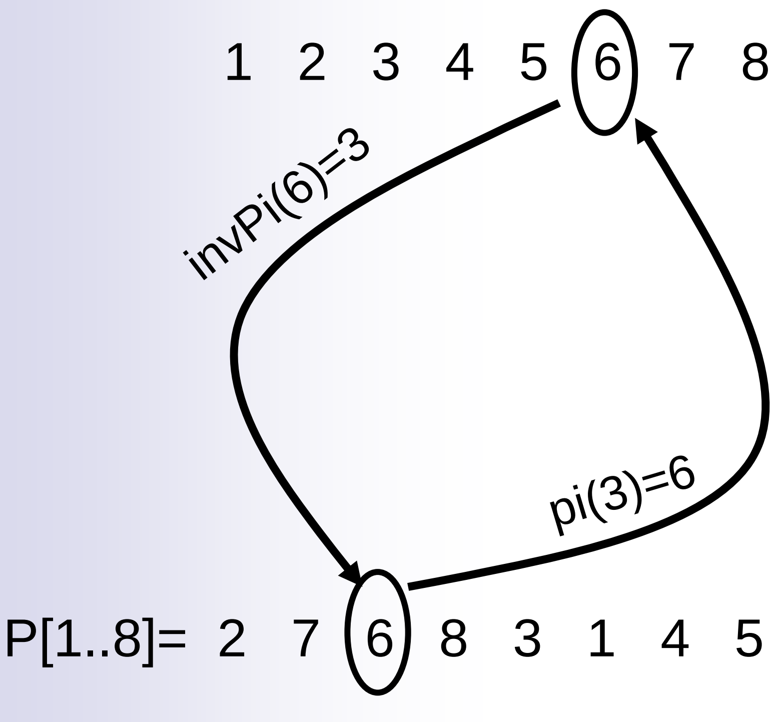
Introduction

- In this work we show:
  - How we can compress the mapping of a canonical Huffman encoding (mapping: code → symbol and symbol → code)
  - How we can reduce the size of a canonical Huffman WT without using pointers

Compressing the mapping of a canonical Huffman encoding

Compressing permutations

- Previous definitions



m = 4 increasing runs

- pi(i): returns symbol in **P**[**i**]
- invPi(i): returns the position in **P** where **i** is located

[STACS09] compresses **P** and access pi(i) and invPi(i) efficiently

Compressing permutations

- [STACS09]
  - *J. Barbay and G. Navarro: "Compressed Representations of Permutations, and Applications", Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS). Pp. 111-122 (2009)*

  - Lets consider a permutation **P[1..p]** with **m** increasing runs, then [STACS09] obtains a compressed representation for **P** that:
    - If we consider only the number of increasing runs **m**:
      - *p log m (1+o(1))+ O(m log p)* bits and solves pi(i) and invPi(i) in *O(log m)* time
    - If we conider the entropy of the runs (being Runs[1..m] a vector that contains the length of each run):
      - *n(2+H(Runs))(1+o(1))+O(m log n)* bits and solves pi(i) and invPi(i) in *O(H(Runs)+1)* time, H(Runs) <= log m

Compressing permutations

- [STACS09]
  - Example: building a compressed permutation using [STACS09]
  - Given a permutation P [1..8]=[2, 7, 6, 8, 1, 3, 4, 5] ,with m=4 increasing runs...
  - It recursively takes pairs of runs and merge them following a “merge sort” strategy

2 7 6 8 3 1 4 5

Compressing permutations

- [STACS09]
  - Example: building a compressed permutation using [STACS09]
  - Given a permutation P [1..8]=[2, 7, 6, 8, 1, 3, 4, 5] ,with m=4 increasing runs...
  - It recursively takes pairs of runs and merge them following a “merge sort” strategy

2 6 7 8 1 3 4 5

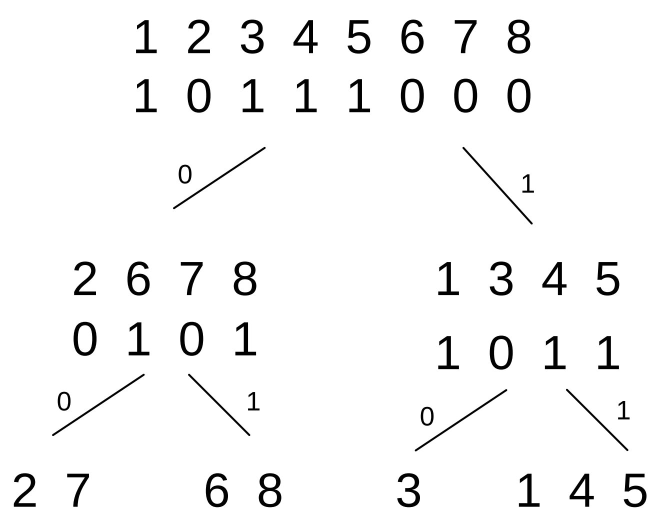
0 1 0 1 1 0 1 1

0 1 0 1

2 7 6 8 3 1 4 5

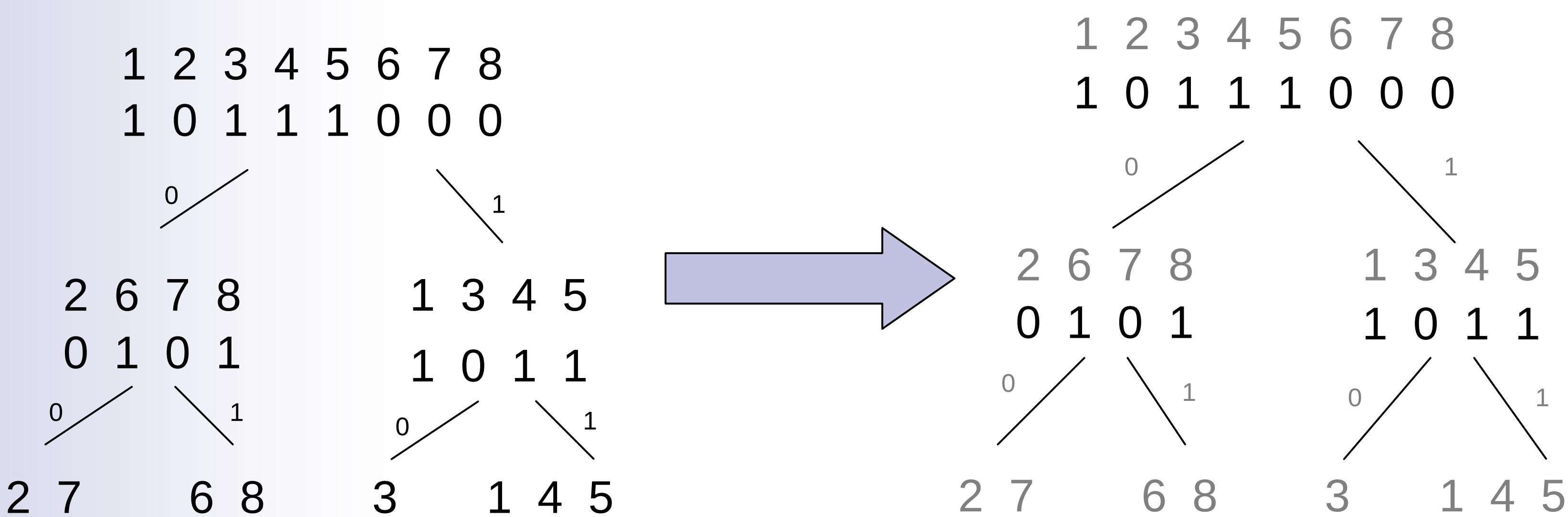
Compressing permutations

- [STACS09]
  - Example: building a compressed permutation using [STACS09]
  - Given a permutation P [1..8]=[2, 7, 6, 8, 1, 3, 4, 5] ,with m=4 increasing runs...
  - It recursively takes pairs of runs and merge them following a “merge sort” strategy



