Space-Efficient Data Structures

Francisco Claude
Gonzalo Navarro

University of Waterloo
Departamento de Ciencias de la Computación
UNIVERSIDAD DE CHILE
Google

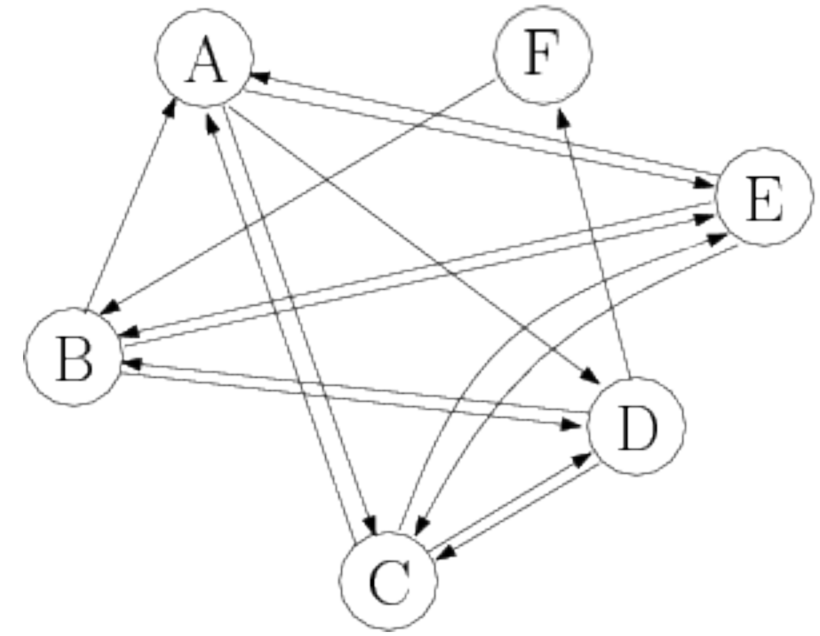
continues...

The Goal

Design data structures that:

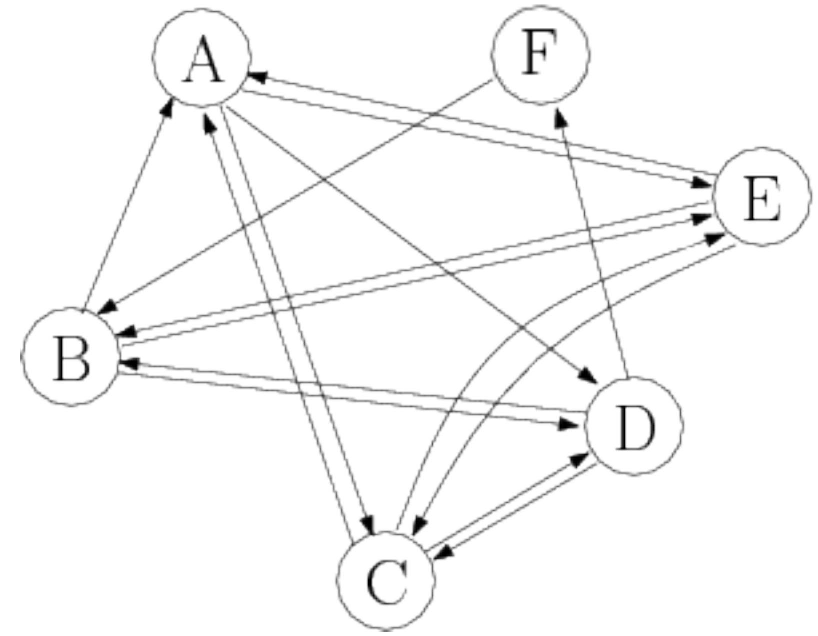
- Have a small memory footprint
- Support fast queries

Web Graphs



- UK-Union-2006-06-2007-05
- Nodes: 133,633,040
- Edges: 5,507,679,822
- Plain representation: 22GB

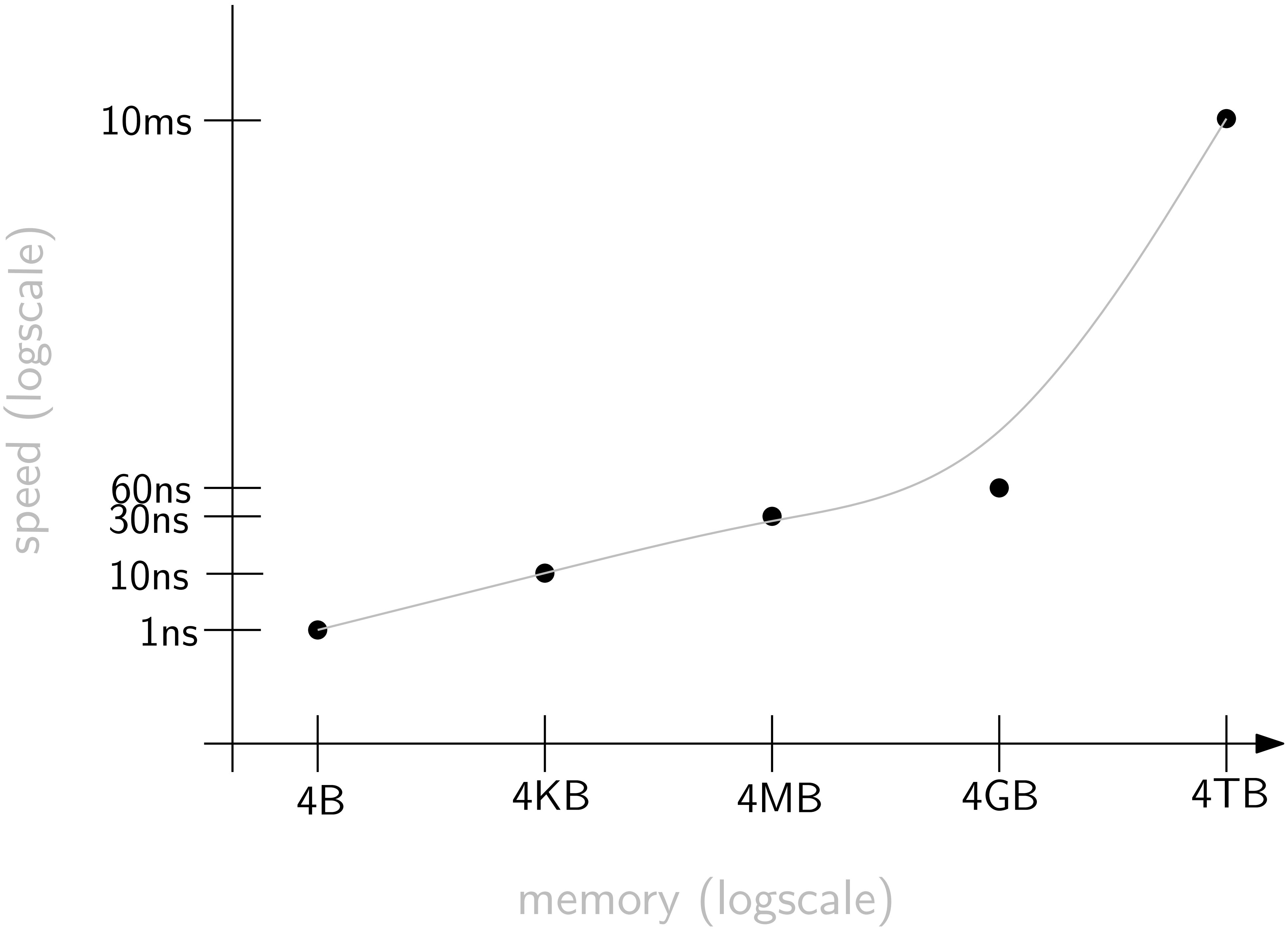
Web Graphs



- UK-Union-2006-06-2007-05
- Nodes: 133,633,040
- Edges: 5,507,679,822
- Plain representation: 22GB

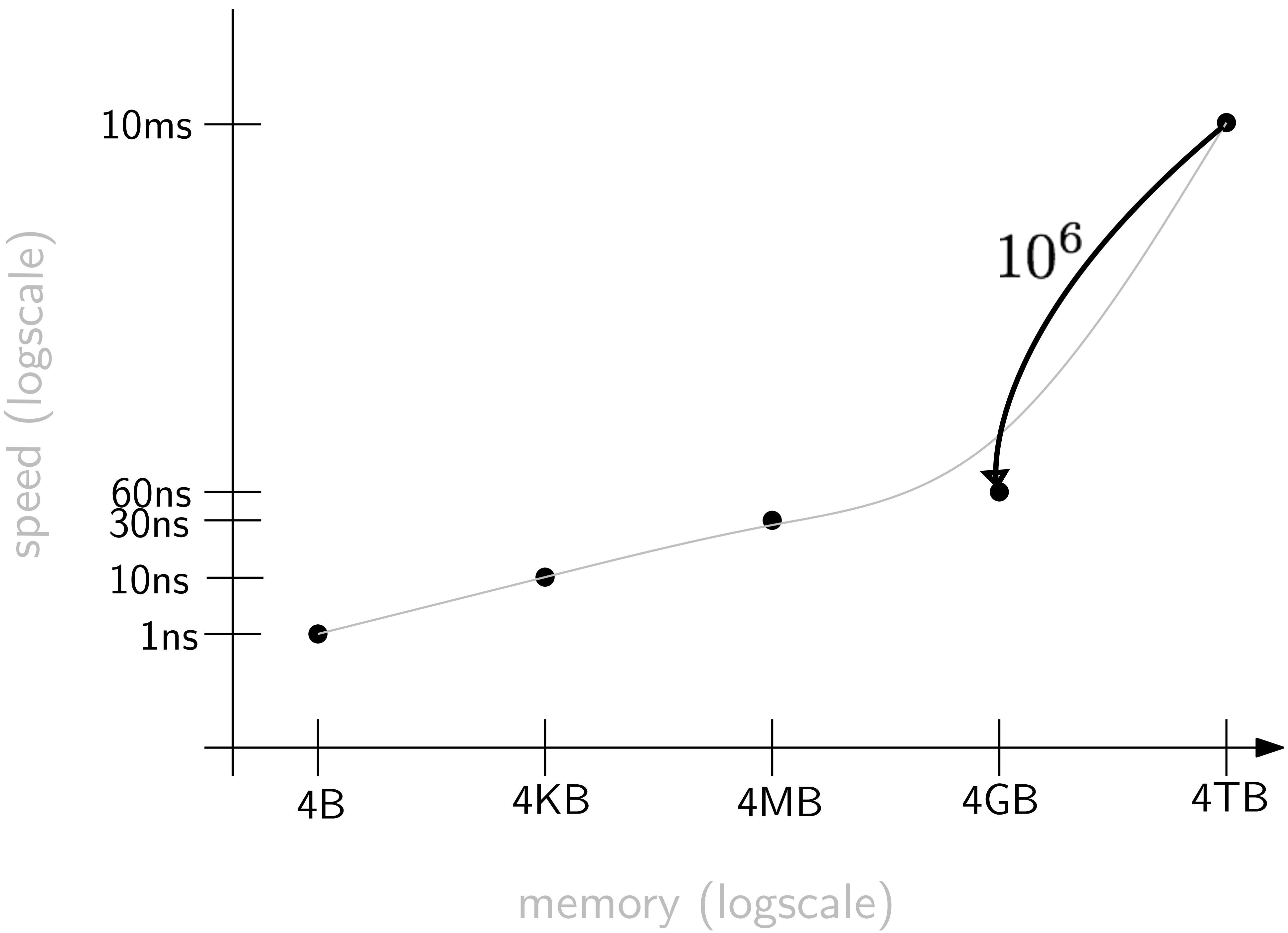
We can get it down to <2GB!!

Motivation



10ms
60ns
30ns
10ns
1ns
speed (logscale)
4B
4KB
4MB
4GB
4TB
memory (logscale)

Motivation


10ms
60ns
30ns
10ns
1ns
speed (logscale)
10^6
4B
4KB
4MB
4GB
4TB
memory (logscale)

Outline

- Representing information
- Bitmaps
  - Trees (intro)
- Sequences (and permutations)
- Applications
- Conclusions

The model

- Word-RAM model
  - RAM of size n
  - We can manipulate $w = \Theta(\log n)$ bits at the time
  - CPU with $O(1)$ registers
  - Operations +, -, *, /, <<, >> take constant time -- we can address with the result

Arrays

- Store elements addressed by an index
- Support efficient access
- Ideally, support some sort of mutation

What we are used to

```
uint *a = (uint*)malloc(sizeof(uint) * n);
... a[i] ...
```

```
uint *a = new uint[n];
... a[i] ...
```

```
a := make([]uint32, n)
... a[i] ...
```

...

What we are used to

```
uint *a = (uint*)malloc(sizeof(uint) * n);
... a[i] ...
```

```
uint *a = new uint[n];
... a[i] ...
```

```
a := make([]uint32, n)
... a[i] ...
```

... We use 32/64 bits per element

What if all values are small?

- We may not need 32/64 bits per element
- Say the maximum value is m
- We can use $\lceil \log_2(m+1) \rceil$ bits per element

What if all values are small?

- We may not need 32/64 bits per element
- Say the maximum value is m
- We can use $\lceil \log_2(m+1) \rceil$ bits per element

```
inline void SetField(cds_word *A, const cds_word len, const cds_word index, const cds_word x)
{
  if (len == 0) {
    return;
  }
  cds_word i = index * len / kWordSize, j = index * len - i * kWordSize;
  cds_word mask = ((j + len) < kWordSize ? ~(cds_word)0 << (j + len) : 0)
                | ((kWordSize - j) < kWordSize ? ~(cds_word)0 >> (kWordSize - j) : 0);
  A[i] = (A[i] & mask) | x << j;
  if (j + len > kWordSize) {
    mask = ((~(cds_word)0) << (len + j - kWordSize));
    A[i + 1] = (A[i + 1] & mask) | x >> (kWordSize - j);
  }
}
```

Space Required (2^32 elements)

Number of bits per element	Total space (GBs)
2	1
4	2
8	4
16	8
32	16
64	32

Arrays in LIBCDS

```
cds_word *values = new cds_word[n];
for (cds_word i = 0; i < n; i++) {
    values[i] = ComputeValueAt(i);
}
Array *A = Array::Create(values, n);
A->SetField(0, 1);
assert(A->GetField(0) == 1);
```

```
Array *A = Array::Create(n, bits);
for (cds_word i = 0; i < n; i++) {
    A->SetField(i, ComputeValueAt(i));
}
```

Compression

- What happens if some values tend to repeat a lot?
- Can we do better?
- Yes, we can assign shorter codes to the most frequent elements (Huffman for example)

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c}$$

Compression

- Sequence S = aaabbcaaabbcaaad

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c}$$

$$H_0(S) = 1.5919$$

symb	freq
a	9
b	4
c	2
d	1

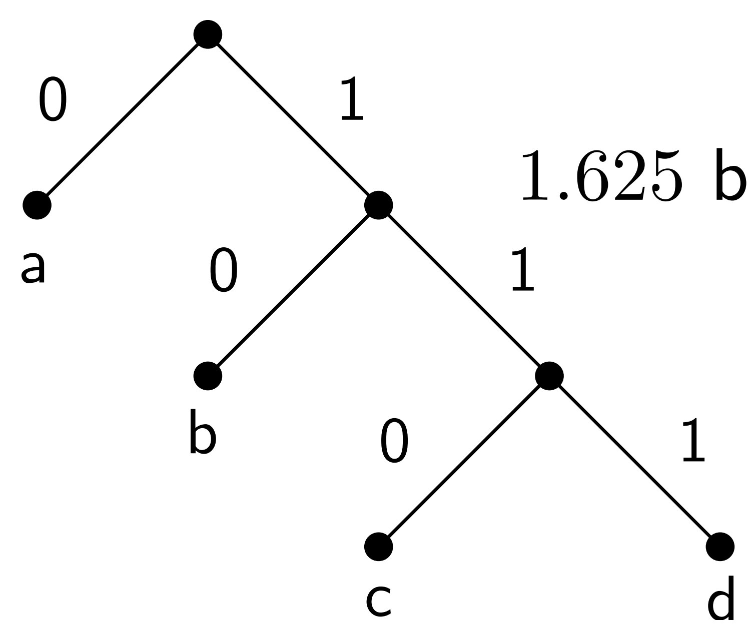
Compression

- Sequence S = aaabbcaaabbcaaad

symb	freq
a	9
b	4
c	2
d	1

$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c}$

$H_0(S) = 1.5919$



1.625 bpc

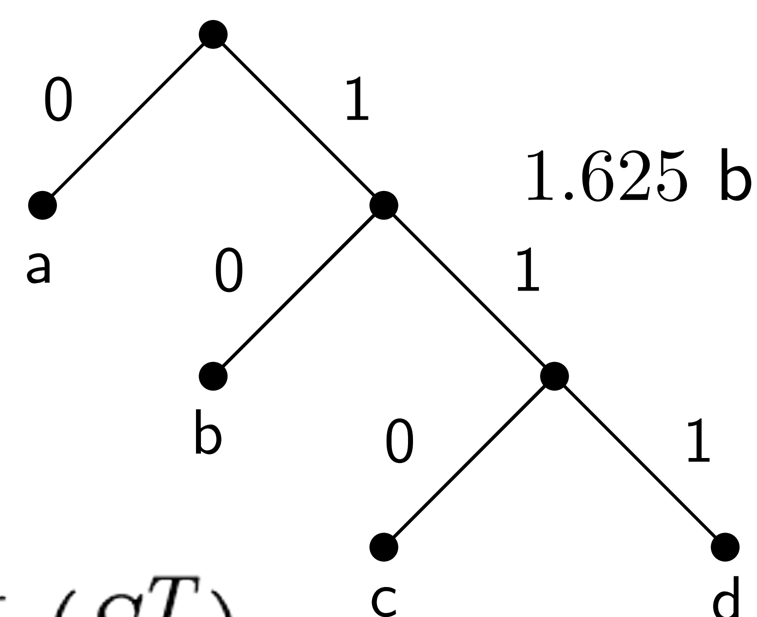
Compression

- Sequence S = aaabbcaaabbcaaad

symb	freq
a	9
b	4
c	2
d	1

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c}$$

$$H_0(S) = 1.5919$$



1.625 bpc

$$H_k(S) = \sum_{|T|=k} |S^T| H_0(S^T)$$

Huffman

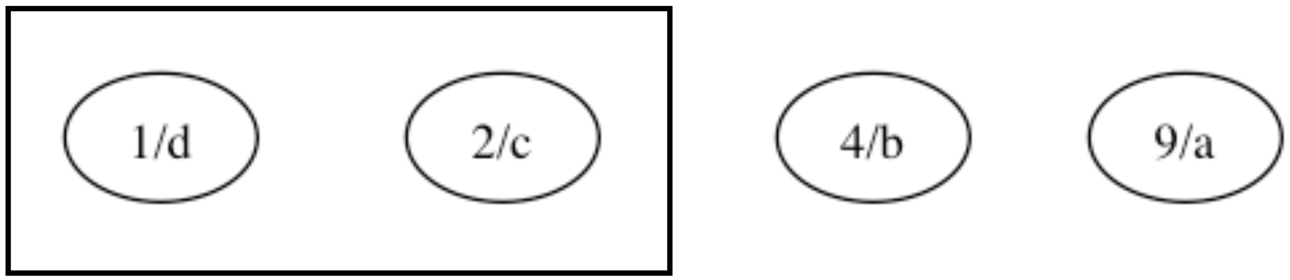
- Huffman's algorithm
  - Every element is a tree
  - Iteratively, take the two least frequent trees and merge them
  - Stop when there is only one tree