Compressing permutations

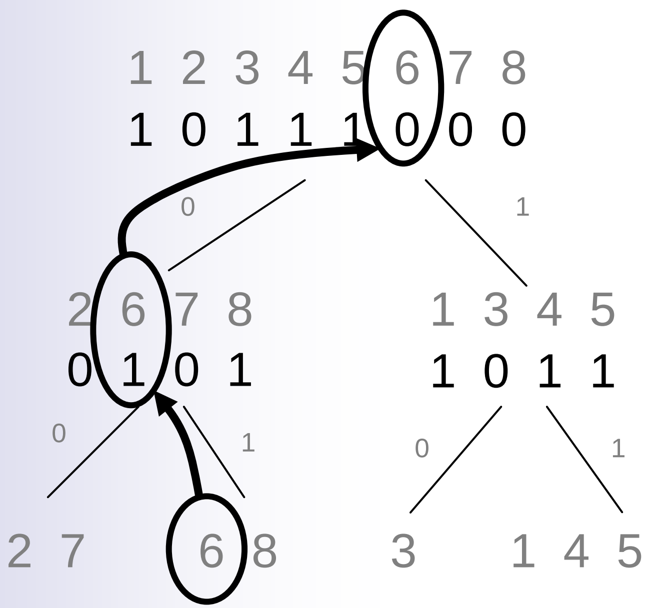
- [STACS09]
  - Example: building a compressed permutation using [STACS09]
  - Given a permutation P [1..8]=[2, 7, 6, 8, 1, 3, 4, 5] ,with m=4 increasing runs...
  - It recursively takes pairs of runs and merge them following a “merge sort” strategy



Shadowed are not stored

Compressing permutations

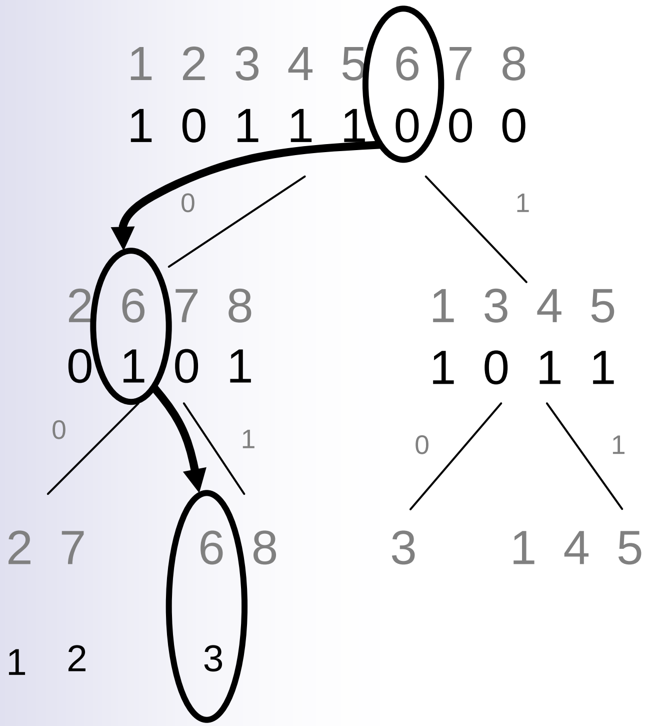
- [STACS09]
  - Example: operations over a compressed permutation with [STACS09]



- pi(3):
  - Locate position 3 in the leaves
  - Bottom-up transversal performing select
    - pi(3) = 6

Compressing permutations

- [STACS09]
  - Example: operations over a compressed permutation with [STACS09]



- invPi(6):
  - Top-down transversal of the tree performing rank
  - returns the offset
    - invPi(6) = 3

Compressing permutations

- [STACS09]
  - *J. Barbay and G. Navarro: "Compressed Representations of Permutations, and Applications", Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS). Pp. 111-122 (2009)*

  - We know that:
    - [STACS09] can solve pi(i) and invPi(i)
    - [STACS09] performs better when:
      - **The number of increasing runs is low (p log m (1+o1(1))…)**
      - **H(Runs) is low**

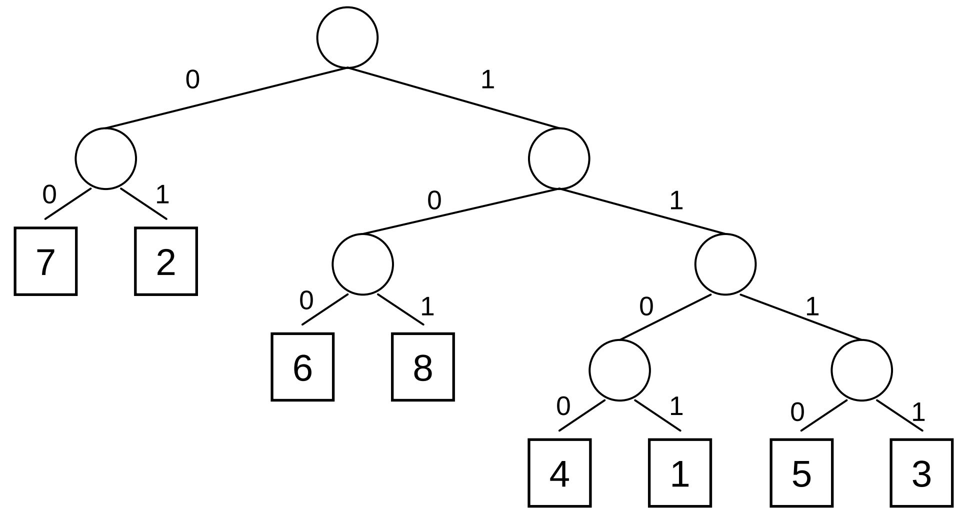
Compressing the mapping of Canonical Huffman shaped Wavelet Tree

- How we can use [STACS09] to reduce the size of a canonical Huffman mapping?