Huffman

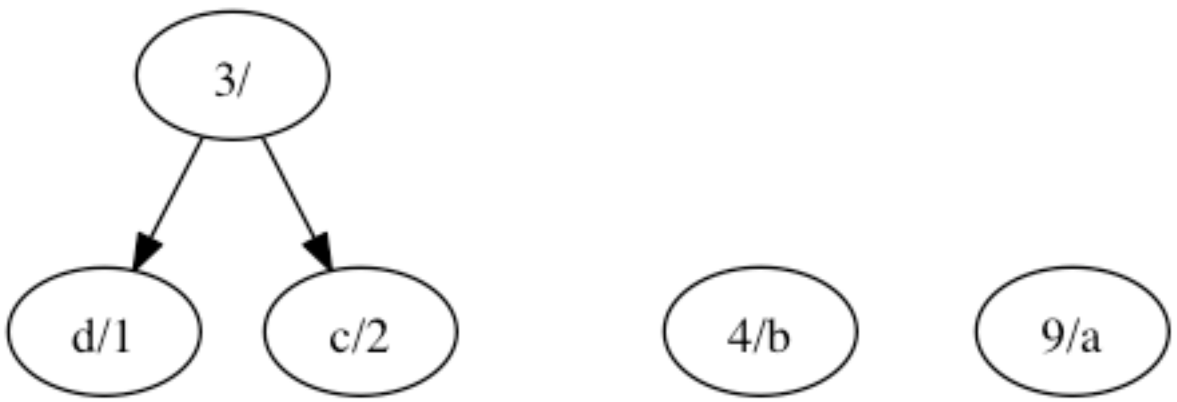
symb	freq
a	9
b	4
c	2
d	1

Huffman

symb	freq
a	9
b	4
c	2
d	1

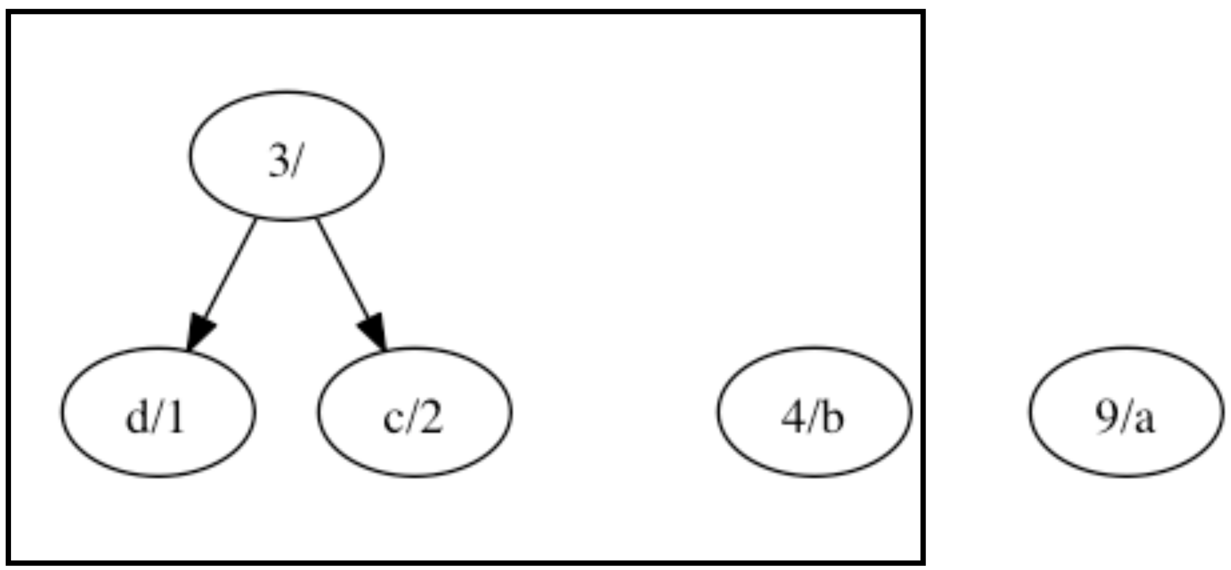


Huffman



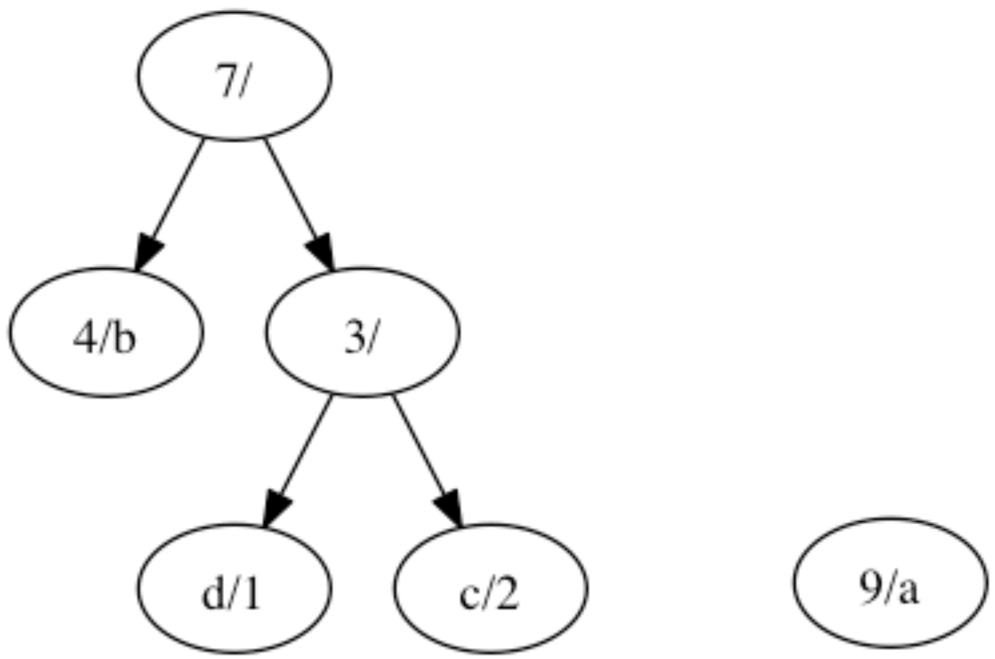
symb	freq
a	9
b	4
c	2
d	1

Huffman



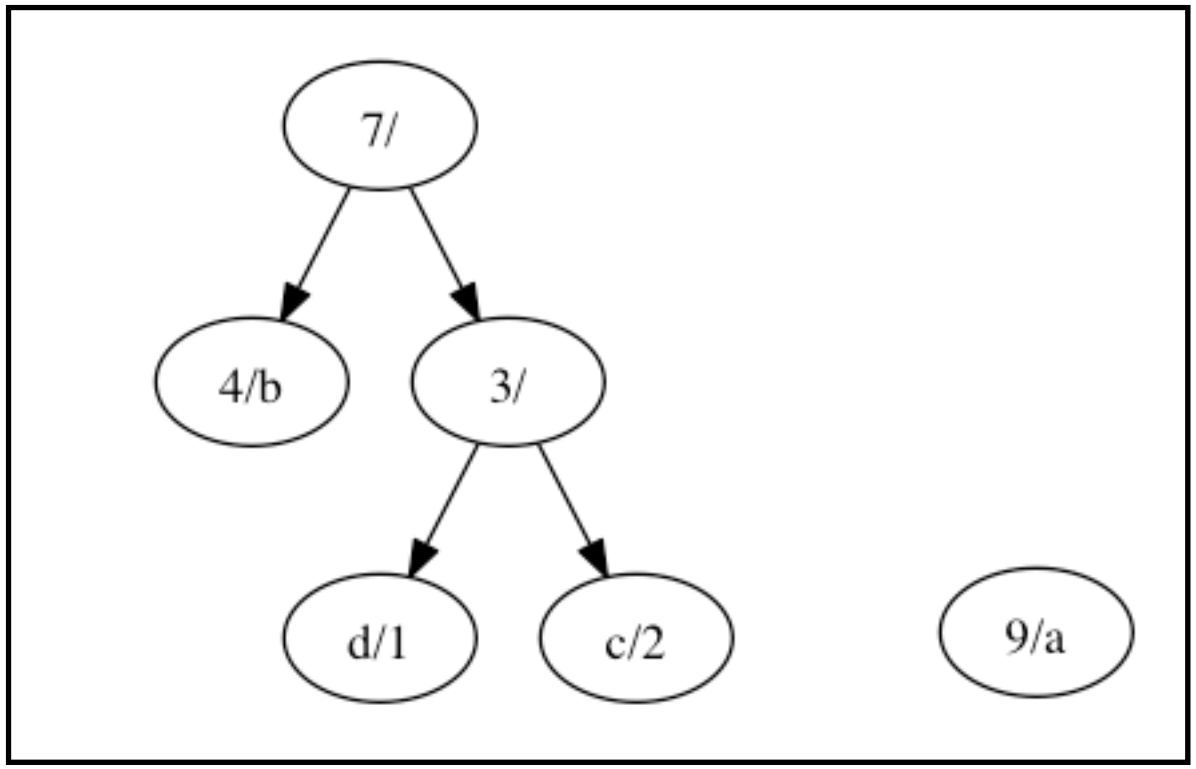
symb	freq
a	9
b	4
c	2
d	1

Huffman



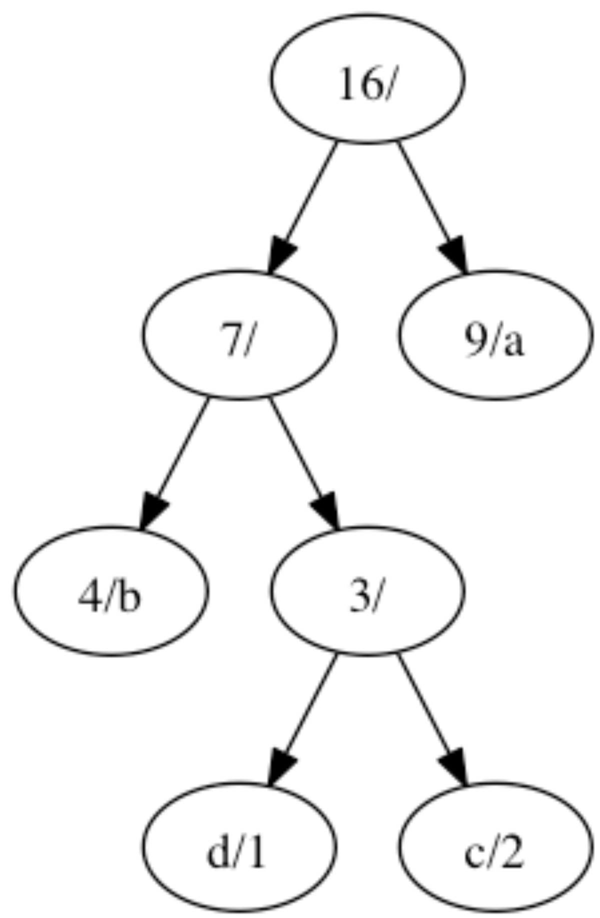
symb	freq
a	9
b	4
c	2
d	1

Huffman



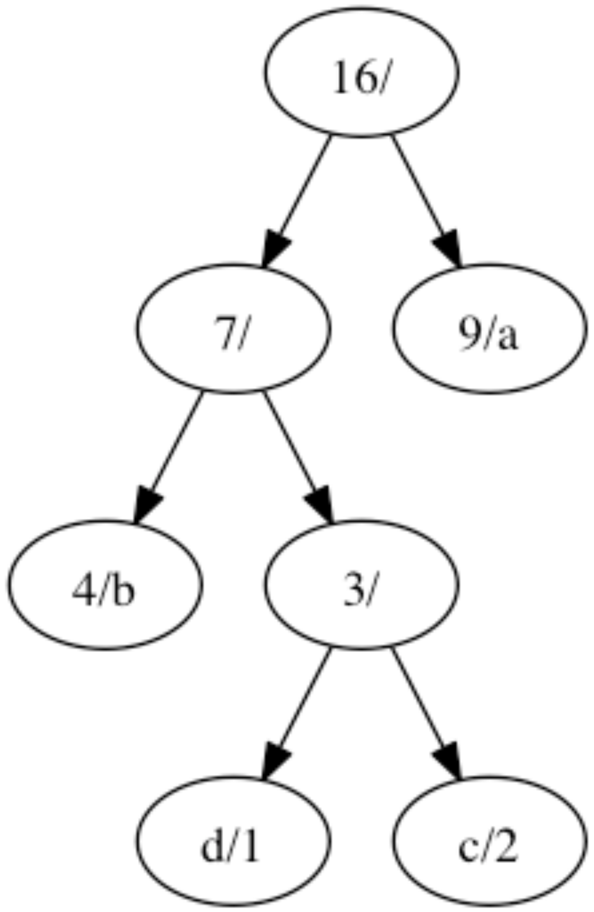
symb	freq
a	9
b	4
c	2
d	1

Huffman



symb	freq
a	9
b	4
c	2
d	1

Huffman



symb	freq
a	9
b	4
c	2
d	1

symb	code
a	1
b	00
c	011
d	010

Huffman

- Sequence S = aaabbcaaabbcaaad

111000011...

Huffman

- Sequence S = aaabbcaaabbcaaad

111000011...

symb	code
a	1
b	00
c	011
d	010

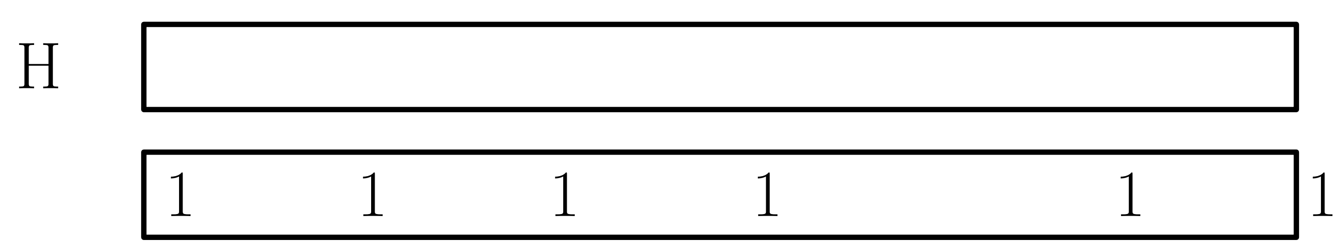
Huffman & Random access

- We can't access an arbitrary position
- One simple solution is to sample the starting position every k elements
- This allows to access in $O(k)$

Huffman + bitmap

- To access position i, we just do

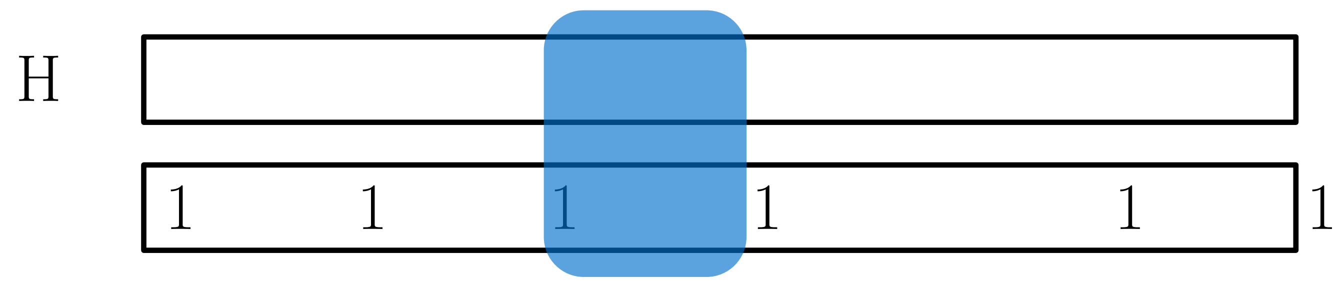
  decode(H[select(1, i)...select(1, i+1)-1])



Huffman + bitmap

- To access position i, we just do

  decode(H[select(1, i)...select(1, i+1)-1])



Huffman & Random access

- We could mark the beginning of each code with a 1 in a separate bitmap that runs in parallel
- If we could find the i-th 1 in the bitmap in constant time, we would be able to access the i-th code.
- This motivates the following...

Huffman & Random access

- Another option is DACs
- In the first level, write down the first bit of each code
- In a bitmap in parallel, mark which codes continue to the next level
- Continue recursively with the next levels

DACs

- Sequence S = aaabbcaaabbcaaad

11100000 1 1...

1	**1**	**1**	**0**	**0**	**0**	**...**		
0	0	0	1	1	1			
0	**0**	**1**	**...**					
0	0	1						
1	**...**							
0								

DACs

- Sequence S = aaabbcaaabbcaaad

1110000011...

1	1	1	0	0	0	...		
0	0	0	1	1	1			
0	0	1	...					
0	0	1						
1	...							
0								

symb	code
a	1
b	00
c	011
d	010

Bitmaps

- access(i): retrieve i-th bit
- rank(0/1, i): count how many 0/1s appear up to position i
- select(0/1, j): find the j-th occurrence of 0/1

$$rank(13) = 7$$

$$B = \boxed{1001110000111}10000\boxed{0}$$

$$access(19) = 0$$

$$select(8) = 14$$

Small bitmaps

- We will build the solution bottom-up
- Consider bitmaps of size $O(\log n)$

Small bitmaps

- Access is trivial using << and >> (v & (1 << i))
- Rank: store all possible answers for bitmaps of length $\frac{\log n}{2}$!

Small bitmaps

- Access is trivial using << and >> (v & (1 << i))
- Rank: store all possible answers for bitmaps of length $\frac{\log n}{2}$!

$$2^{\frac{\log n}{2}} \text{ bitmaps of length } \frac{\log n}{2}$$

Small bitmaps

- Access is trivial using << and >> (v & (1 << i))
- Rank: store all possible answers for bitmaps of length $\frac{\log n}{2}$!

$2^{\frac{\log n}{2}}$ bitmaps of length $\frac{\log n}{2}$

$\frac{\log n}{2}$ queries for each, and the answer takes $\log \log n$ bits

Small bitmaps

- Access is trivial using << and >> (v & (1 << i))
- Rank: store all possible answers for bitmaps of length $\frac{\log n}{2}$!

$2^{\frac{\log n}{2}}$ bitmaps of length $\frac{\log n}{2}$

$\frac{\log n}{2}$ queries for each, and the answer takes $\log \log n$ bits

Total Space: $\frac{\sqrt{n} \log n \log \log n}{2}$

Small bitmaps

00111001 1
rank(1,8)

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

$\boxed{001}110011$
rank(1,8)

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

$\boxed{001}110011$

rank(1,8)

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

001 110011
rank(1,8)

1

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

001110011
rank(1,8)

1

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

0011110011
rank(1,8)

1

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

0011100011

rank(1,8)

1+2

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

001110011

rank(1,8)

1+2

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

001110011
rank(1,8)

1+2

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

0011100011

rank(1,8)

1+2 +1

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

$\boxed{00111001}1$

rank(1,8)

1+2 +1

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

$\boxed{00111001}1$

rank(1,8)

1+2 +1 **=4**

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Small bitmaps

$\boxed{00111001}1$

rank(1,8)

1+2 +1 **=4**

B	p=1	p=2	p=3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

For a word of size $c \log n$ we do $2c$ lookups

Small bitmaps

- How big is the table?

$\frac{\log n}{2}$	KBs
8	0.75
16	512

Small bitmaps

- How big is the table?

Total Space: $$\frac{\sqrt{n}\log n \log\log n}{2}$$

$\frac{\log n}{2}$	**KBs**
8	0.75
16	512

Small bitmaps

- Rank takes constant time on small bitmaps (a computer word)
- Same idea works for select, the possible answers for a block are either a position or "not present"
- Together with the tables for rank, that is enough for answering select in constant time for small bitmaps

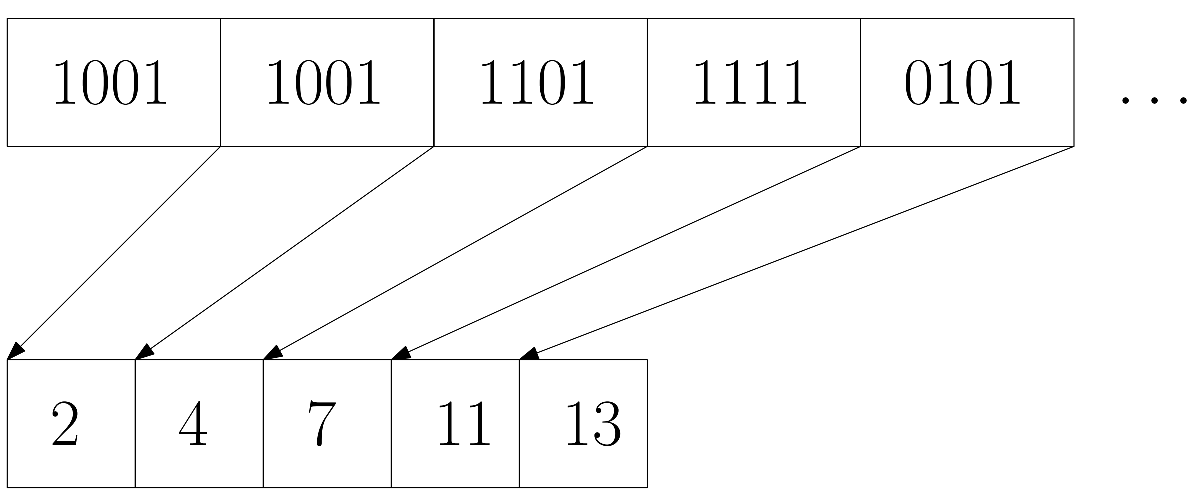
Small bitmaps

- In practice, we can use processor instructions to replace the rank tables (this is called popcount).

```
inline cds_word popcount(cds_word x) {
  if (unlikely(x == 0)) {
    return 0;
  }
  return __builtin_popcountl(x);
}
```

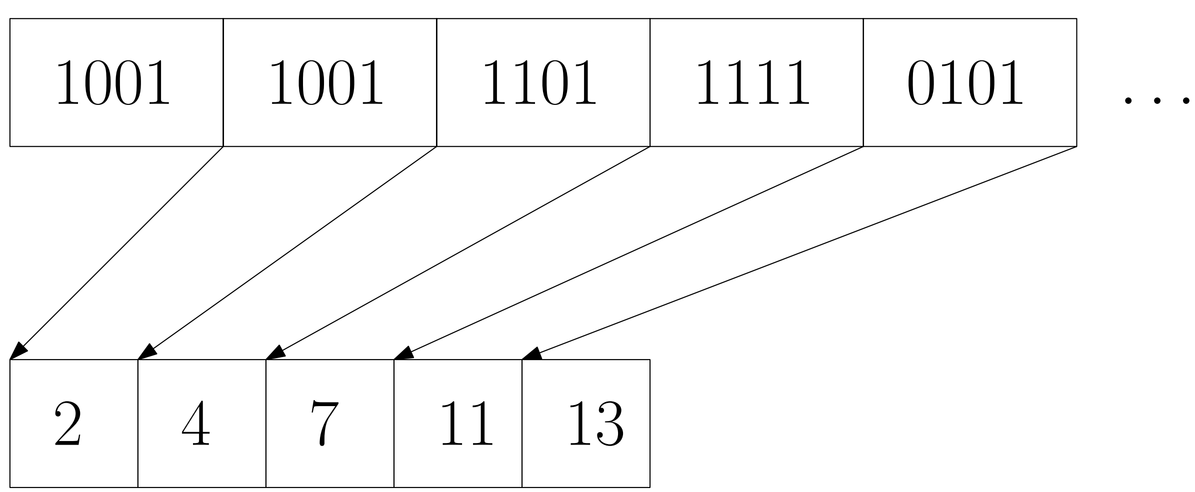
Rank

- We know how to solve for small bitmaps, so try reducing to that
- Lets start by storing some partial answers every s bits



Rank

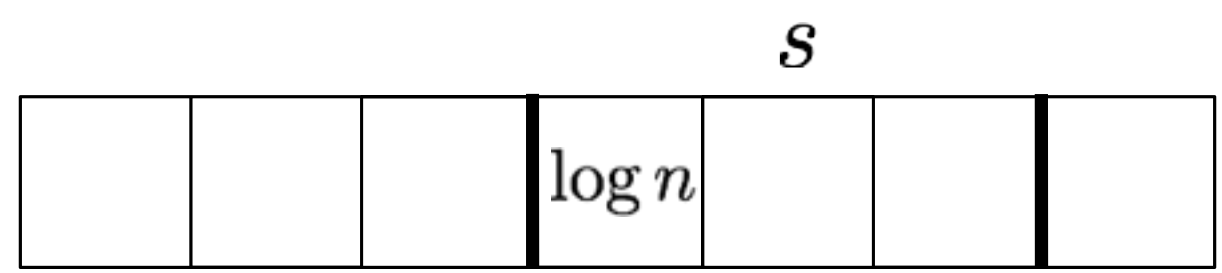
- We know how to solve for small bitmaps, so try reducing to that
- Lets start by storing some partial answers every s bits



$$\frac{n \log n}{s} \ bits$$

Rank

- Sampling every s elements



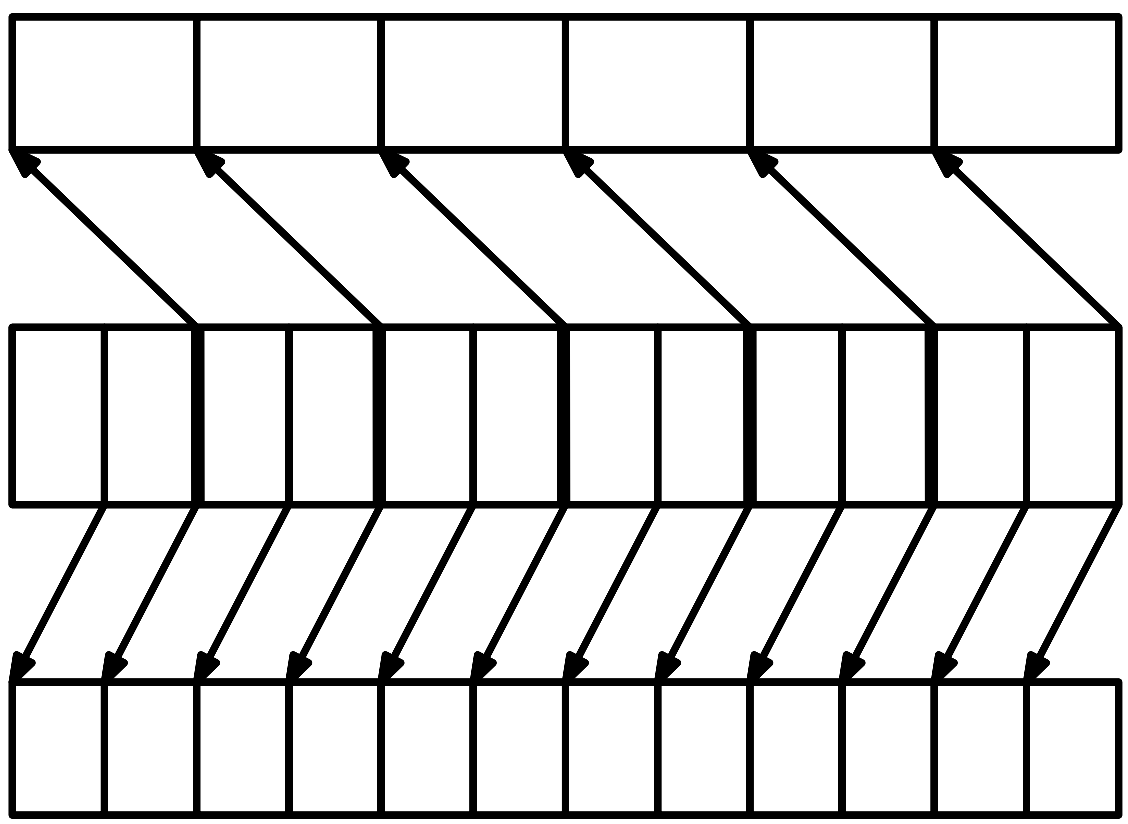
- We can answer rank in $O\left(\frac{s}{\log n}\right)$ time using samples and the tables
- The way to improve further is to consider the blocks generated by the samples as independent problems

Rank

- We sample every $b \le s$ bits, each sample requires $\log s$ bits

- We want to be left with blocks of size $\frac{\log n}{2}$
- We achieve this setting $b = \frac{\log n}{2}$ and $s = \log^2 n$

Rank

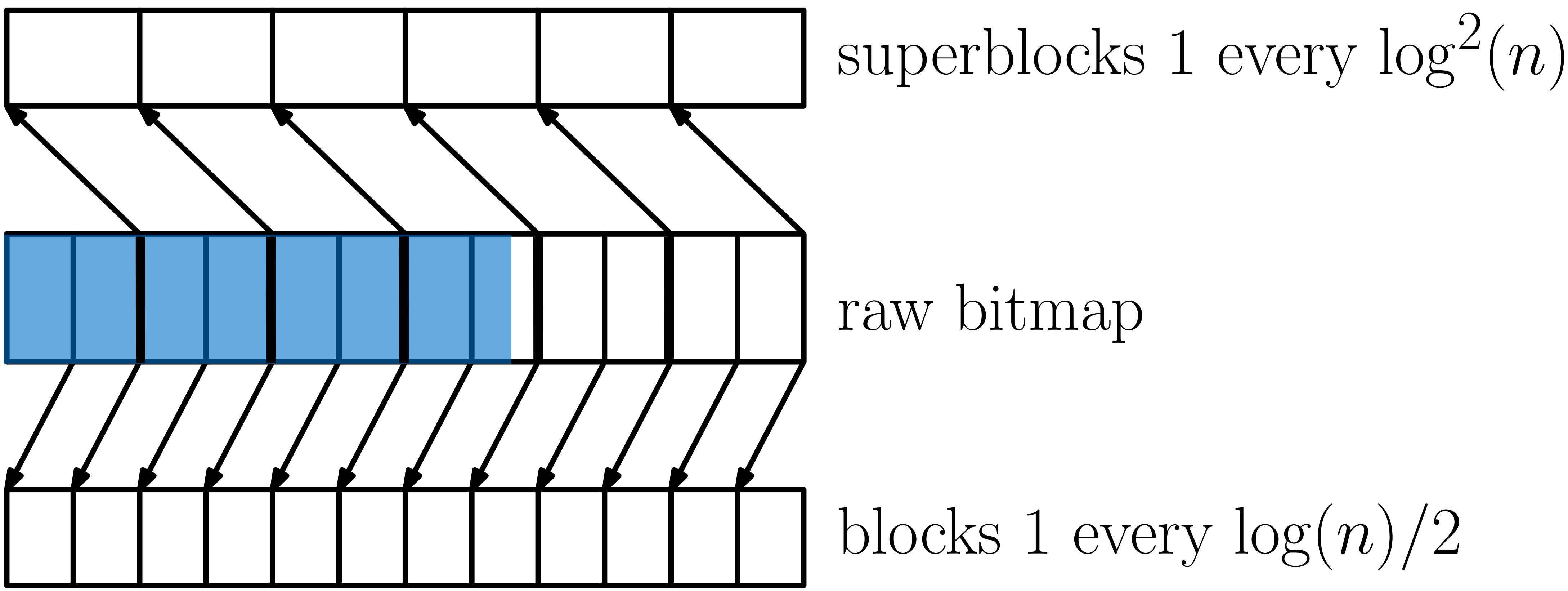


superblocks 1 every $\log^2(n)$

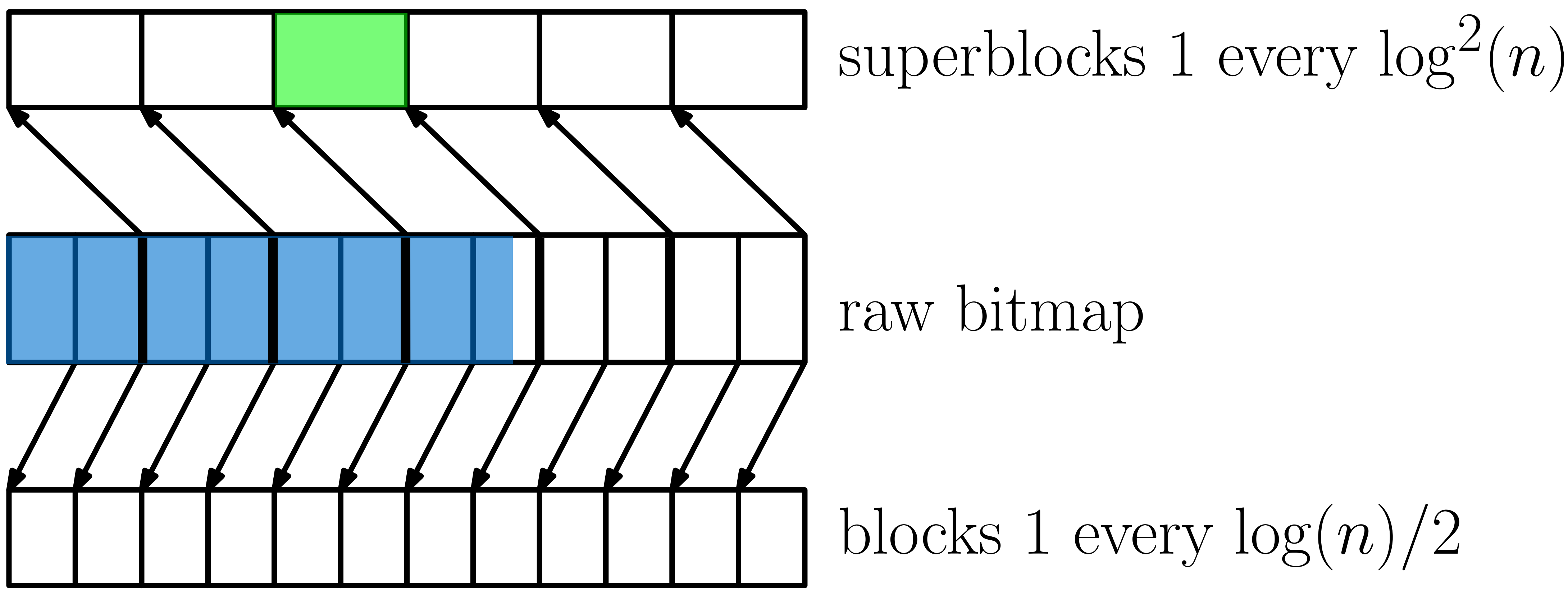
raw bitmap

blocks 1 every $\log(n)/2$

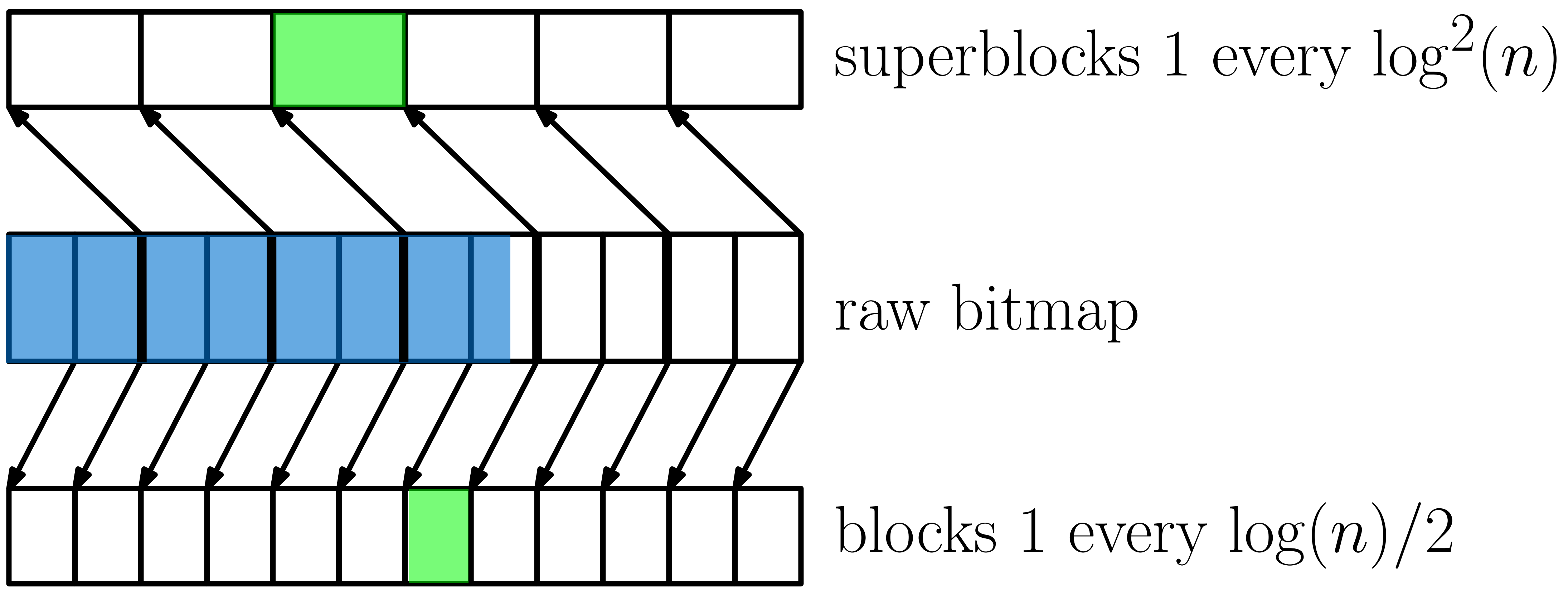
Rank



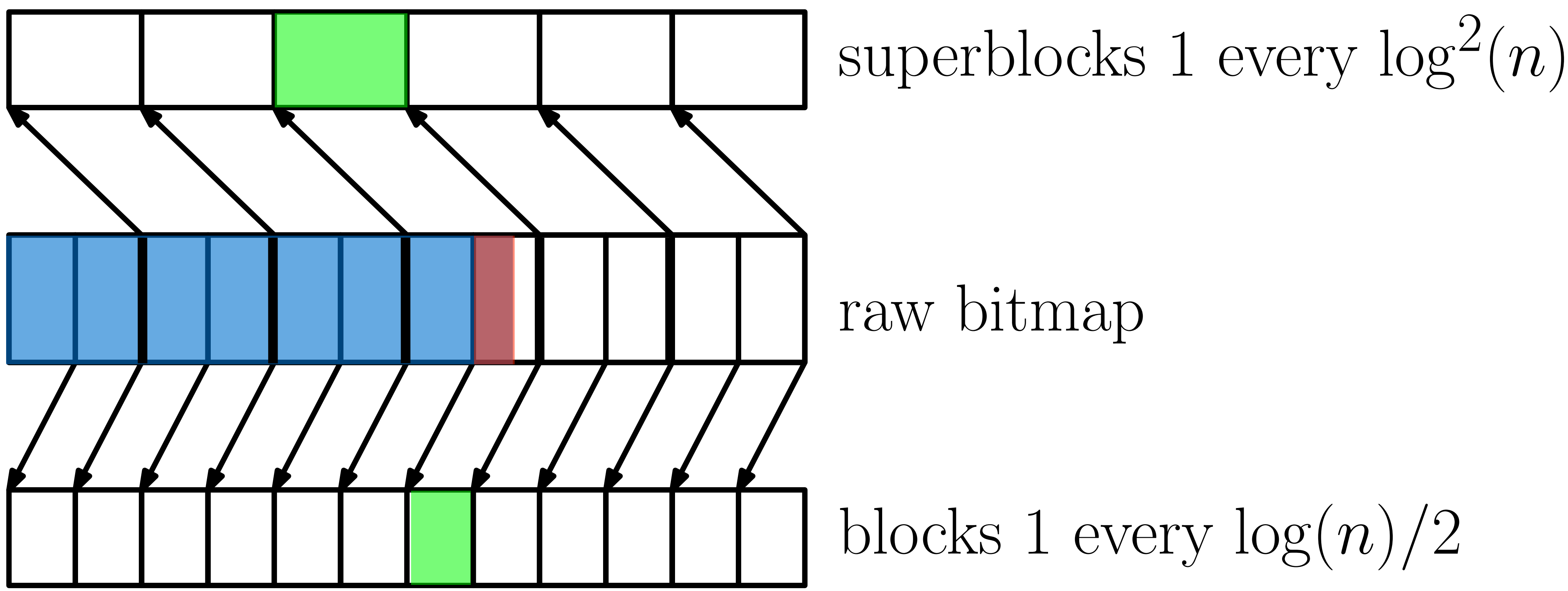
Rank



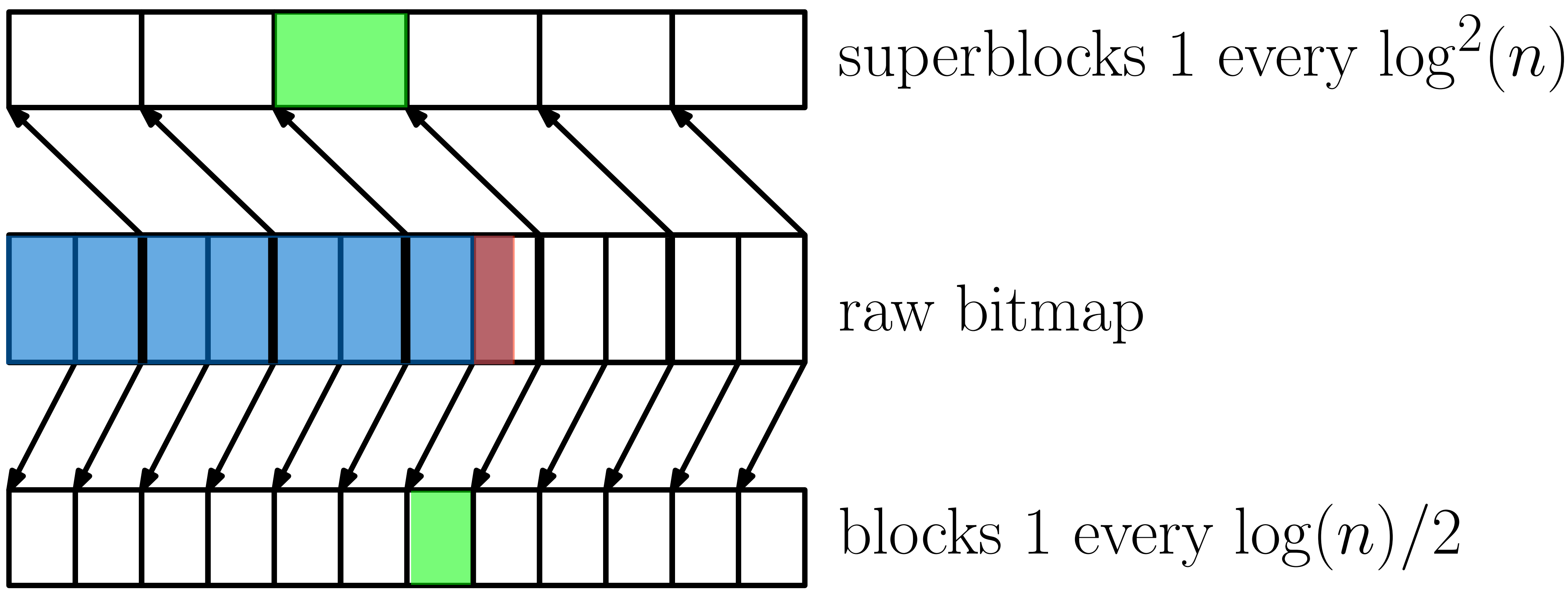
Rank



Rank



Rank



3 additions + popcount!