Mapping

1 → 1101
2 → 01
3 → 1111
4 → 1100
5 → 1110
6 → 100
7 → 00
8 → 101

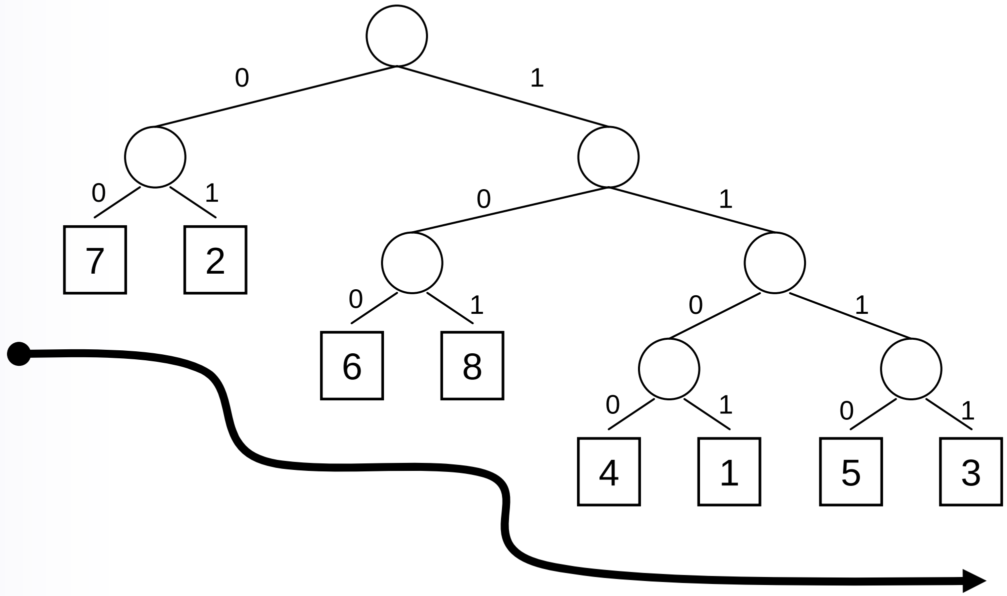
Canonical Huffman Tree



- Codes of the same length are consecutives
- **s** different symbols
- *O(log n)* is the max. Length of a Huffman code (being *n* the length of the indexed sequence)
  - Mapping takes
    - **O(s log n) bits**

Compressing the mapping of Canonical Huffman shaped Wavelet Tree

- How we can convert the mapping into a permutation?
  - Read Huffman tree leaves from left to right



P[1..8] = **7**, 2, 6, 8, **4**, 1, 5, **3**

with m = 5 increasing runs

Compressing the mapping of Canonical Huffman shaped Wavelet Tree

- Using [STACS09] we obtain better performance as the number of runs becomes smaller.
- How we can reduce the number of runs?

Compressing the mapping of Canonical Huffman shaped Wavelet Tree

- Using [STACS09] we obtain better performance as the number becomes smaller.

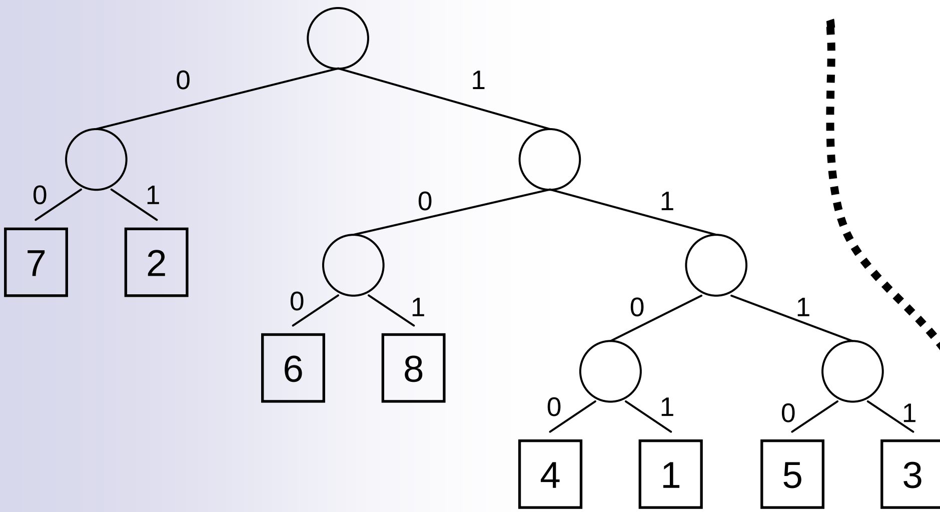
- How we can reduce the number of runs?

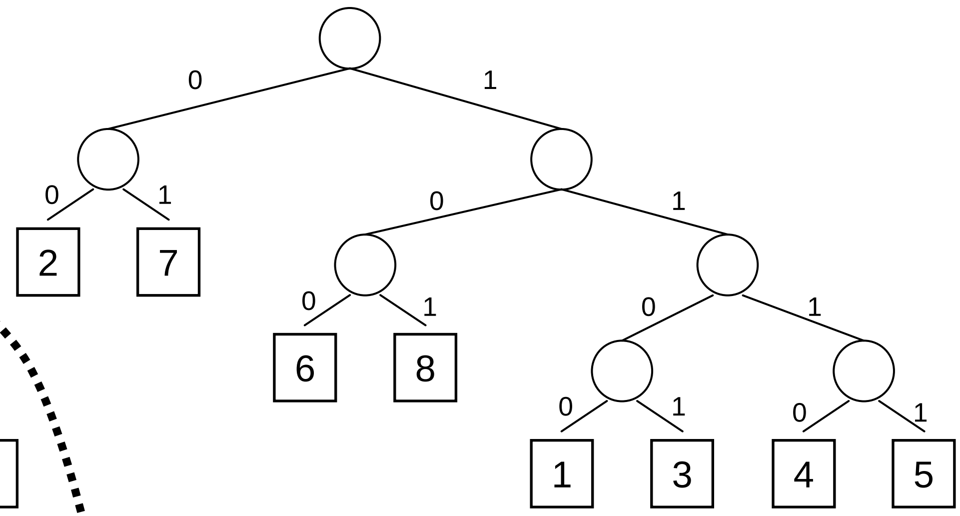
  - **KEY:** Huffman assigns a code-**length** to each symbol, NOT A CODE

Compressing the mapping of Canonical Huffman shaped Wavelet Tree

- Reducing the number of runs
  - Sort symbols in increasing order for each code length (for each Huffman tree level)

Both encodings are optimal





P[1..8] = **7**, 2, 6, 8, **4**, 1, 5, **3**

m = 5 increasing runs

P[1..8] = 2, 7, 6, 8, 1, 3, 4, 5

m = 3 increasing runs

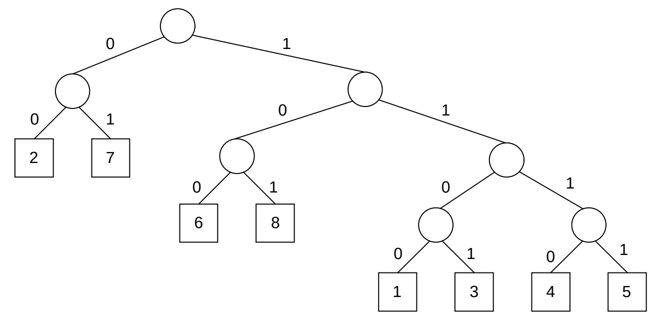
= Max. code length

Compressing the mapping of Canonical Huffman shaped Wavelet Tree

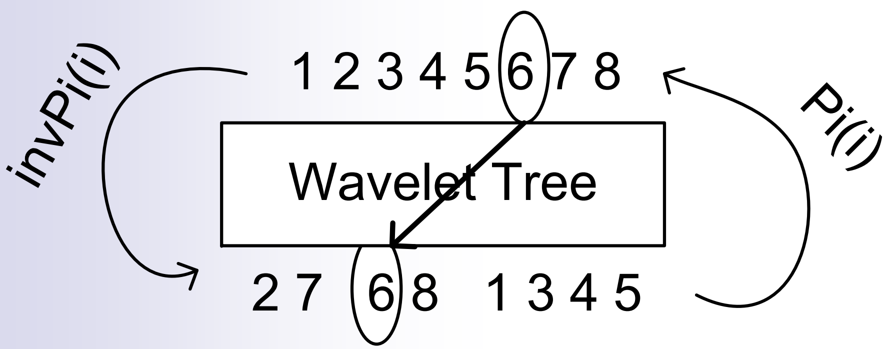
- Result:
  - As the maximum code length of a canonical Huffman encoding is $O(\log n)$ (the Huffman tree has $O(\log n)$ levels) we can obtain, reorganizing symbols at each level, at most $O(\log n)$ increasing runs. So:
    - Considering only the number of runs, we can compress the mapping from $O(s \log n)$ bits to $\mathbf{O(s \log \log n) + O(\log^2 n)}$ **bits** and solve symbol→code and code→symbol in $\mathbf{O(\log \log n)}$ **time** using [STACS09]
      - $O(\log^2 n)$ bits to:
        - Store where each run starts in P: iniRuns $O(\log s \log n)$
        - Store the first code of each level: C $(\log^2 n)$ (Codes of the same length are consecutives in canonical Huffman encoding)

Compressing the mapping of Canonical Huffman shaped Wavelet Tree

Canonical Huffman Tree (NOT STORED)



- Obtaining symbol→code
  - Applying invPi(symbol) we obtain:
    - the position **pos** in P of symbol and
    - the **run** where pos belongs
  - Return code = C[run] + pos – iniRuns[run]



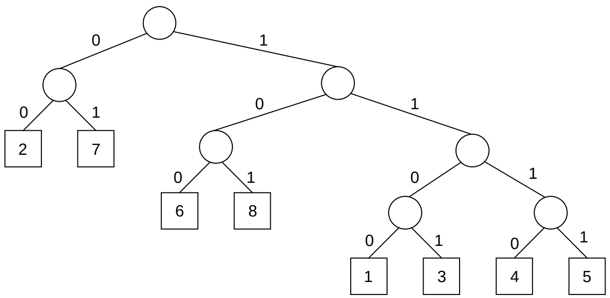
- invPi(6):
  - pos = 3, run = 2
- Code =

  C[2] + iniRuns[2] – pos =
  4 + 3 – 3 = 4 →

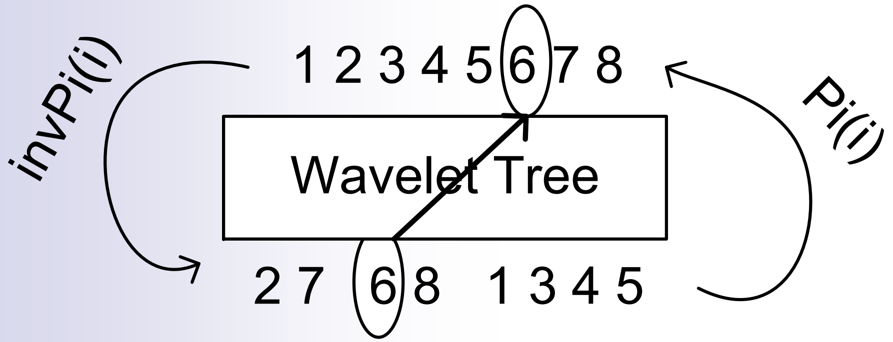
- Code = $4_{(10}$, $100_{(2}$

Compressing the mapping of Canonical Huffman shaped Wavelet Tree

Canonical Huffman Tree (NOT STORED)



- Obtaining (code, len)→symbol
  - Locate the position in P where code is located:
    - From len we can obtain the run (data structure that takes O (log n log log n) bits)
    - pos = iniRuns[run] + code – C[run]
  - Return symbol = pi(pos)



- (100, 3) → symbol?
  - Len 3 → run 2
- Pos = iniRuns[2] + 4 – C[2] = 3 + 4 – 4 = 3
- Apply pi(pos) = pi(3) = 6
- Code $100_{(2}$ → symbol 6

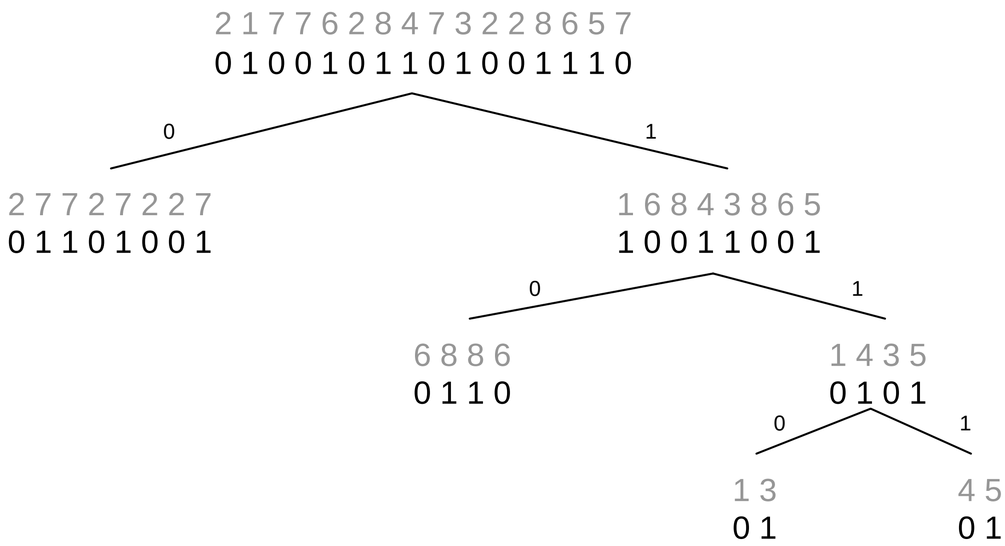
Removing pointers from a Canonical Huffman Wavelet Tree

Removing pointers from Canonical Huffman WT

- How to represent a canonical Huffman WT without using pointers?

Keys:

- Canonical Huffman implies that codes at the same level are consecutives

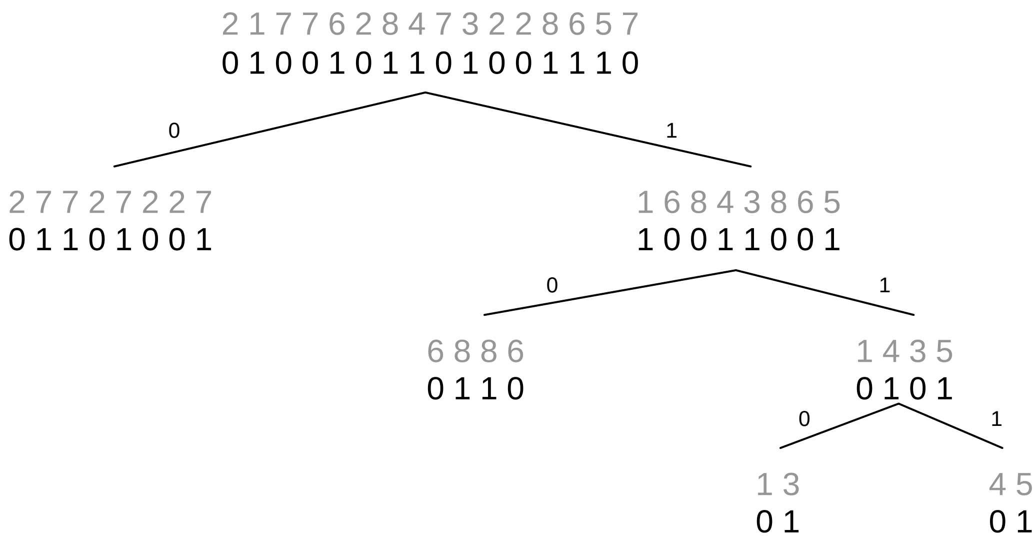


Removing pointers from Canonical Huffman WT

- How to represent a canonical Huffman WT without using pointers?