Rank

- Tables: $\dfrac{\sqrt{n}\log n \log\log n}{2}$
- Blocks: $\dfrac{2n\log\log n}{\log n}$
- Super blocks: $\dfrac{n}{\log n}$
- Bitmap: n
- **Total:** $n + O\left(\dfrac{n\log\log n}{\log n}\right) = n + o(n)$

Select

- We can try something similar to rank, but there is a catch: we cannot use fixed-sized blocks.

$$B = \boxed{10100101}\boxed{001001011}\underbrace{\boxed{0010101001}}_{\log^2 n \text{ 1s}}$$

Select

- We know the answer every $\log^2 n$ 1s and this generates blocks
- We split into two cases: sparse and dense blocks
- We store the answer for all possible arguments for sparse blocks, and recurse on the dense ones

Select

- Sparse blocks (length at least $\log^4 n$):
  - Each answer requires $\log n$ bits
  - The maximum space we will spend is:

$$\frac{n}{\log^4 n} \times \log^3 n = \frac{n}{\log n}$$

Select

- Dense blocks (length at most $\log^4 n$):
  - Split into blocks with $(\log\log n)^2$ ones
  - These sub-blocks are classified as sparse or dense
  - A sub-block is sparse if its length is at least

$$4 \times (\log\log n)^4$$

  - Same idea as before, now the overhead is:

$$\frac{n}{4(\log\log n)^4}(\log\log n)^2 \times 4\log\log n = \frac{n}{\log\log n}$$

Select

- Answering a query:
  - If the block is sparse, return the answer
  - Else go to the corresponding sub-block
    - If the sub-block is sparse, return the answer
    - Else it is not sparse, but it fits in a word

In practice...

- We only keep one level of blocks for rank
- Select is solved the following way:
  - Binary search which block contains the answer
  - Sequentially traverse the block to find the position
- There is another solution storing samples for select + the ones for rank

In practice...

```
Array *a = Array::Create(n, 1);
...
BitSequence *bs = new BitSequenceOneLevelRank(a, sample);
cout << bs->Access(i) << endl;
cout << bs->Rank0(i) << " " << bs->Rank1(i) << endl;
cout << bs->Select0(i) << " " << bs->Select1(i) << endl;
```

Other representations

- Raman, Raman and Rao (constant time)

$$nH_o(B) + o(n)$$

- Okanohara and Sadakane (not constant time)

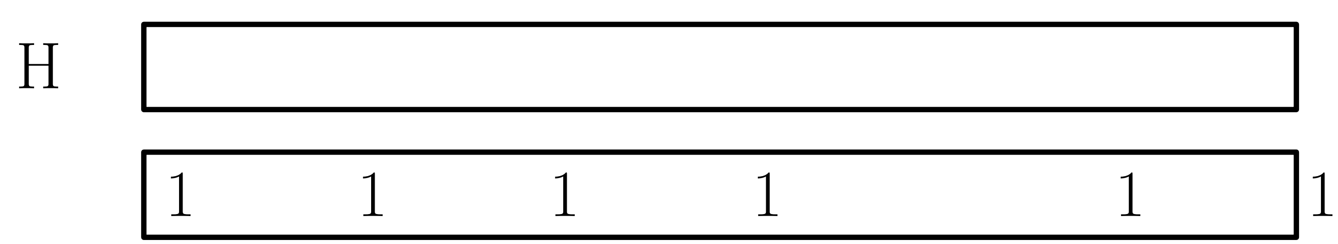
$$nH_o(B) + O(m)$$

- Patrascu (constant time)
  - Reduced the lower order term for compressed bitmaps

Huffman + bitmap

- To access position i, we just do

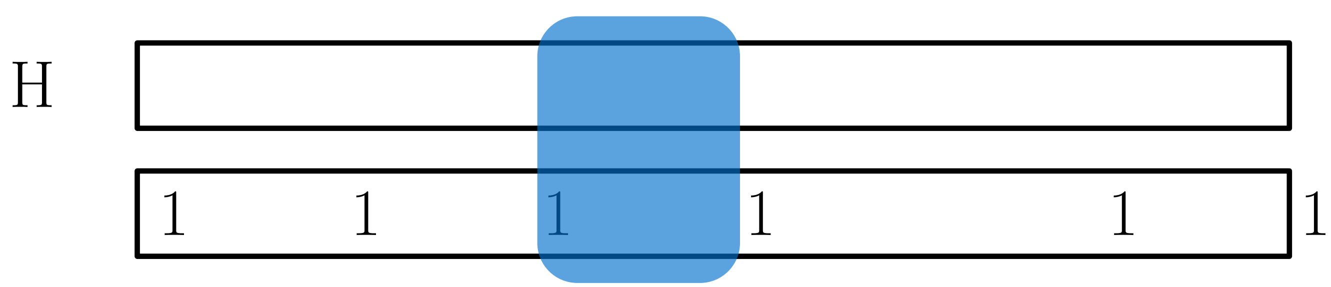
  decode(H[select(1, i)...select(1, i+1)-1])



Huffman + bitmap

- To access position i, we just do

  decode(H[select(1, i)...select(1, i+1)-1])

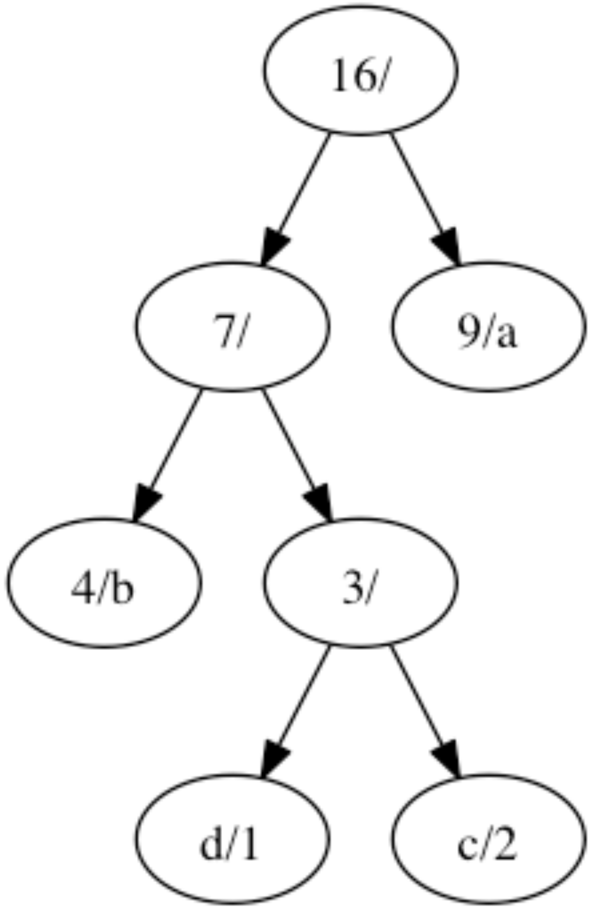


Going back to Huffman

- We can use a compressed bitmap instead
- We mark the beginning of each code in a bitmap of length
- There are n ones $nH_0(S) + n$
- The space for the whole representation is

$$nH_0(S) + o(nH_0(S))$$

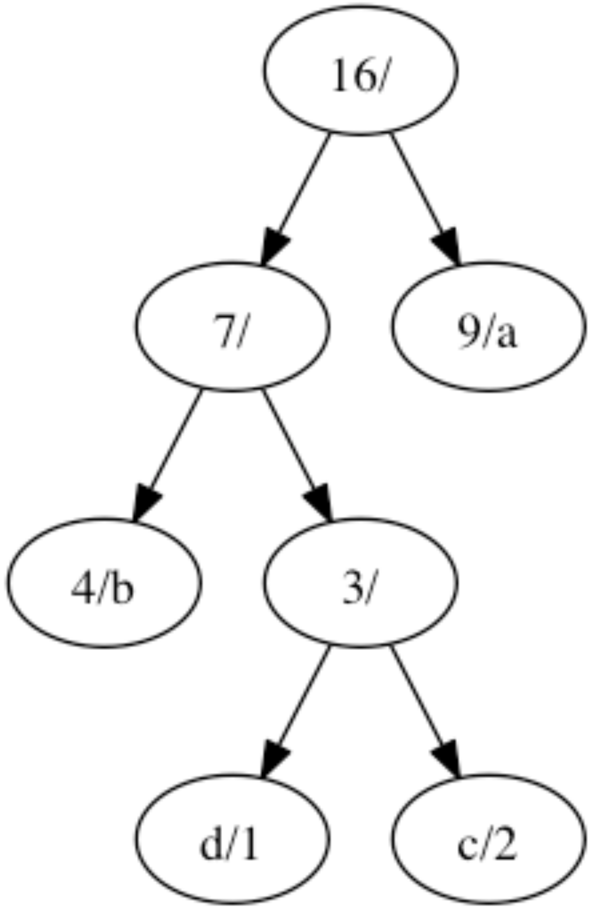
Huffman



- A pointer-based representation requires $O(n \log n)$ bits

```
class Node {
    void *data;
    Node *left, *right;
}
```

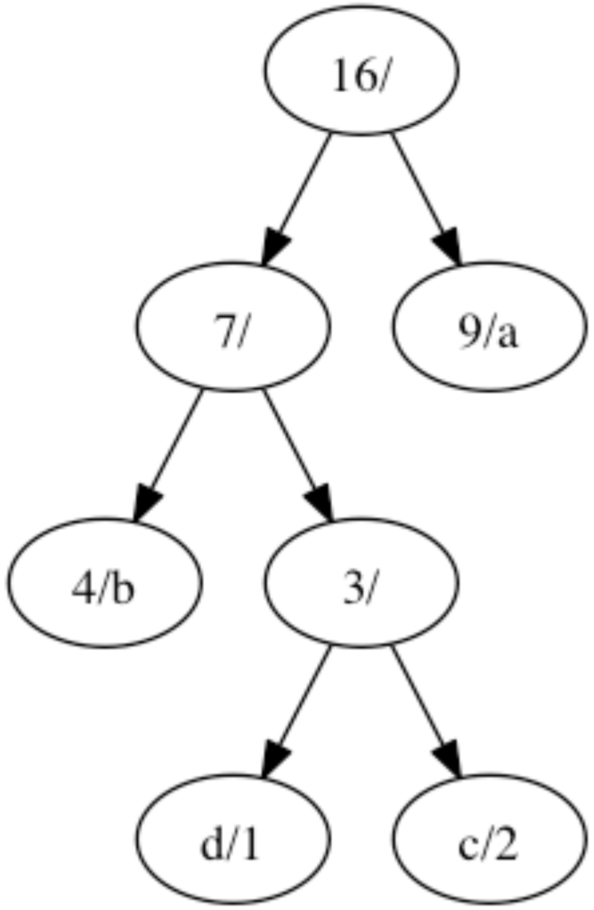
Huffman



- A pointer-based representation requires $O(n \log n)$ bits

```
class Node {
    void *data;
    Node *left, *right;
}
```

Huffman

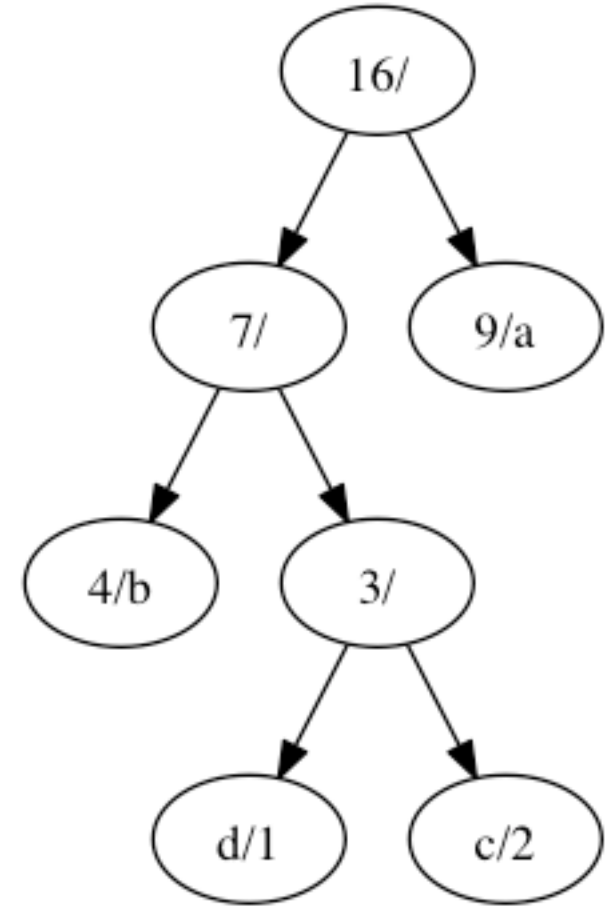


- A pointer-based representation requires $O(n \log n)$ bits

```
class Node {
    void *data;
    Node *left, *right;
}
```

What if we want to know our parent?

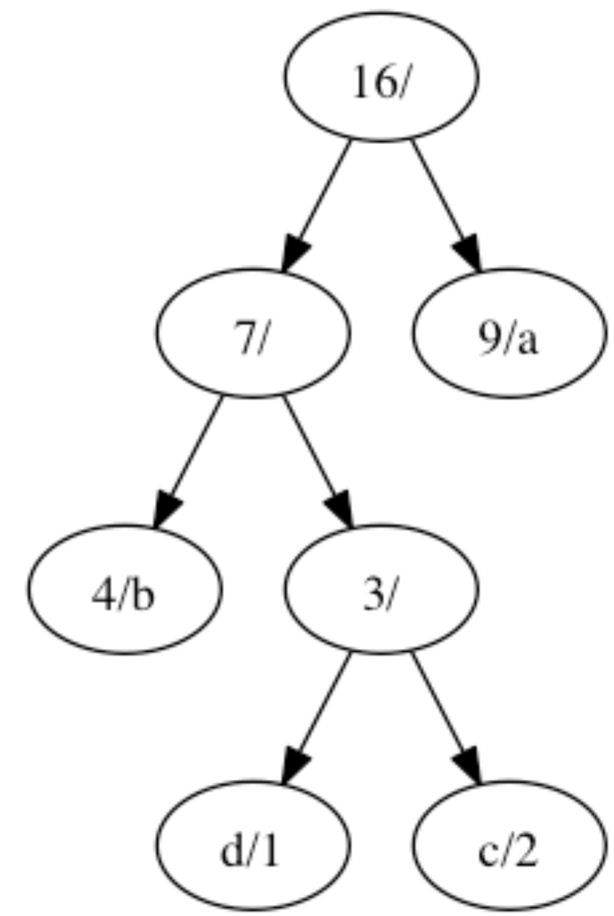
Huffman



- Good example of a simple tree
- Every node has 2 children or is a leaf
- Can we represent the shape of the tree efficiently?

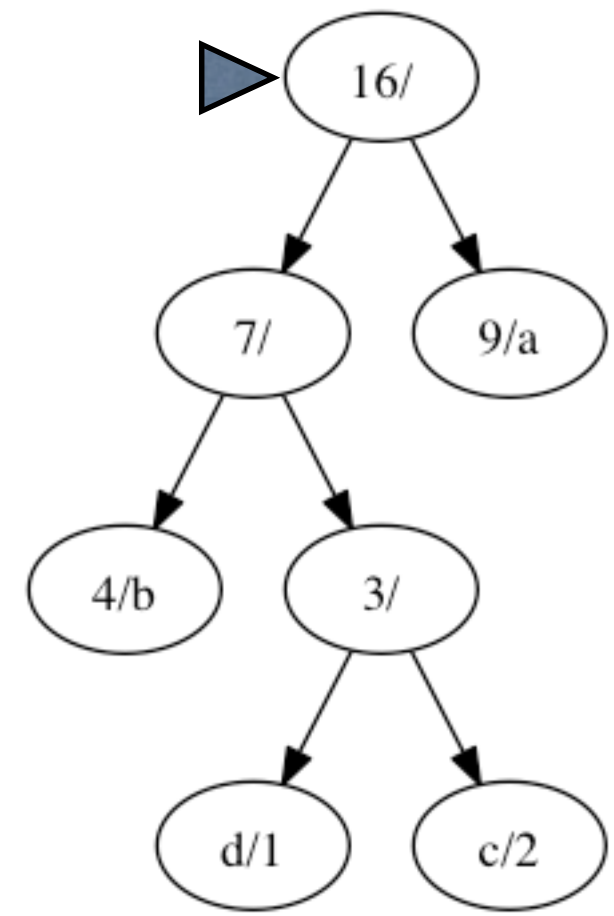
Simple representation

- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



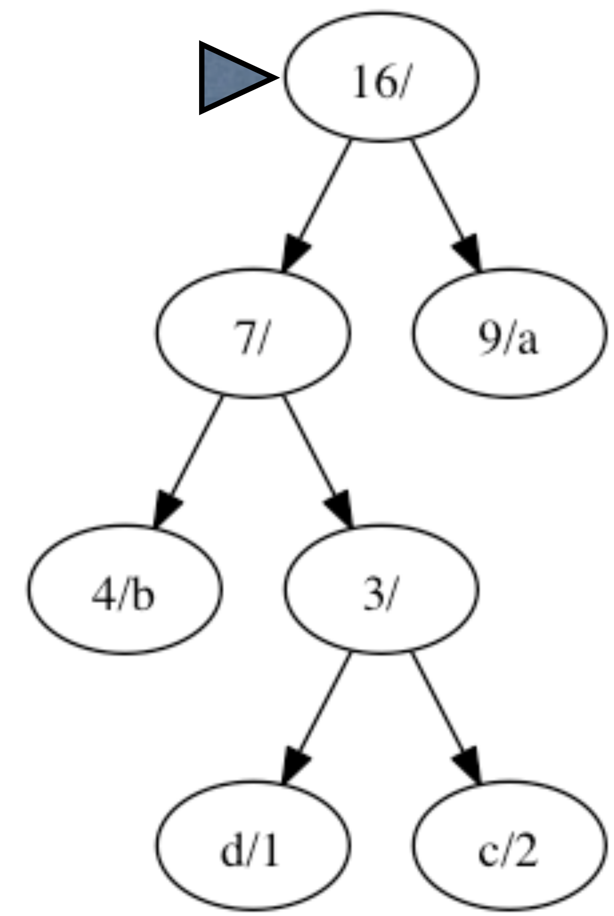
Simple representation

- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



Simple representation

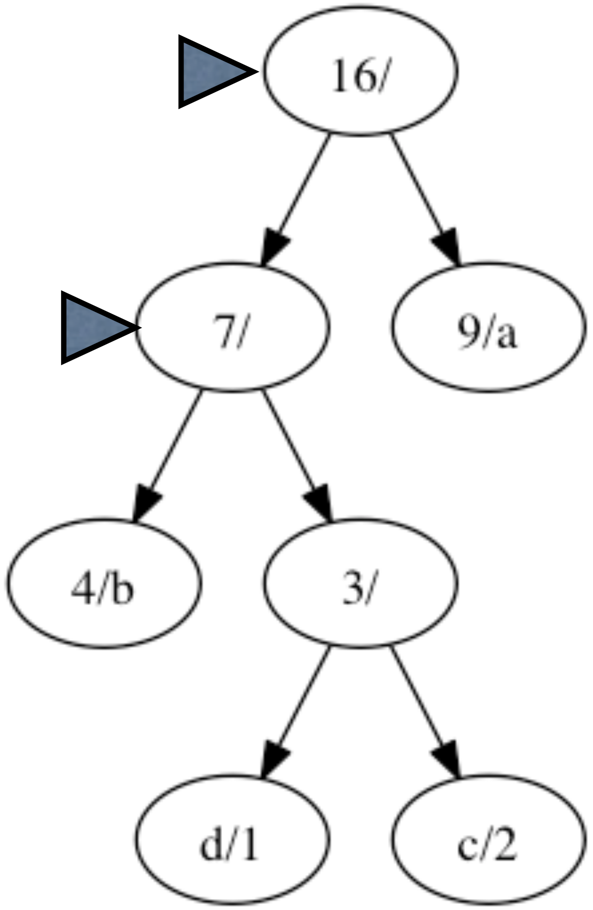
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1