Keys:

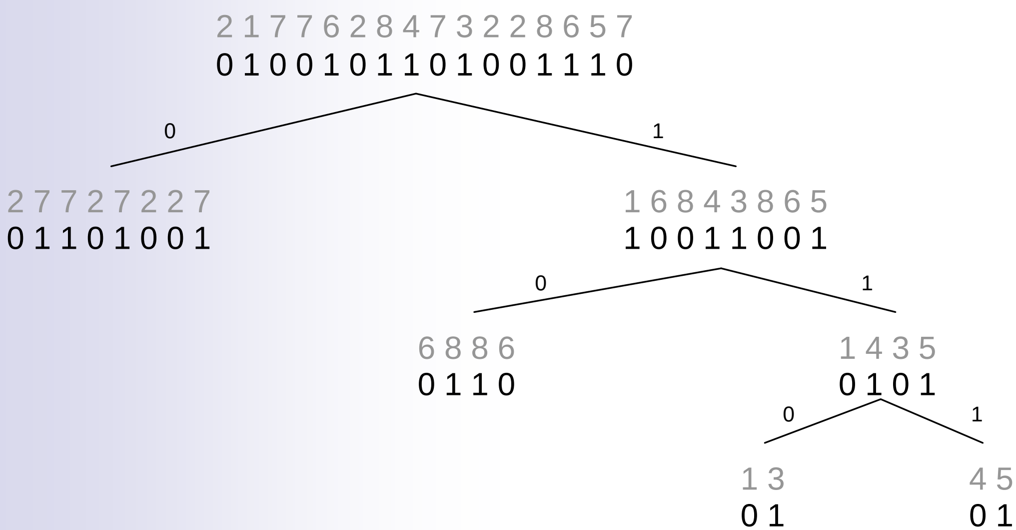
- Canonical Huffman implies that codes at the same level are consecutives
- Shortest codes are located in the left-most part of the WT



Removing pointers from Canonical Huffman WT

- Levelwise canonical Huffman WT

Canonical Huffman WT using pointers



Canonical Huffman WT without pointers

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

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

6 8 8 6 1 4 3 5
0 1 1 0 0 1 0 1

1 3 4 5
0 1 0 1

Removing pointers from Canonical Huffman WT

- Levelwise canonical Huffman WT

2 1 7 7 6 2 8 4 7 3 2 2 8 6 5 7
B_1= 0 1 0 0 1 0 1 1 0 1 0 0 1 1 1 0

2 7 7 2 7 2 2 7 1 6 8 4 3 8 6 5
B_2= 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 1

6 8 8 6 1 4 3 5
B_3= 0 1 1 0 0 1 0 1

1 3 4 5
B_4= 0 1 0 1

- F[i] = how many elements finish at level *i*
- C(i) = first code of each level
- N(i) = #codes per level

C[1]=0, C[2]=0, C[3]=100, C[4]=1100
N[1]=0, N[2]=2, N[3]=2, N[4]=4
F[1]=0, F[2]=8, F[3]=4, F[4]=4

i in [1..*O(log n)*]

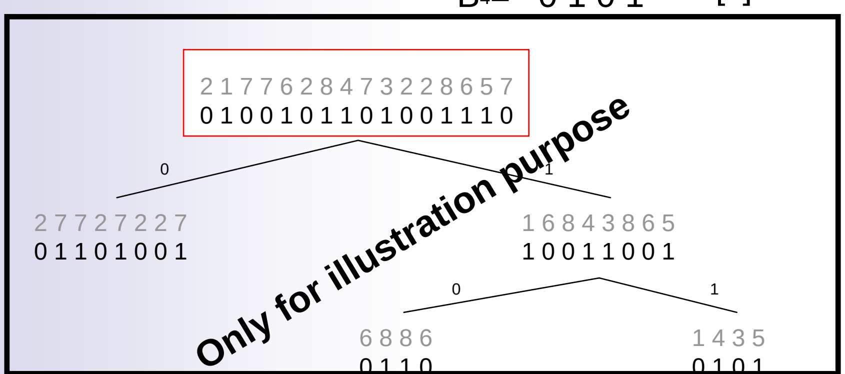
Removing pointers from Canonical Huffman WT

- Solving rank
  - $rank_3(12)$ = #3 up to position 12
    - Operations over on a WT turn into operations on bitmaps

s=0 e=16

$rank_1(12) = 5$

	2 1 7 7 6 2 8 4 7 3 2 2 8 6 5 7	
B_1=	0 **1** 0 0 **1** 0 **1** **1** 0 **1** 0 0 1 1 1 0	F[1]=0
	2 7 7 2 7 2 2 7 1 6 8 4 3 8 6 5	
B_2=	0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 1	F[2]=8
	6 8 8 6 1 4 3 5	
B_3=	0 1 1 0 0 1 0 1	F[3]=4
	1 3 4 5	
B_4=	0 1 0 1	F[4]=4



Symbol 3 → Code = **1**101, len=4

s = 0;

e = 16;

pos = 12;

Move to right:

$n_0 = rank_0(B_1,e) - rank_0(B_1,s) = 8 - 0 = 8;$

$s = s + n_0 - F[1] = 0 + 8 - 0 = 8;$

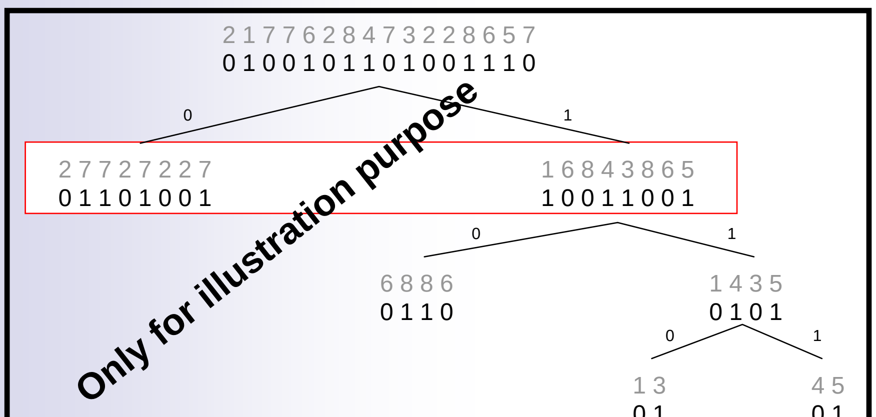
$pos = s + rank_1(12) - rank_1(s) = 8 + 5 - 0 = 13;$

Removing pointers from Canonical Huffman WT

- Solving rank
  - $rank_3(12)$ = #3 up to position 12
    - Operations over on a WT turn into operations on bitmaps

pos=13
s=8 e=16

	2 7 7 7 2 7 2 2 7 | 1 6 8 4 3 | 8 6 5		
B_2=	0 1 1 0 1 0 0 1 | 1 0 0 1 1 | 0 0 1	F[2]=8
	6 8 8 6 1 4 3 5		
B_3=	0 1 1 0 0 1 0 1	F[3]=4
	1 3 4 5		
B_4=	0 1 0 1	F[4]=4



Symbol 3 → Code = 1**1**01, len=4

s = 8;

e = 16;

pos = 13;

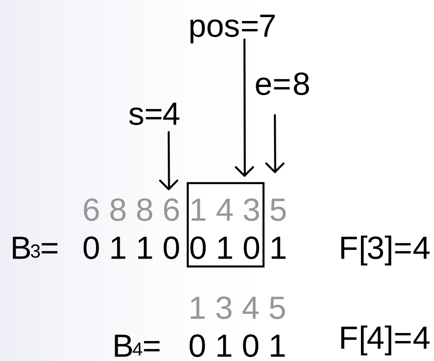
Move to right:

$n_0 = rank_0(B_2,e) - rank_0(B_2,s) = 8-4 = 4;$

$s = s + n_0 - F[2] = 8 + 4 - 8 = 4$

$pos = s + rank_1(pos) - rank_1(s) = 4 + 7 - 4 = 7;$

- Solving rank
  - $rank_3(12)$ = #3 up to position 12
    - Operations over on a WT turn into operations on bitmaps



Symbol 3 → Code = 11**0**1, len=4

s = 4;

e = 8;

pos = 7;

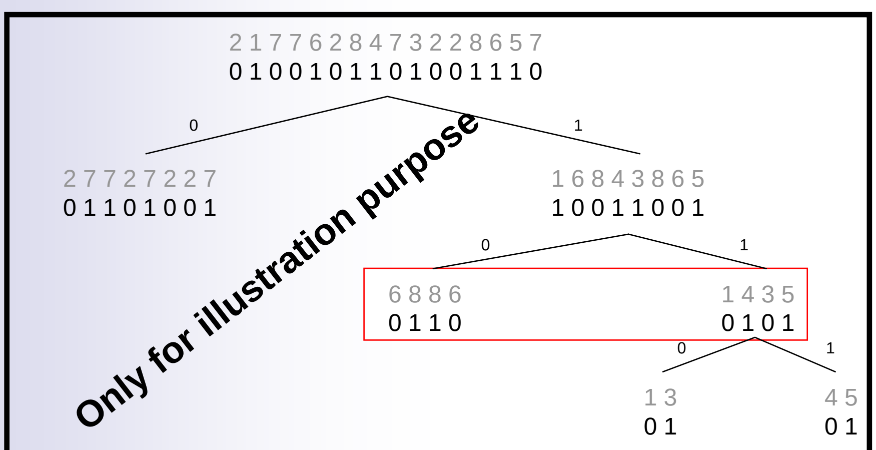
Move to left:

$n_0 = rank_0(B_3,e) - rank_0(B_3,s) = 4 - 0 = 4;$

$s = s - F[3] = 4 - 4 = 0;$

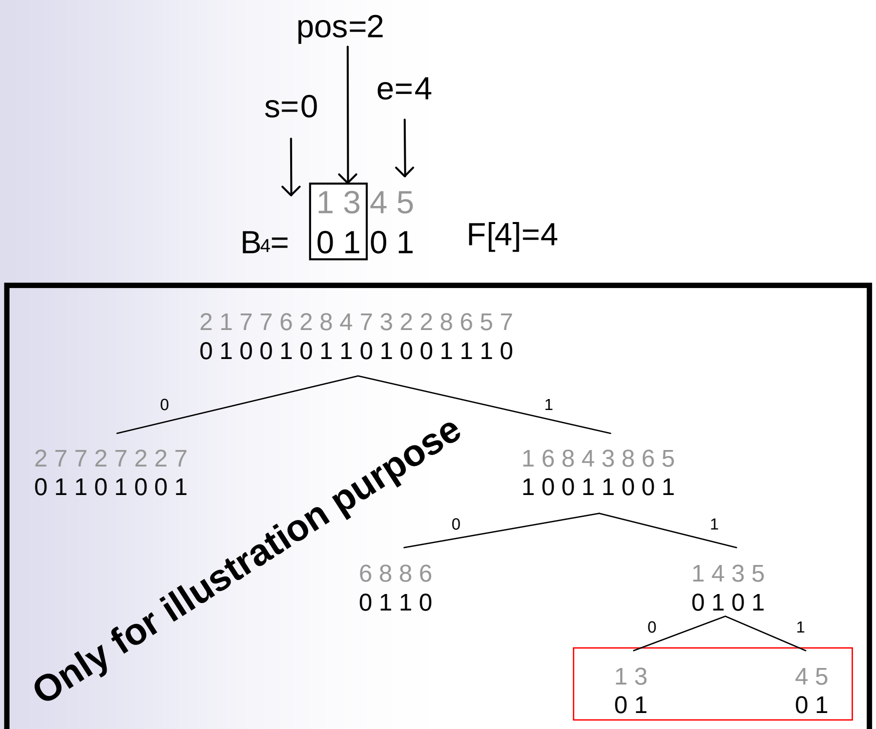
$e = s + n_0 = 0 + 4 = 4$

$pos = rank_0(pos) - rank_0(s) = 0 + 4 - 2 = 2$



Removing pointers from Canonical Huffman WT

- Solving rank
  - $rank_3(12)$ = #3 up to position 12
    - Operations over on a WT turn into operations on bitmaps



Symbol 3 → Code = 110**1**, len=4

s = 0;

e = 4;

pos = 2;