Simple representation

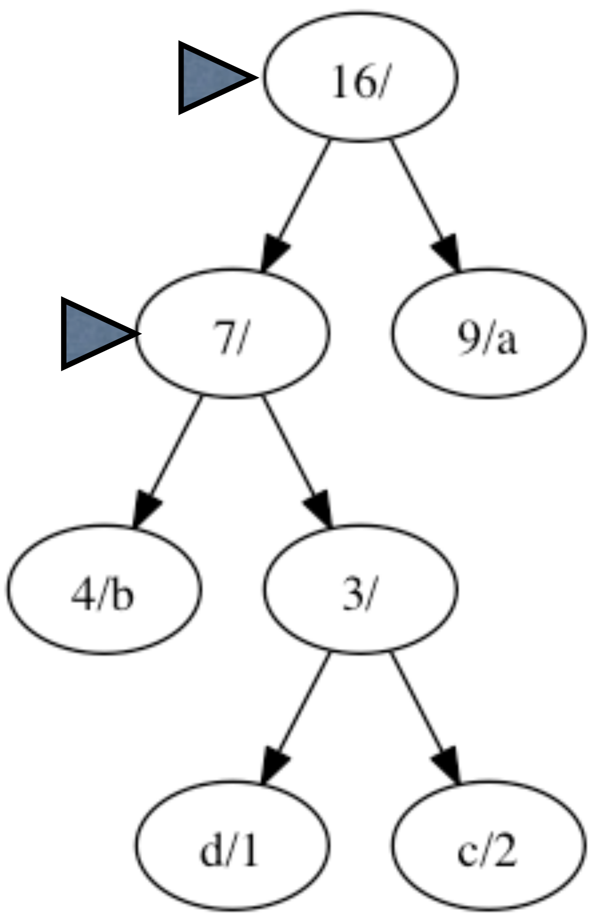
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1

Simple representation

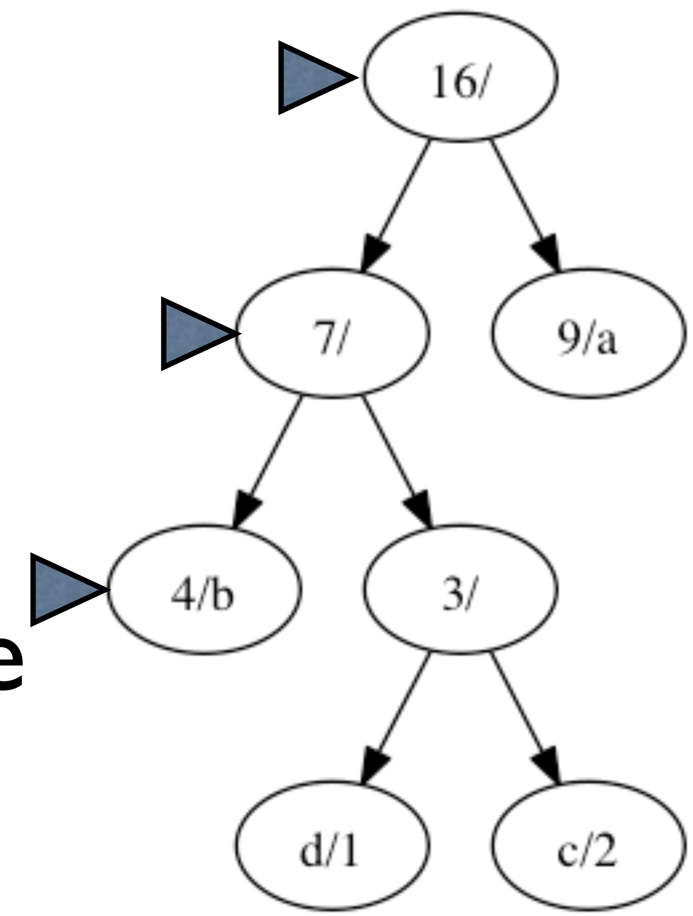
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1 1

Simple representation

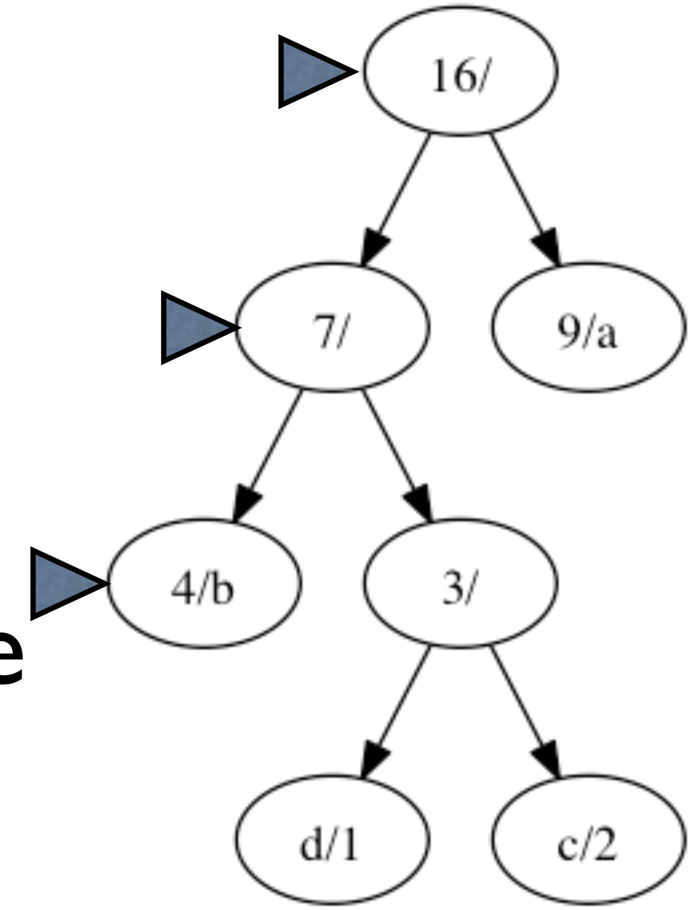
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1 1

Simple representation

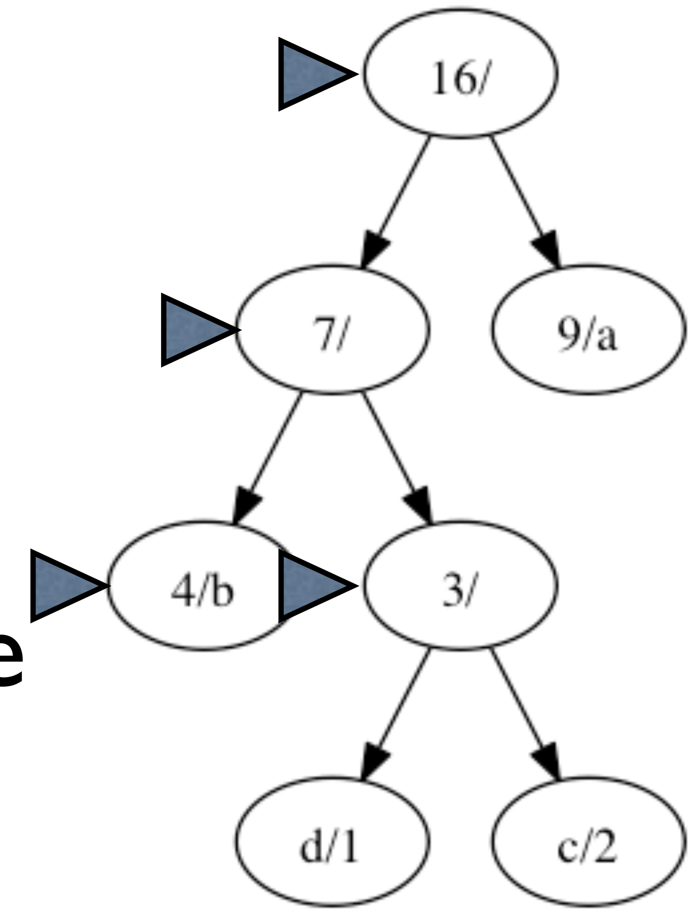
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1 1 0

Simple representation

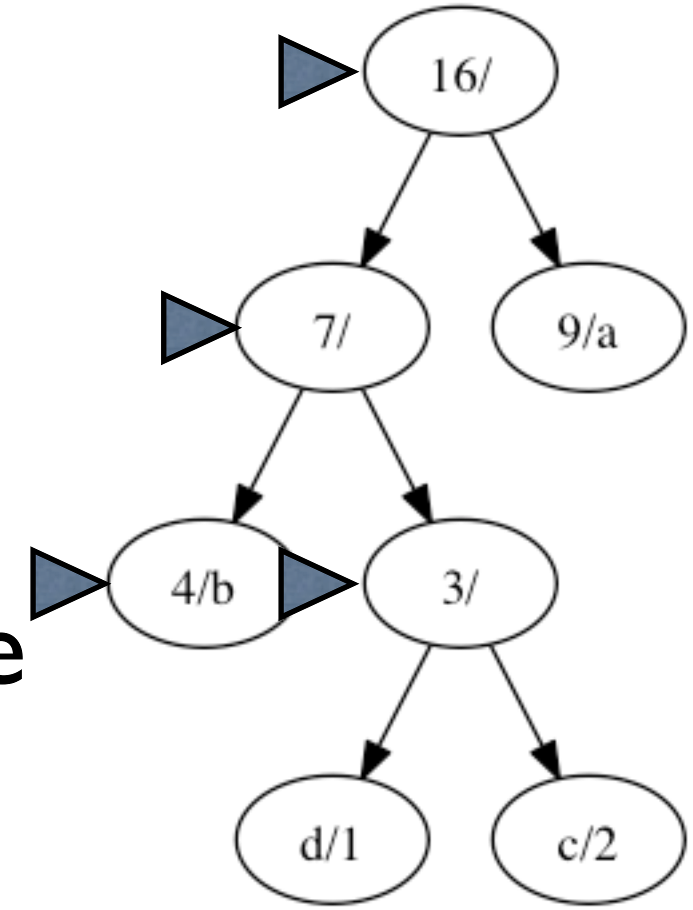
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1 1 0

Simple representation

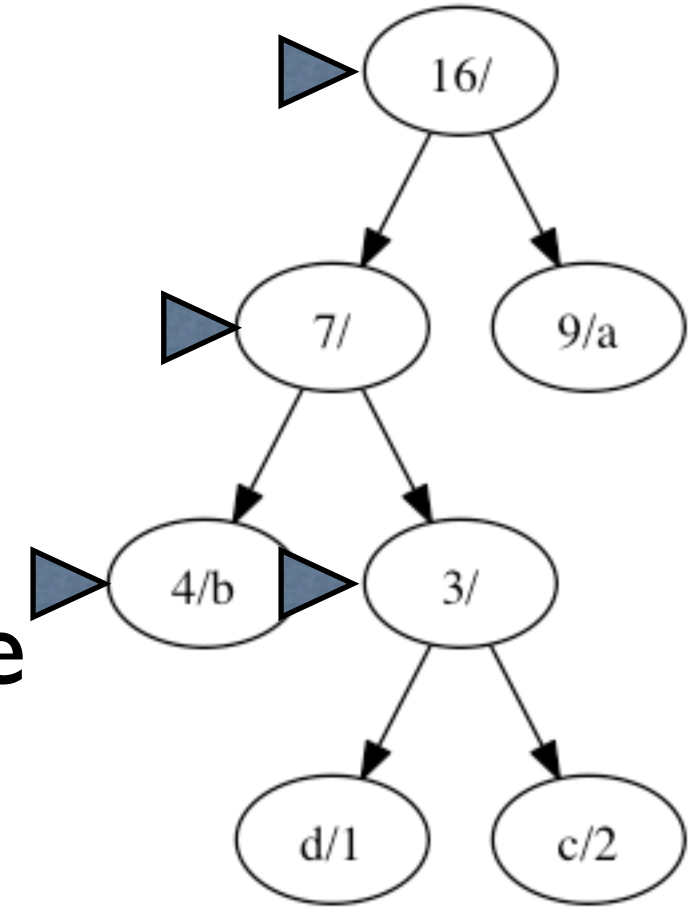
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1 1 0 1

Simple representation

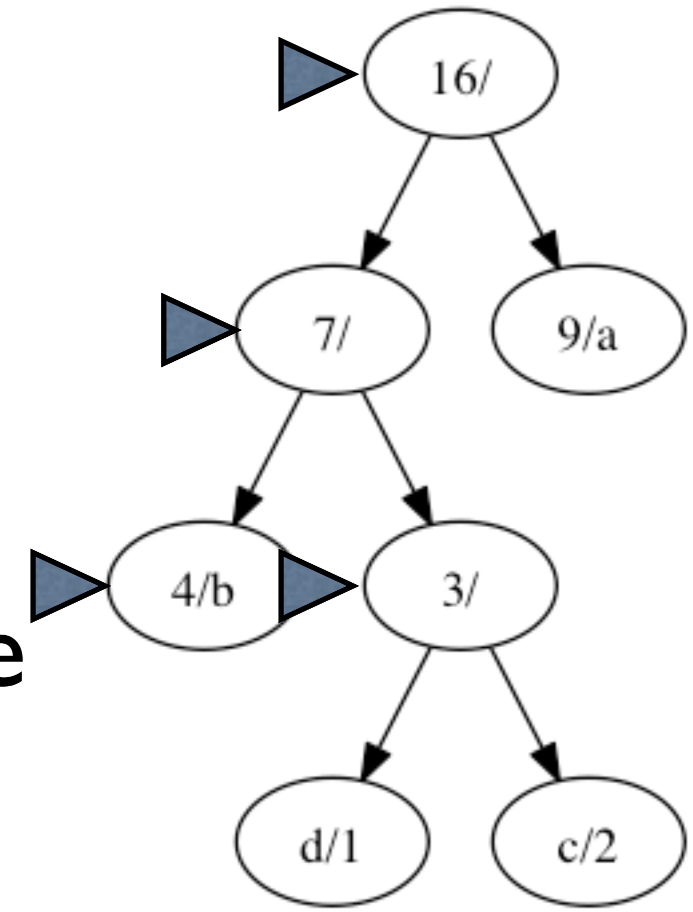
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1 1 0 1 0

Simple representation

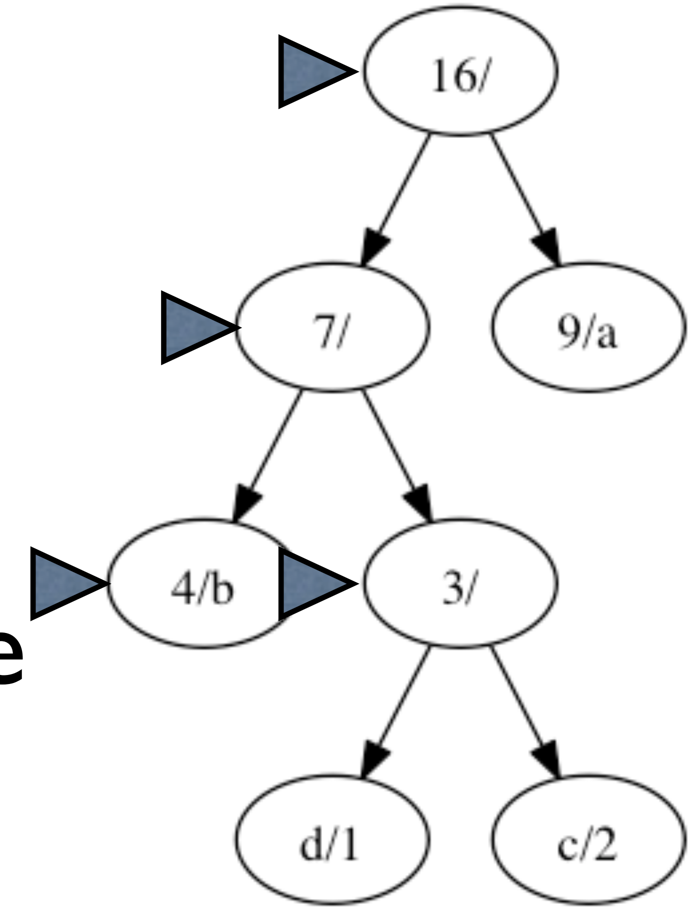
- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



1 1 0 1 0 0

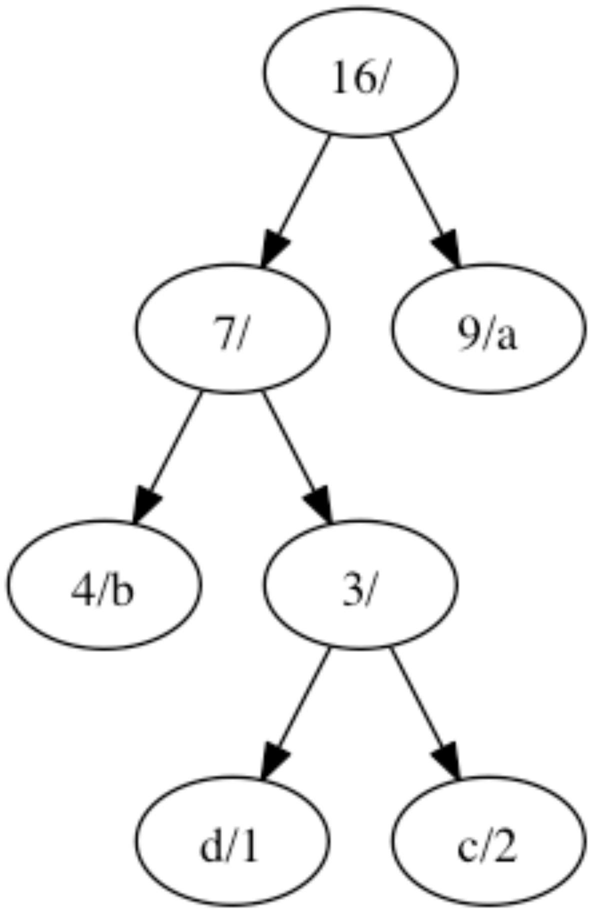
Simple representation

- We will traverse the tree DFS
- Every time we see an internal node, we write a 1
- Every time we see a leaf, we write a 0



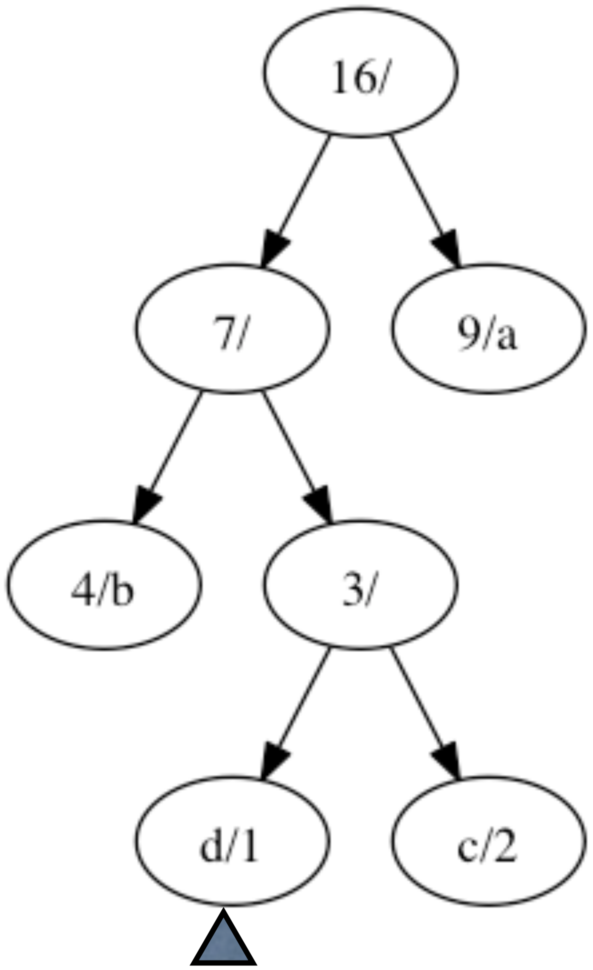
1 1 0 1 0 0 0

Can we navigate this?



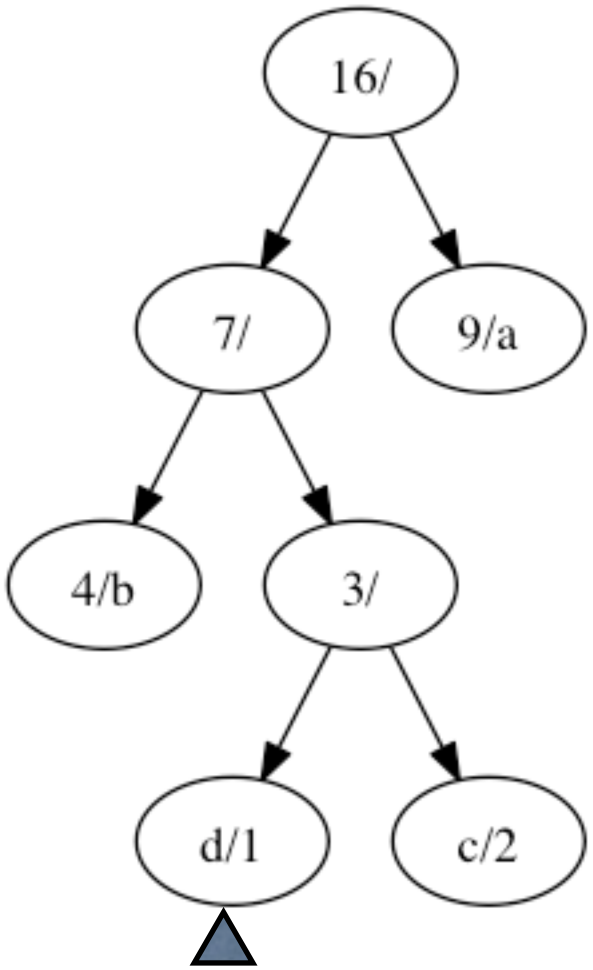
1101000

Can we navigate this?



1101000

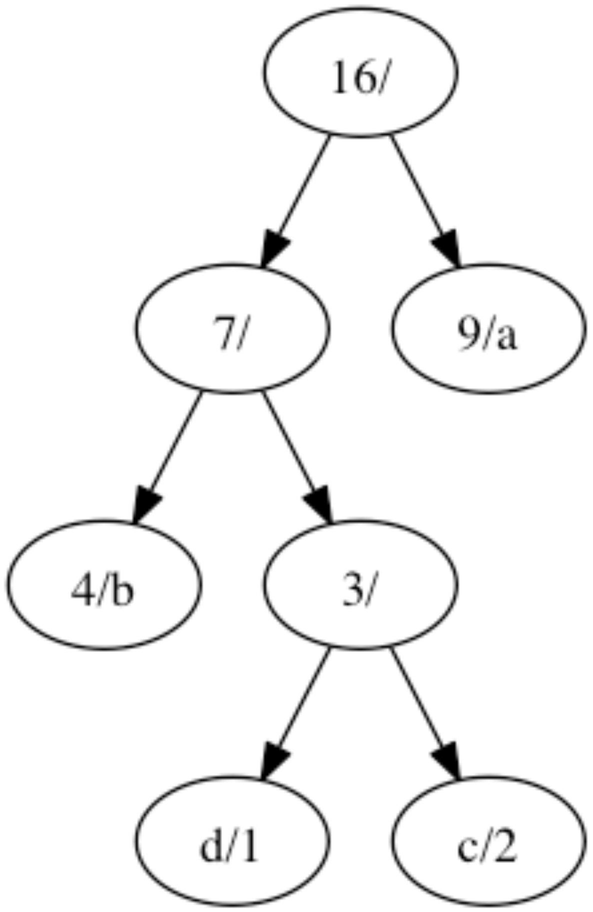
Can we navigate this?



1101000

Who’s my parent?

Can we navigate this?

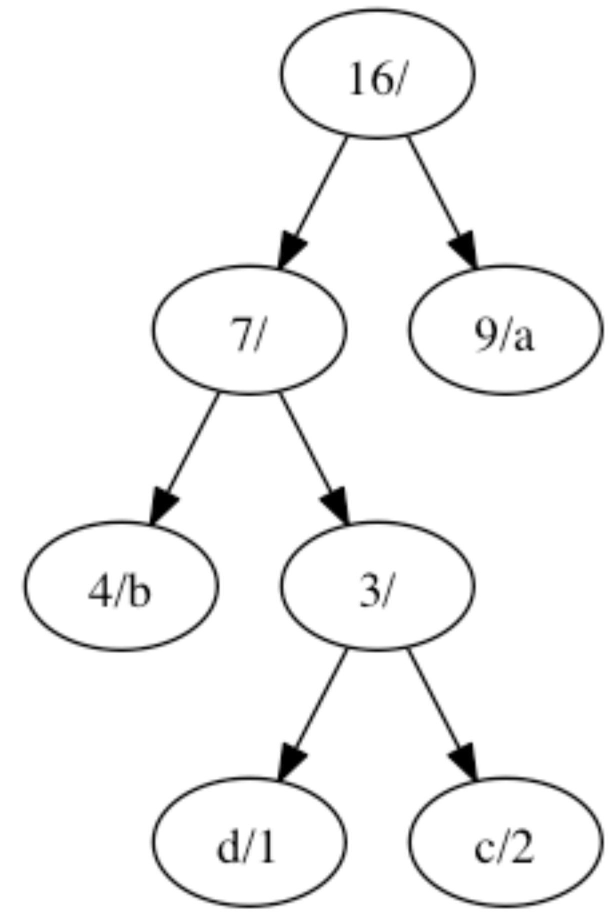


1101000

Who’s my parent?

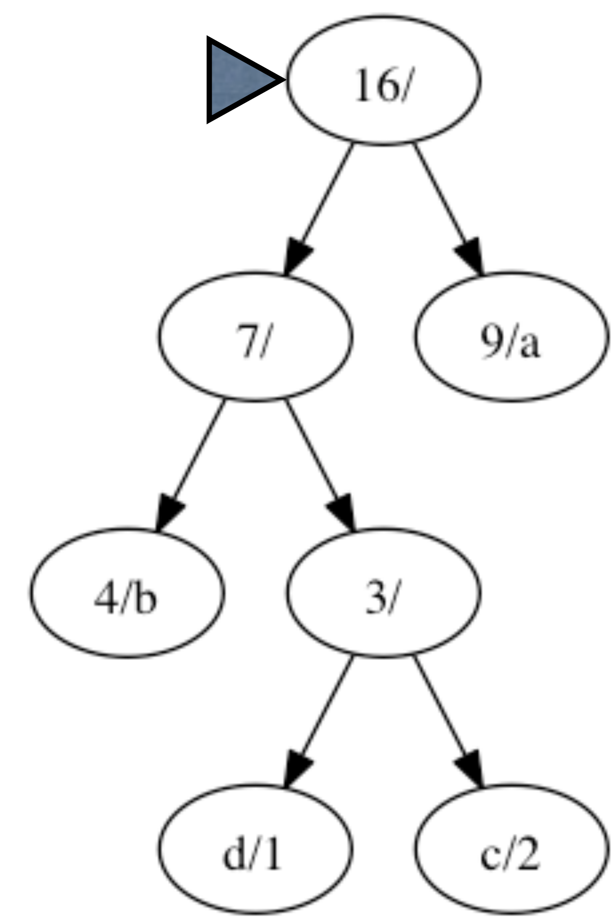
Another try

- Let’s give our tree another try
- Now using BFS



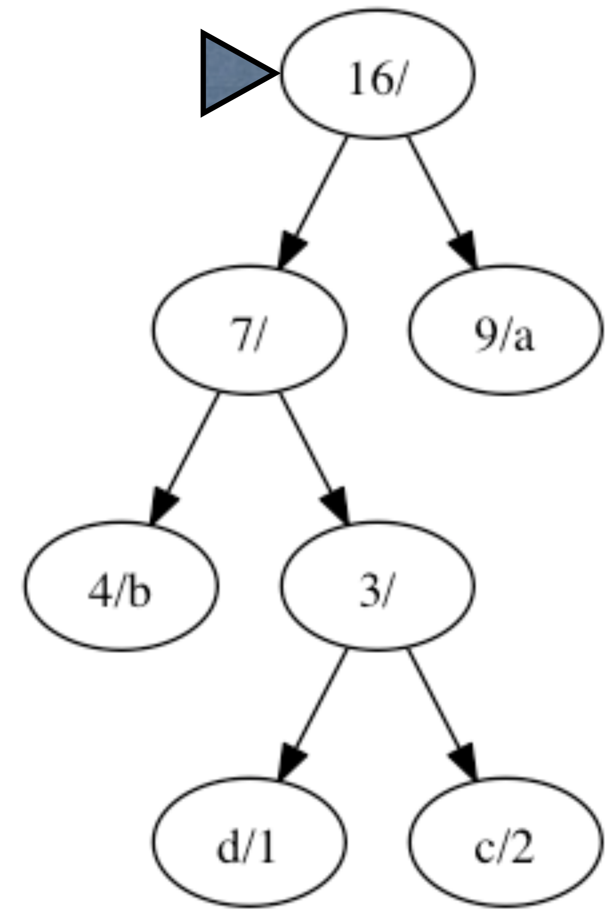
Another try

- Let's give our tree another try
- Now using BFS



Another try

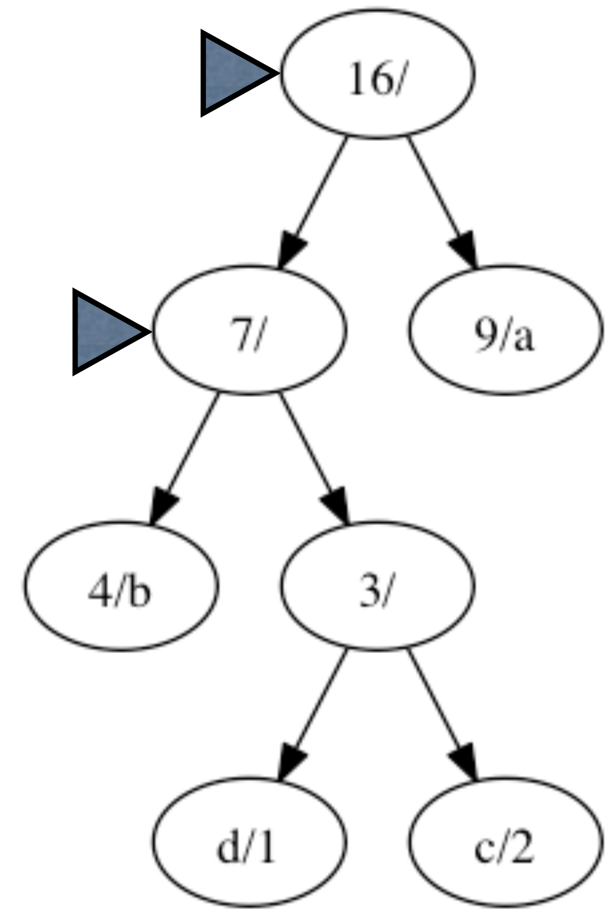
- Let's give our tree another try
- Now using BFS



I

Another try

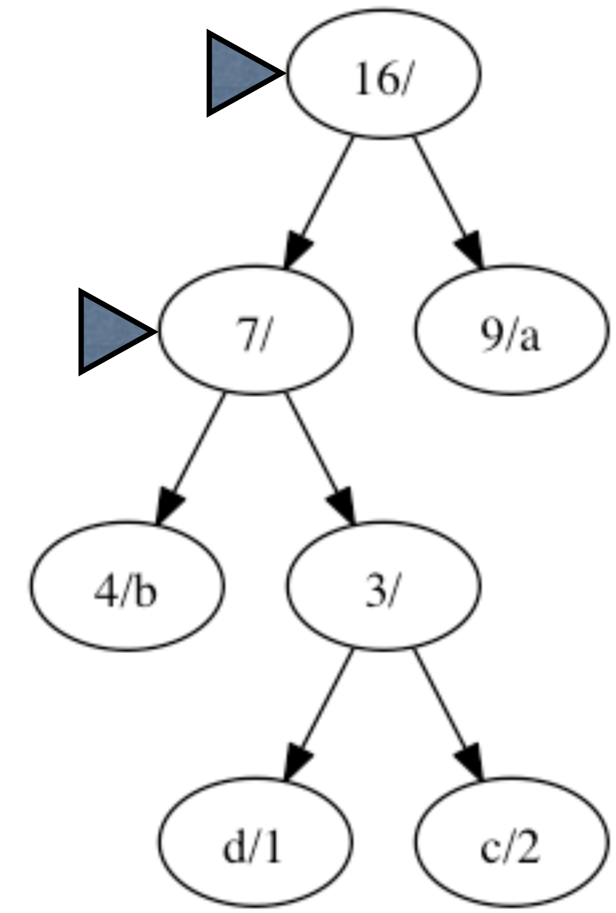
- Let's give our tree another try
- Now using BFS



1

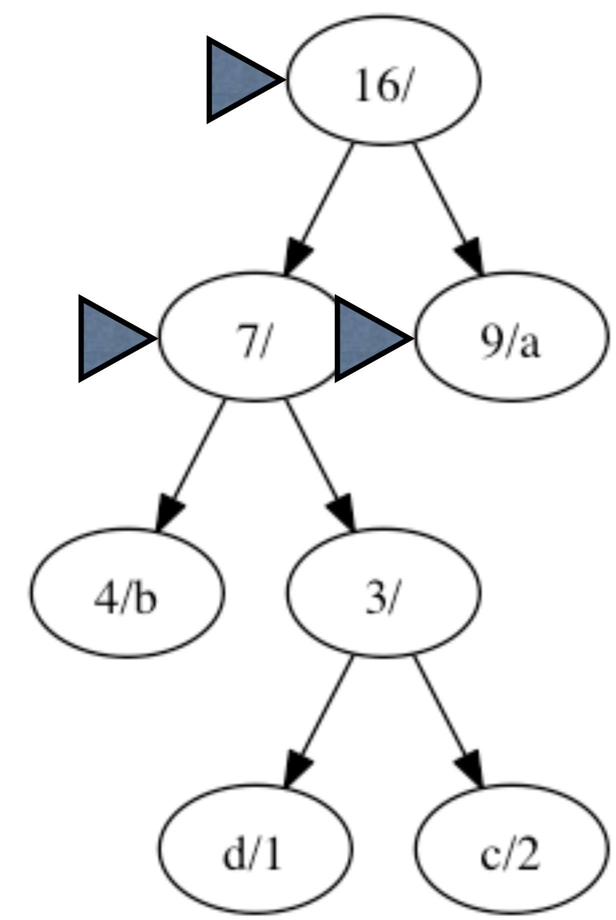
Another try

- Let's give our tree another try
- Now using BFS



Another try

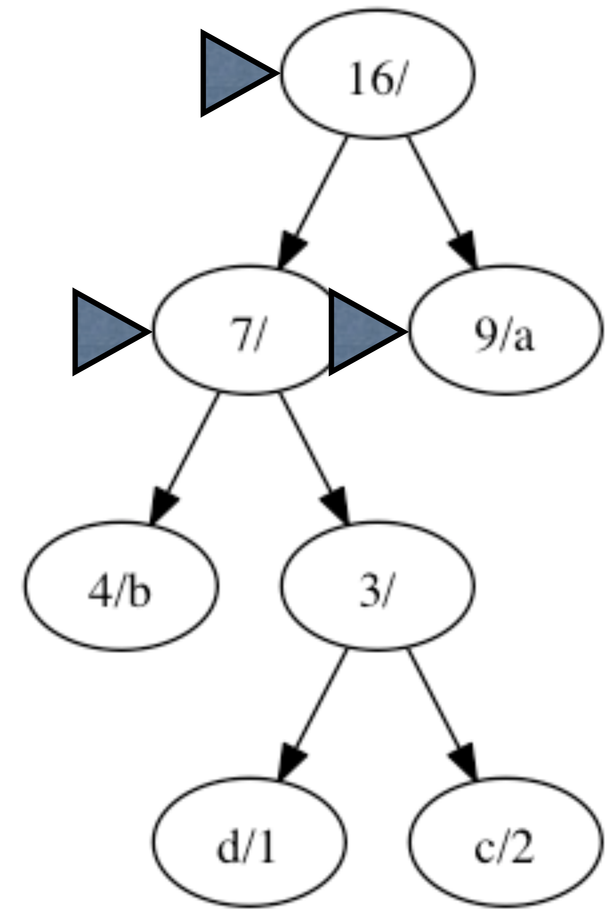
- Let’s give our tree another try
- Now using BFS