---

return

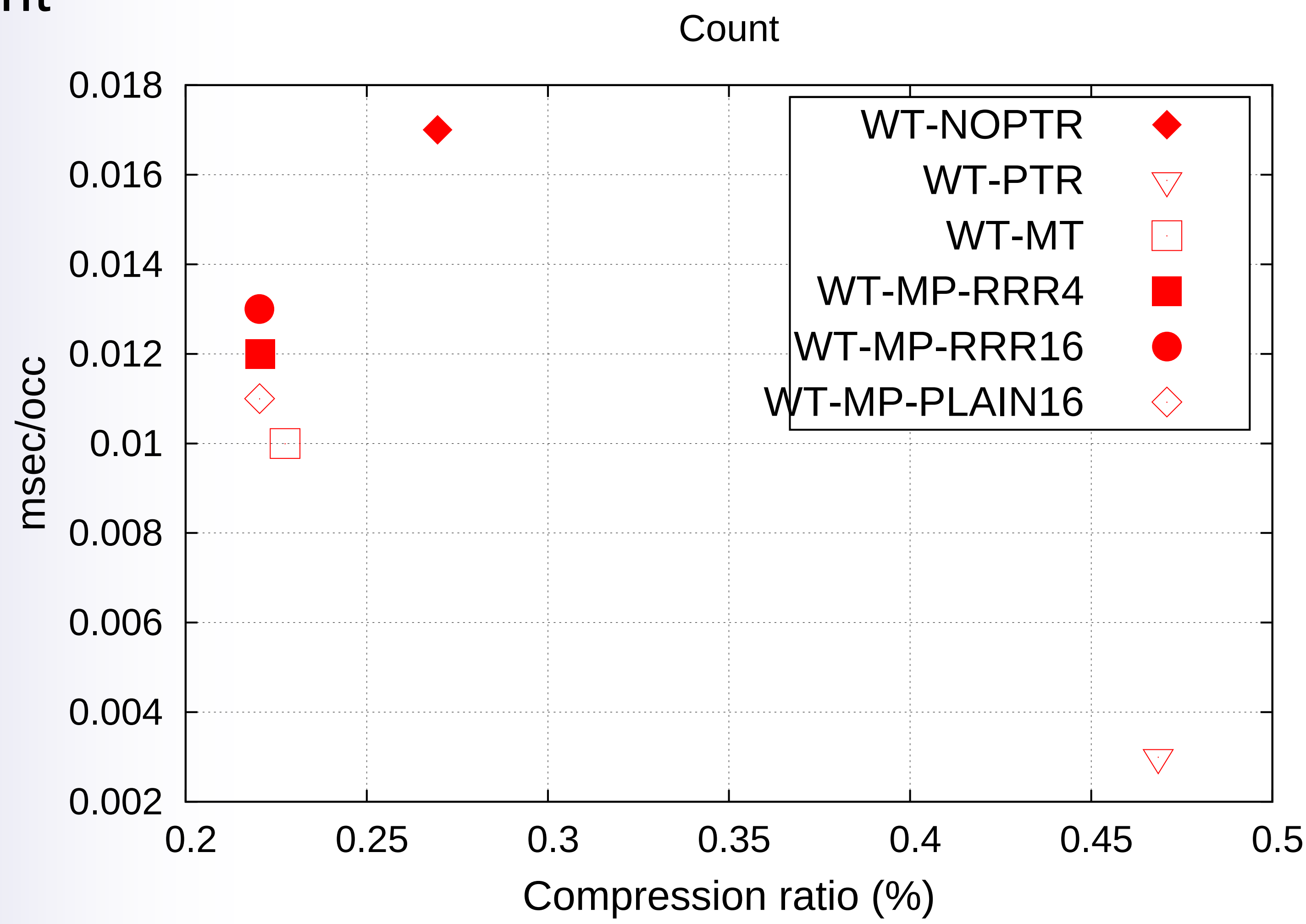
$rank_1(pos) - rank_1(s) = 1 - 0 = 1$

Experimental evaluation

- Set up
  - CR (from TREC)
  - Machine: Inter®Xeon®-E5446@2.00GHz with 16GB of RAM, Ubuntu 9.10. gcc 4.4.3 with flag –O9 set on.
  - Queries: count, select and access
  - Wavelet Trees:
    - Huffman Shaped WT with pointers: WT-PTR
    - Levelwise WT without pointers and without Huffman: WT-NOPTR
    - Levelwise Canonical Huffman WT without pointers with O(s log n) + O(s log s) bits to store the model and solve the mapping in O(1)
    - Levelwise Canonical Huffman WT without pointers that uses a permutation to compress the model: WT-MP
      - WT-MP-PLAIN#: WT-MP using uncompressed bitmaps. Sampling rate on bitmaps of #.
      - WT-PLAIN-RRR#: WT-MP using the Raman, Raman, and Rao technique to compress bitmaps. Sampling rate of #.