Another try

- Let's give our tree another try
- Now using BFS

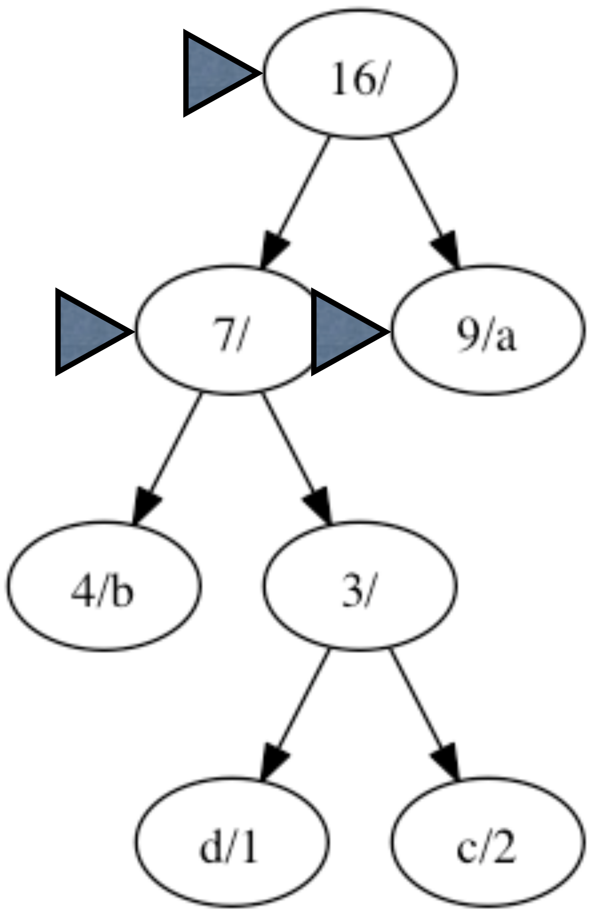
110



Another try

- Let's give our tree another try
- Now using BFS

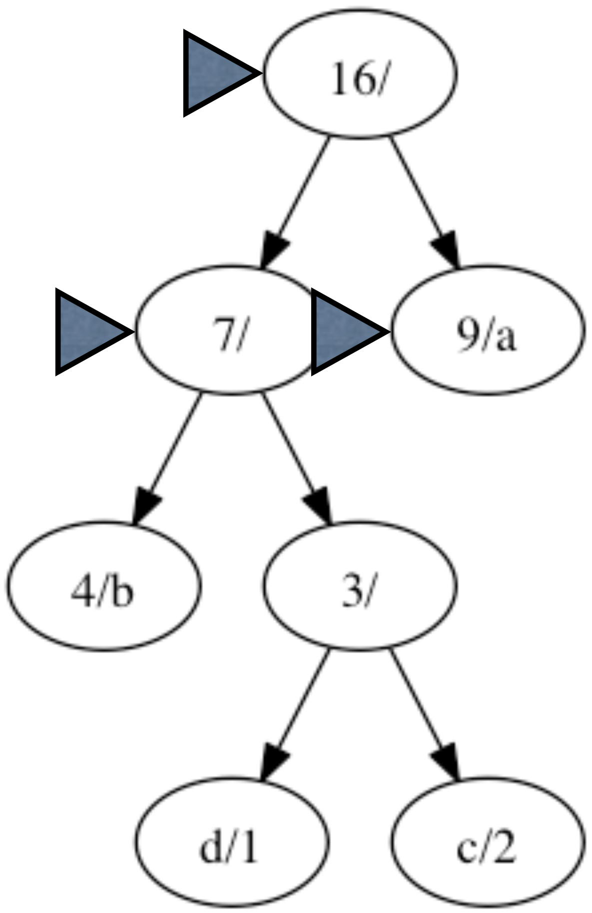
1 1 0 0



Another try

- Let's give our tree another try
- Now using BFS

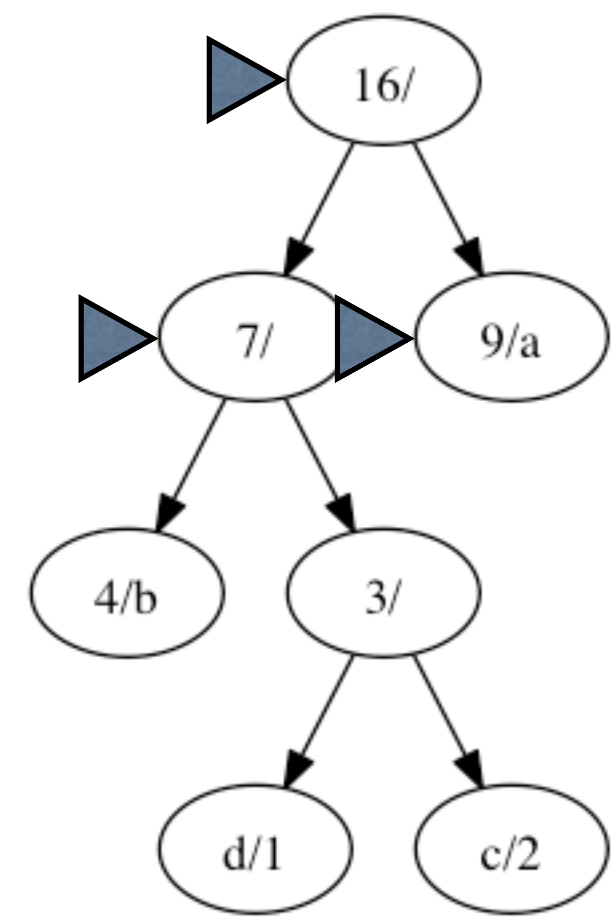
1 1 0 0 1



Another try

- Let's give our tree another try
- Now using BFS

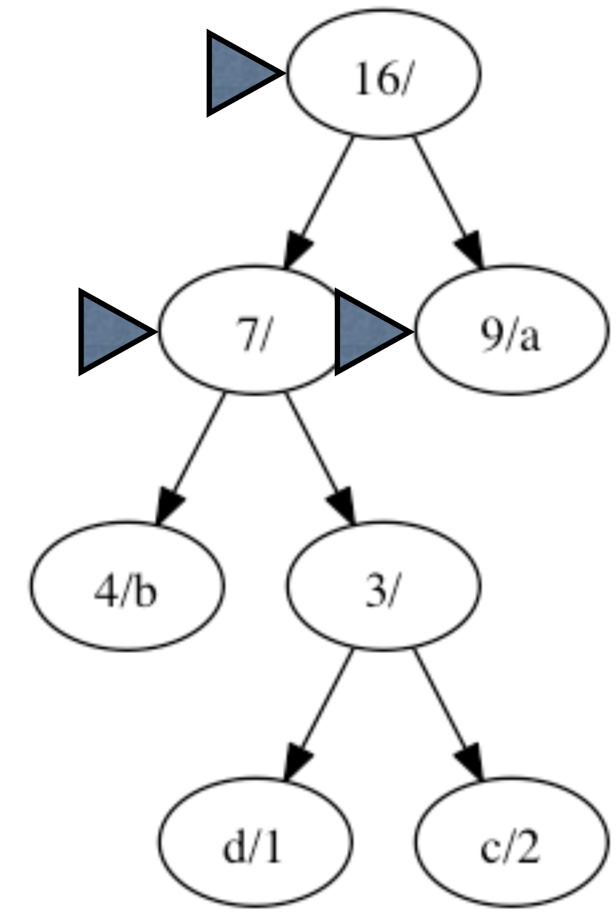
1 1 0 0 1 0



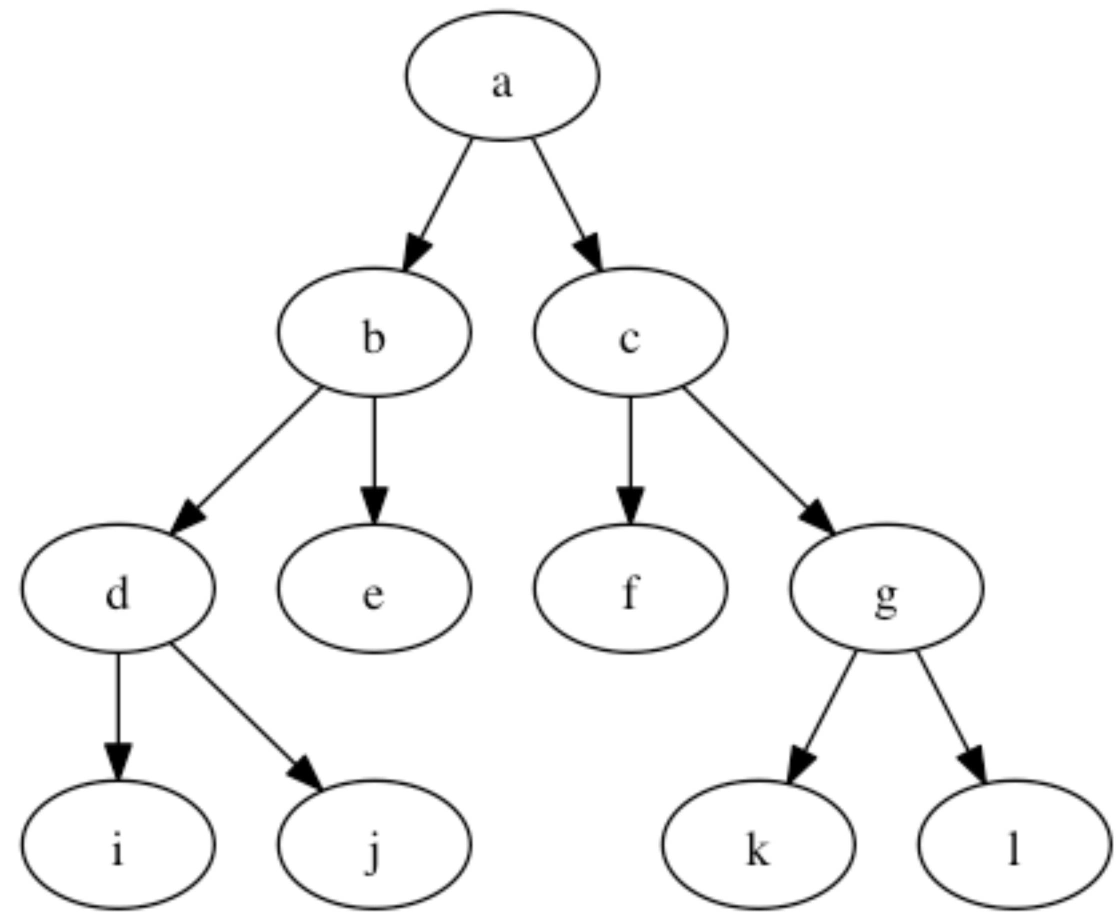
Another try

- Let's give our tree another try
- Now using BFS

1100100

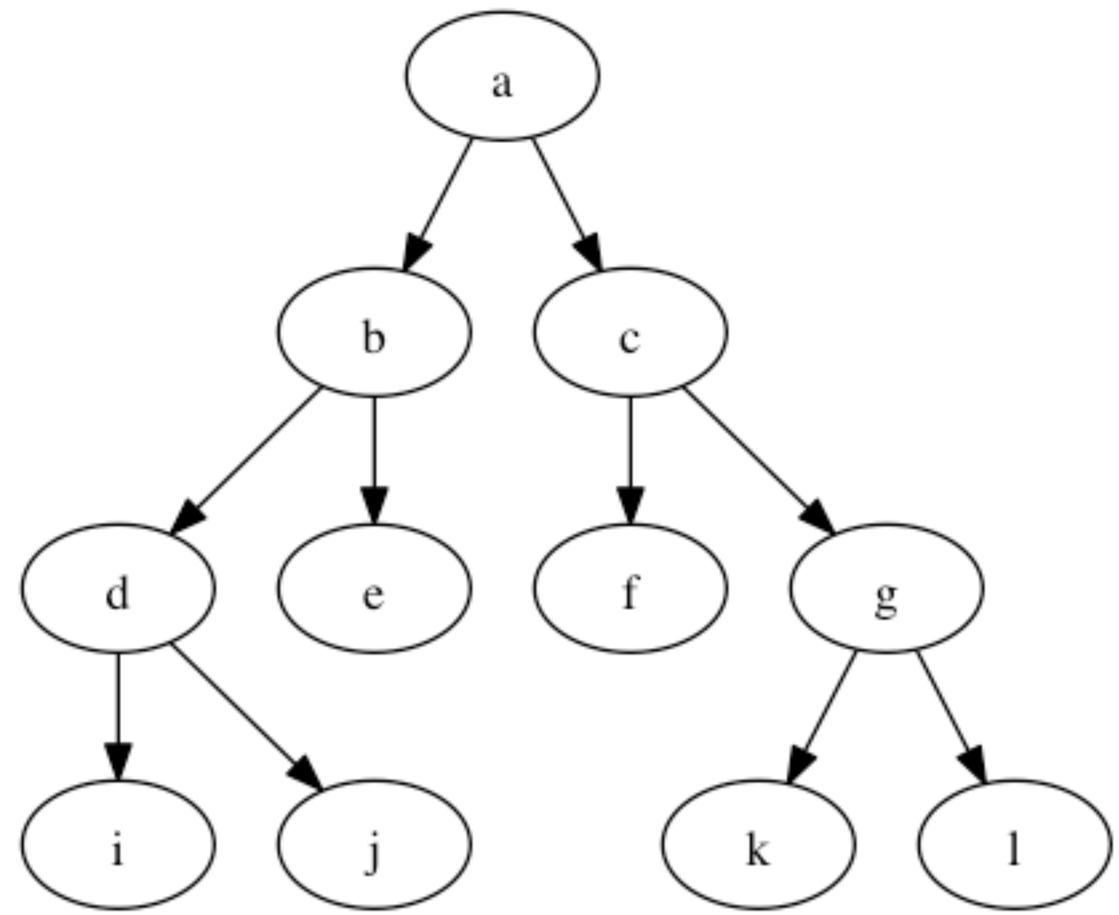


Another try (part 2)



Another try (part 2)

1
11
1001
0000



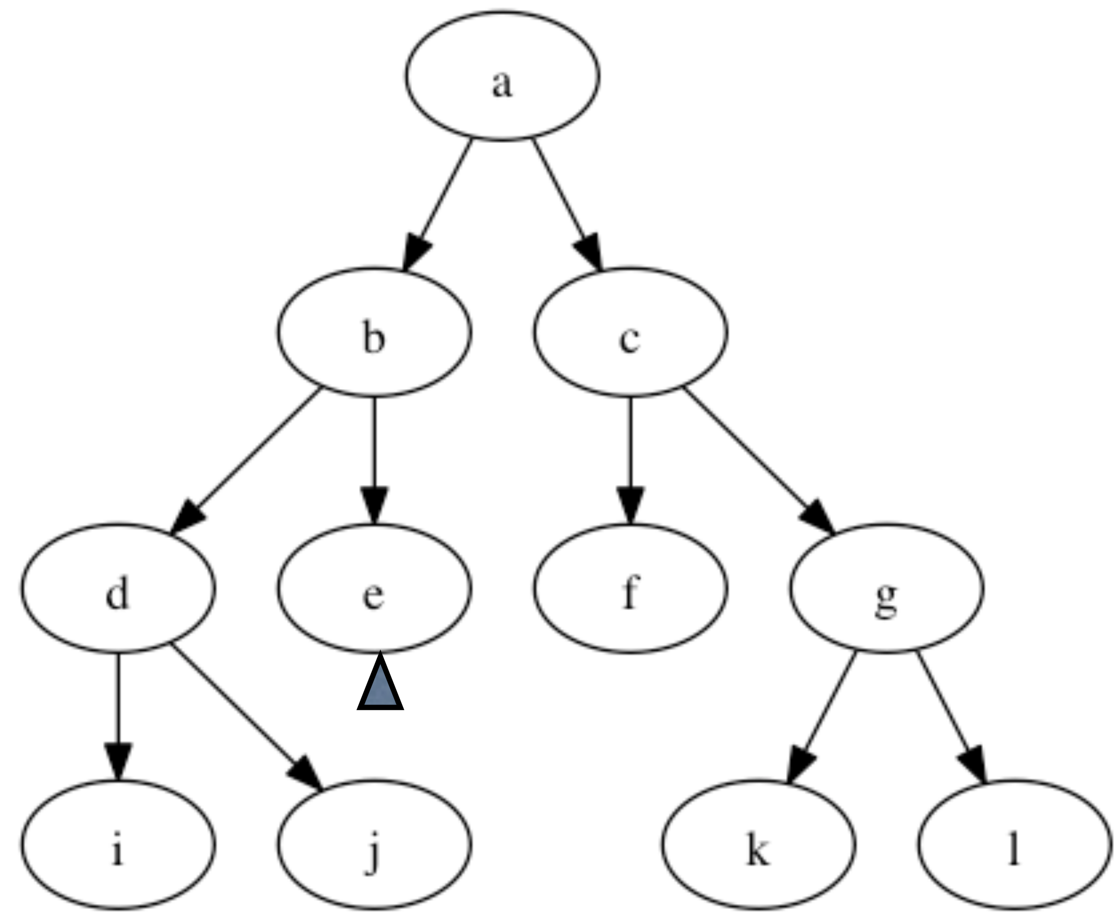
Another try (part 2)

1

11

1001

0000



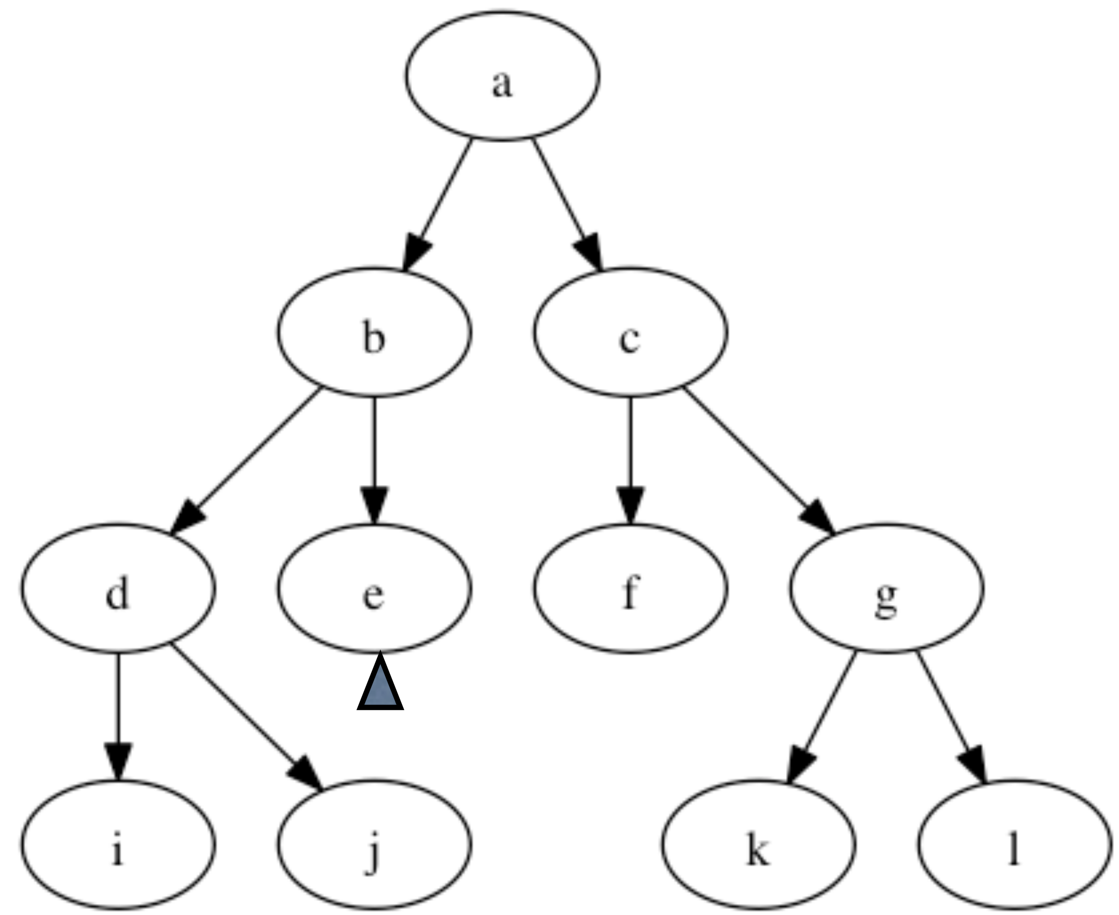
Another try (part 2)

1

11

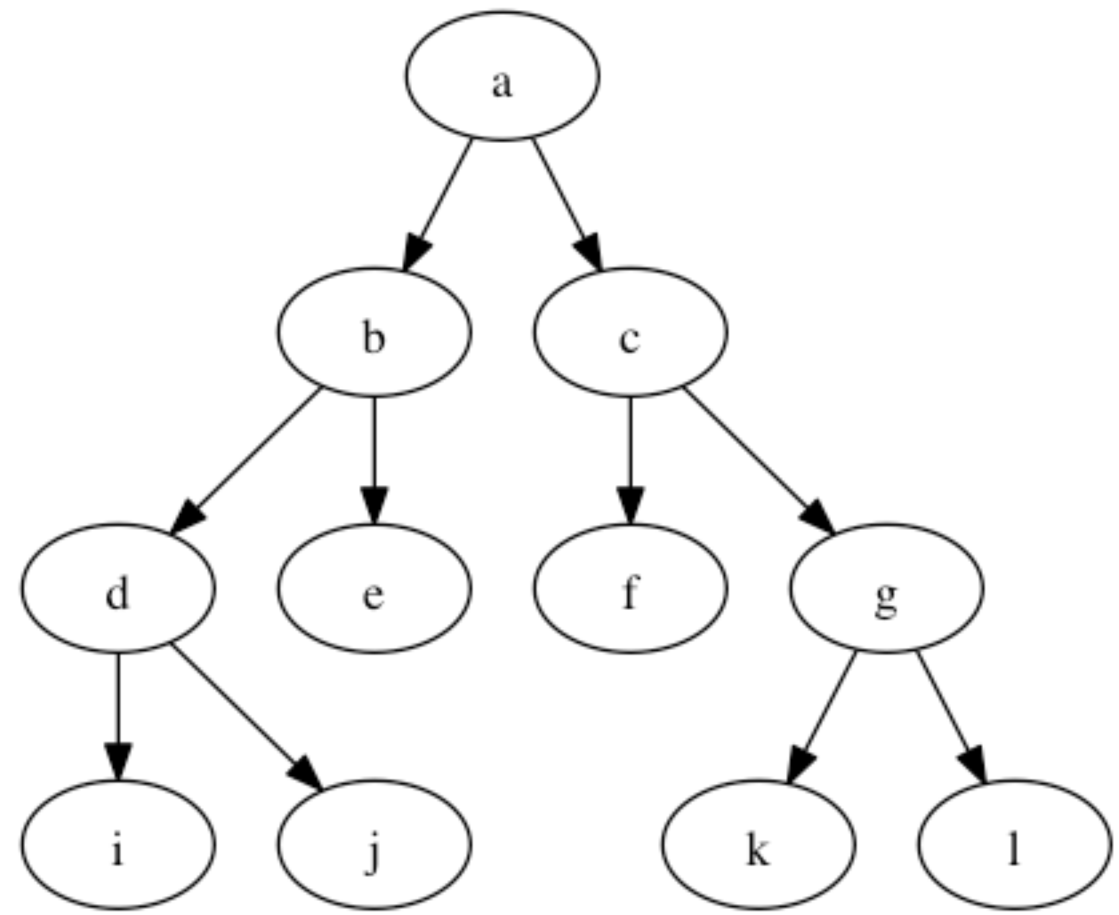
1001

0000



Another try (part 2)