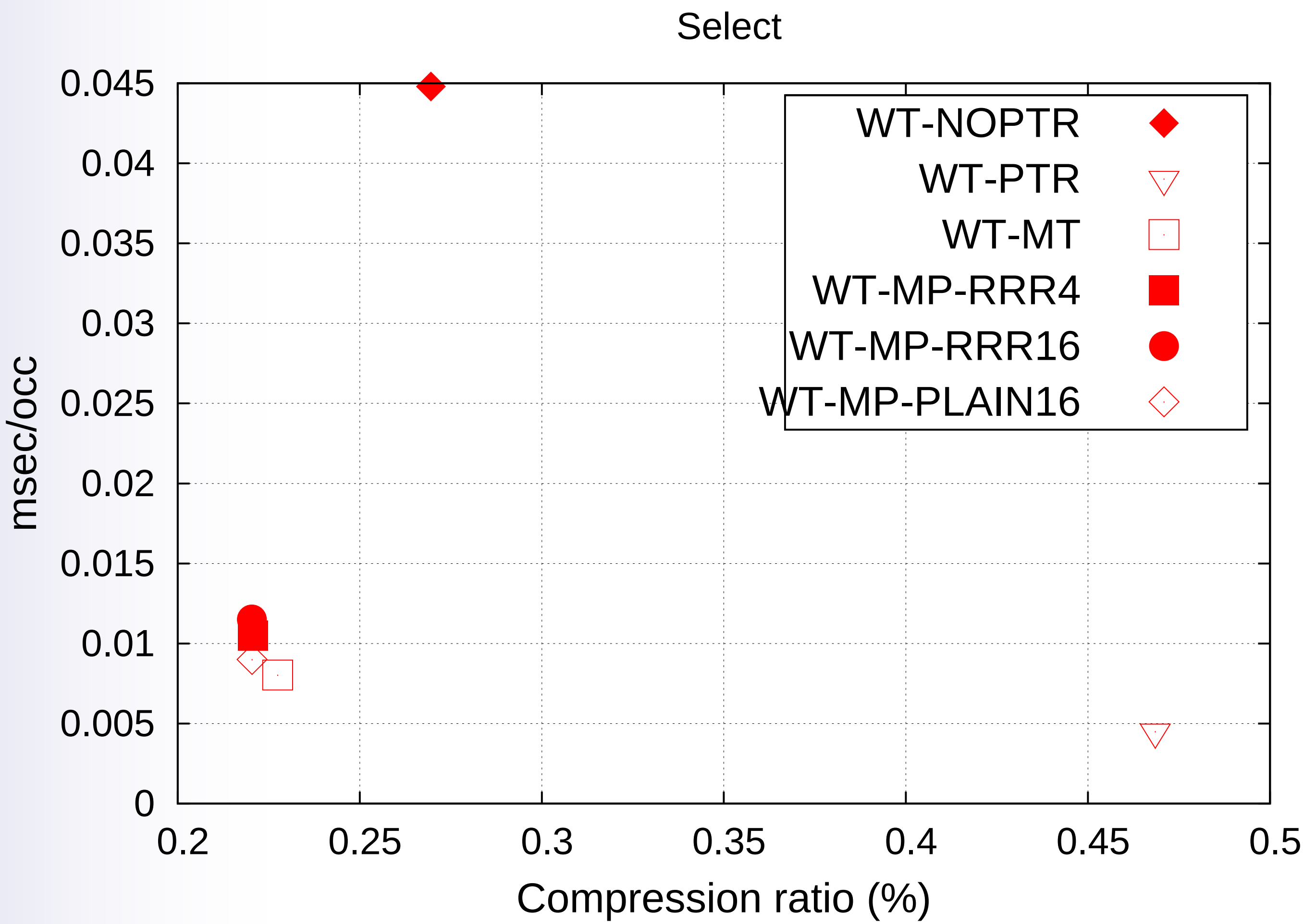
Experimental evaluation

- Count



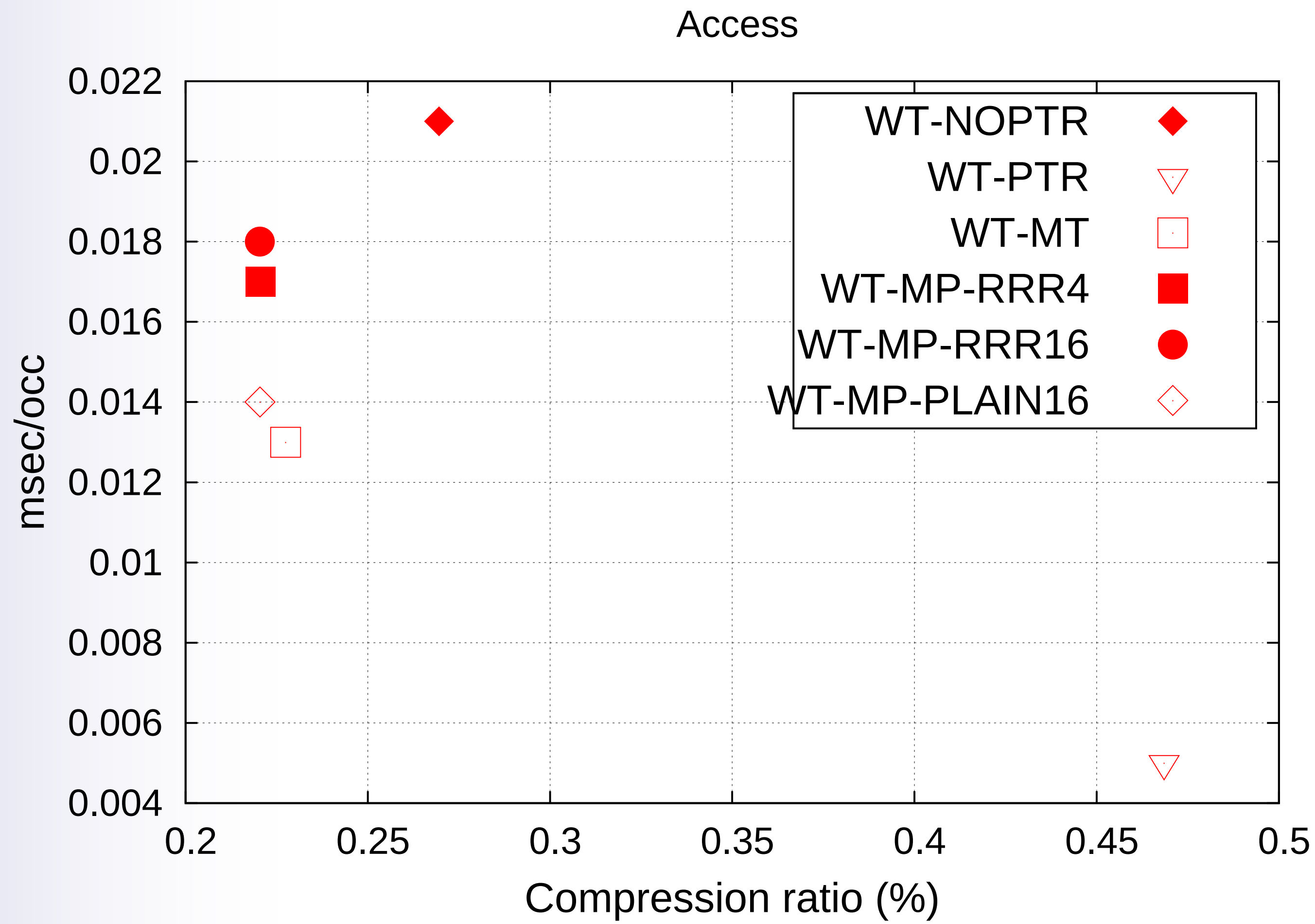
Experimental evaluation

- Select



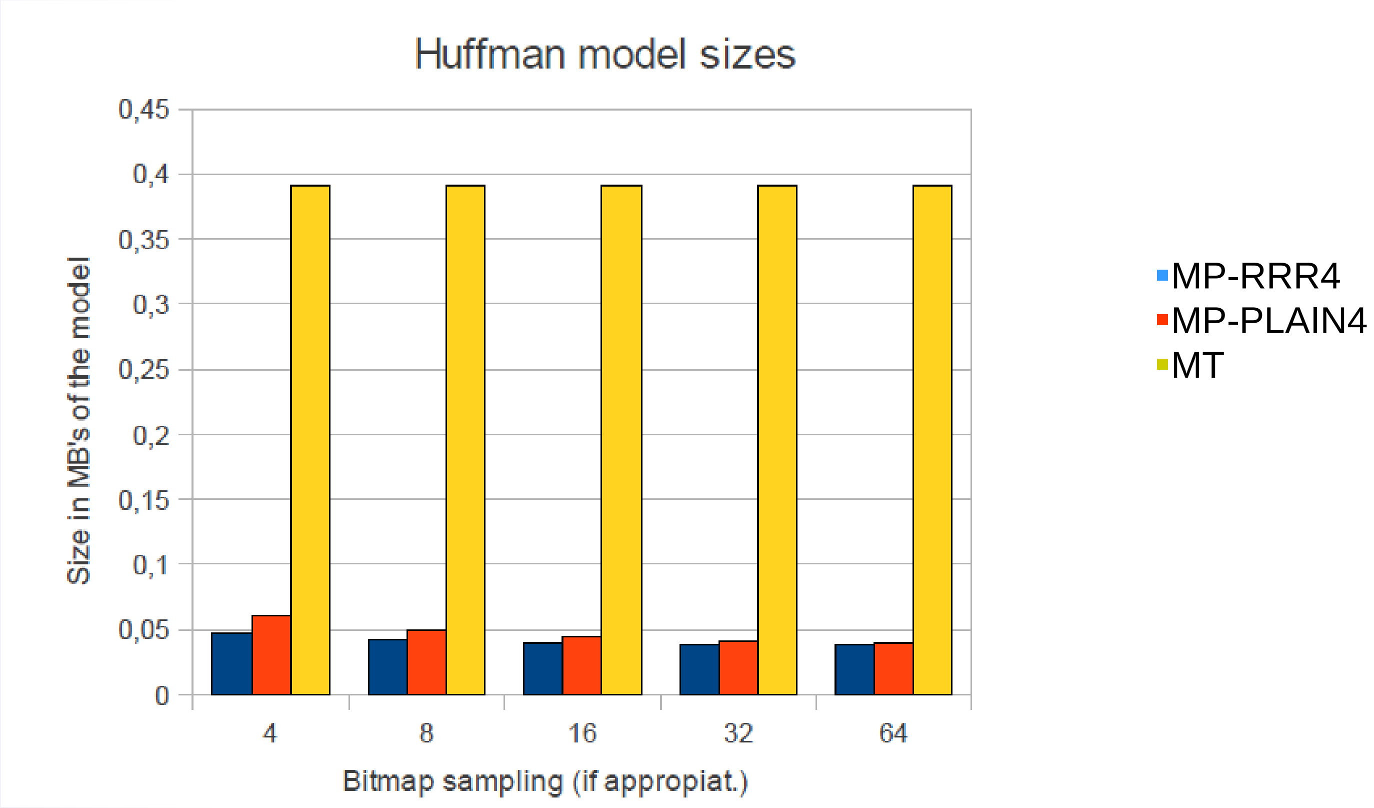
Experimental evaluation

- Access



Experimental evaluation

- Model size:



MT (Model using a Table) takes more than **7 times** the size of the compressed model using permutations (MP).

Questions?