1
11
1001
0000



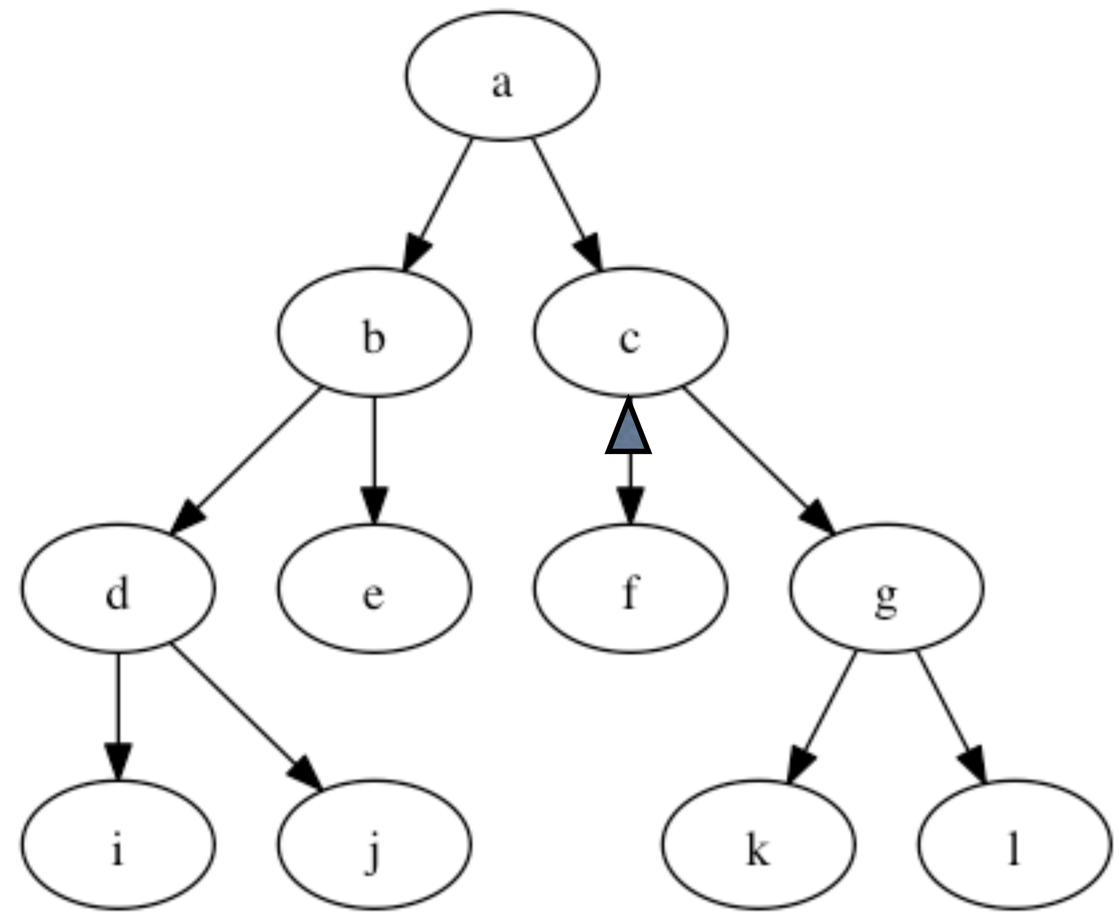
Another try (part 2)

1

1 1

1001

0000



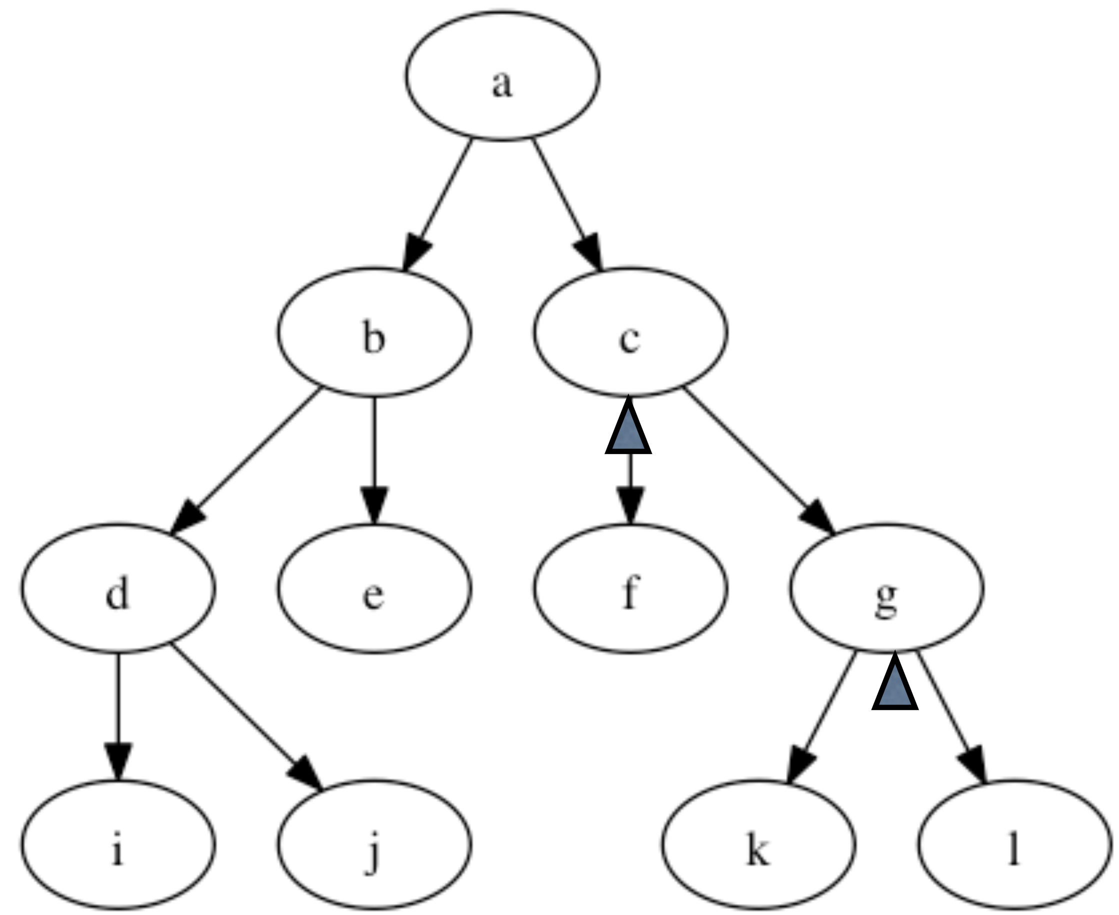
Another try (part 2)

1

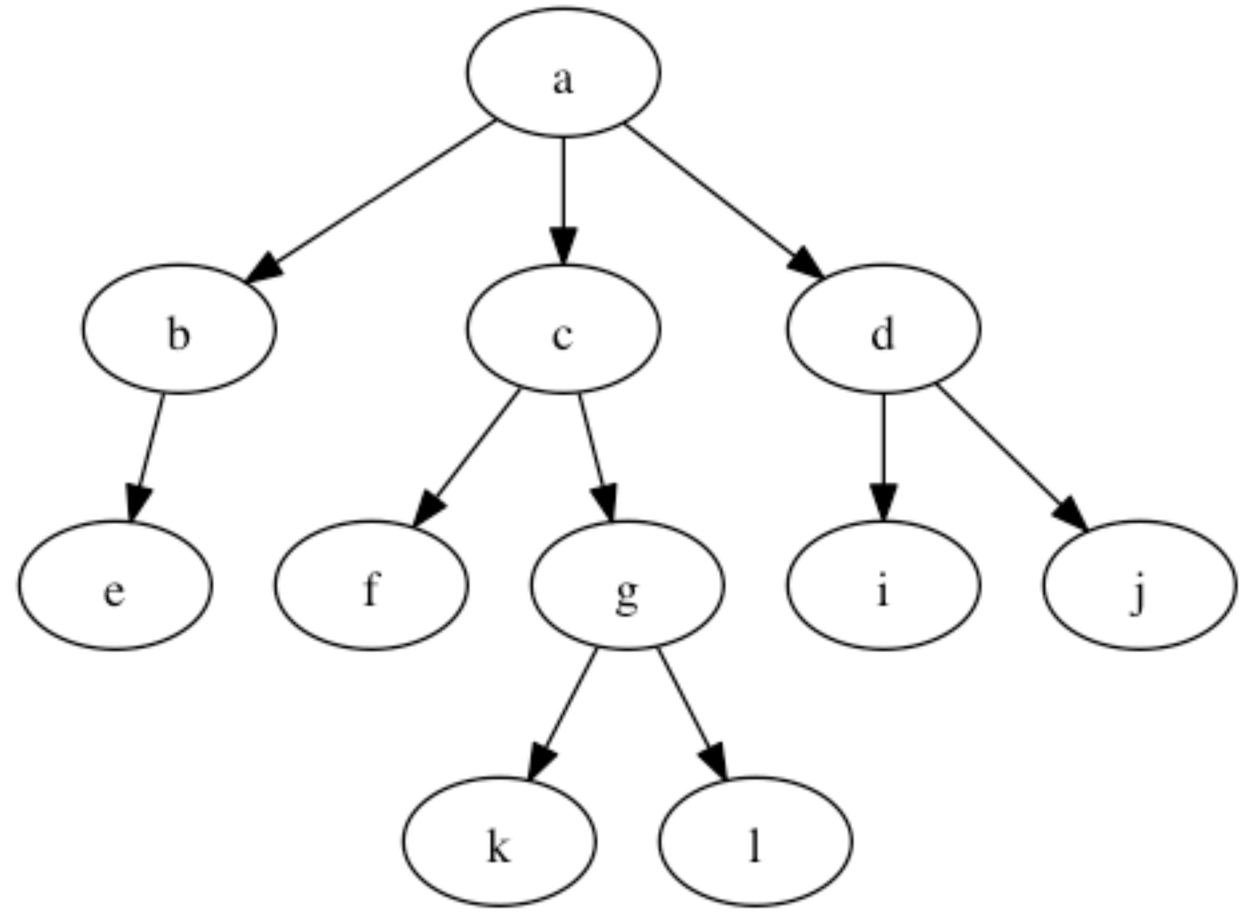
11

1001

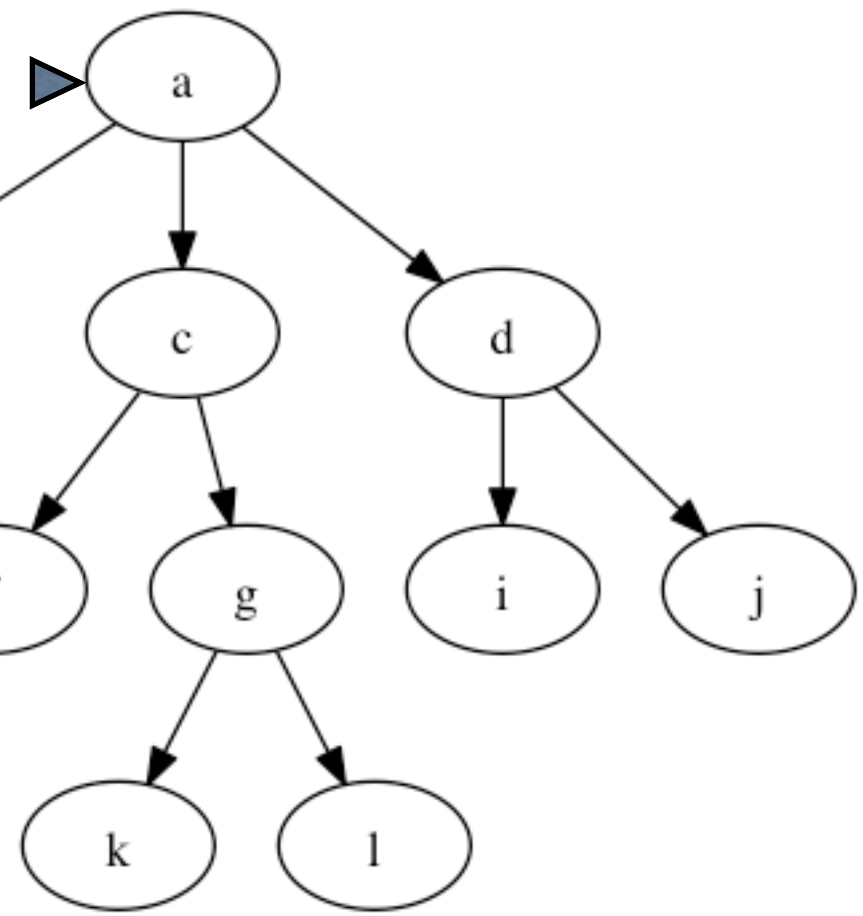
0000



LOUDS

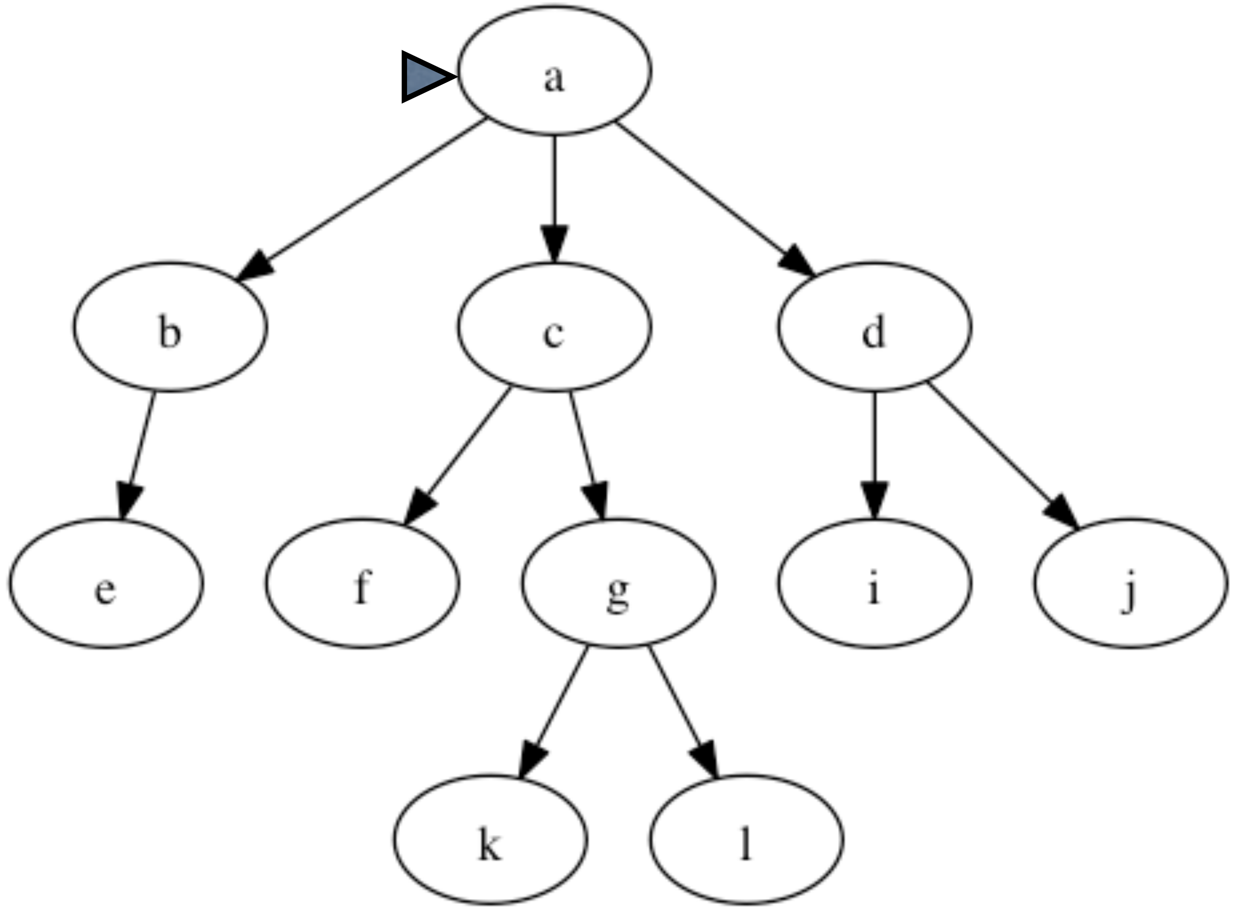


LOUDS



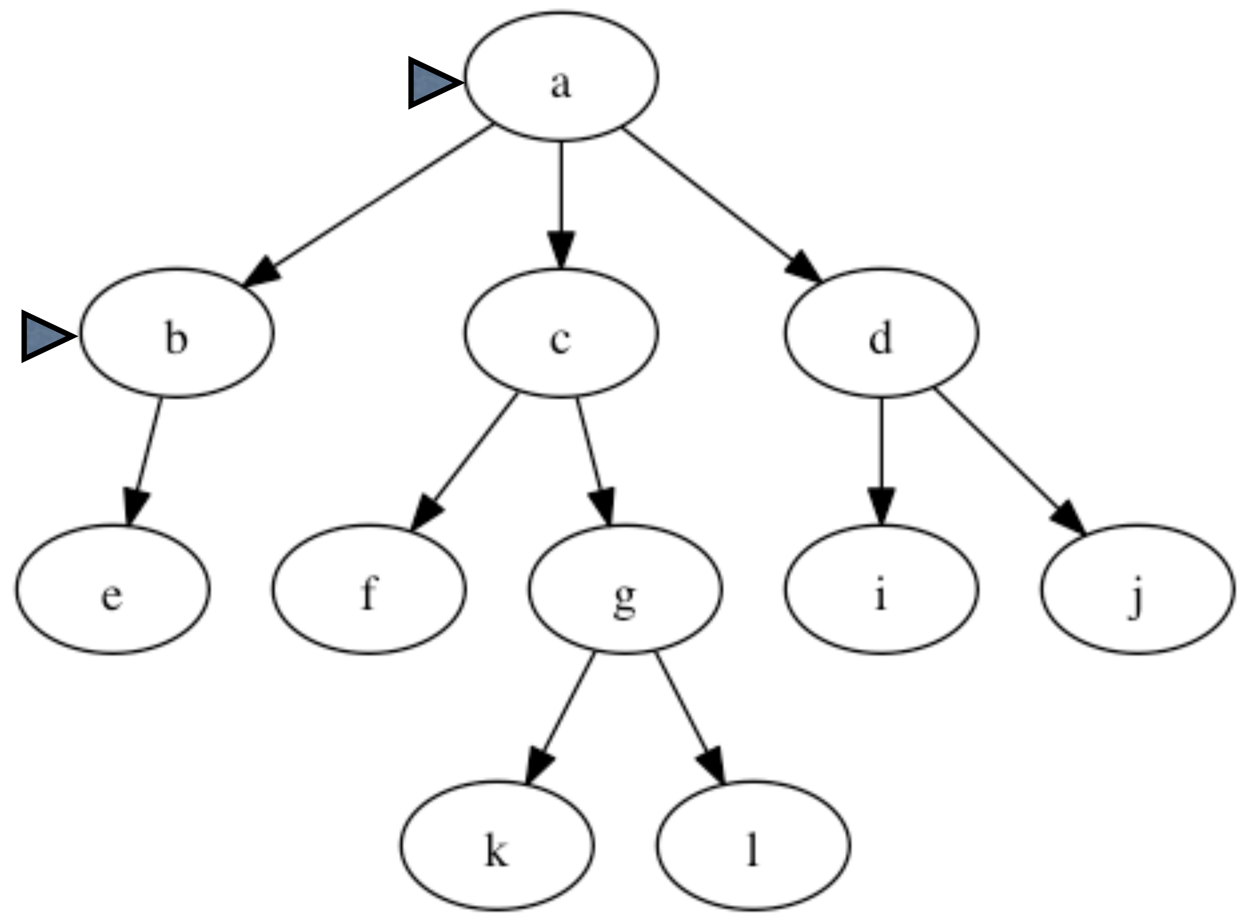
LOUDS

1110



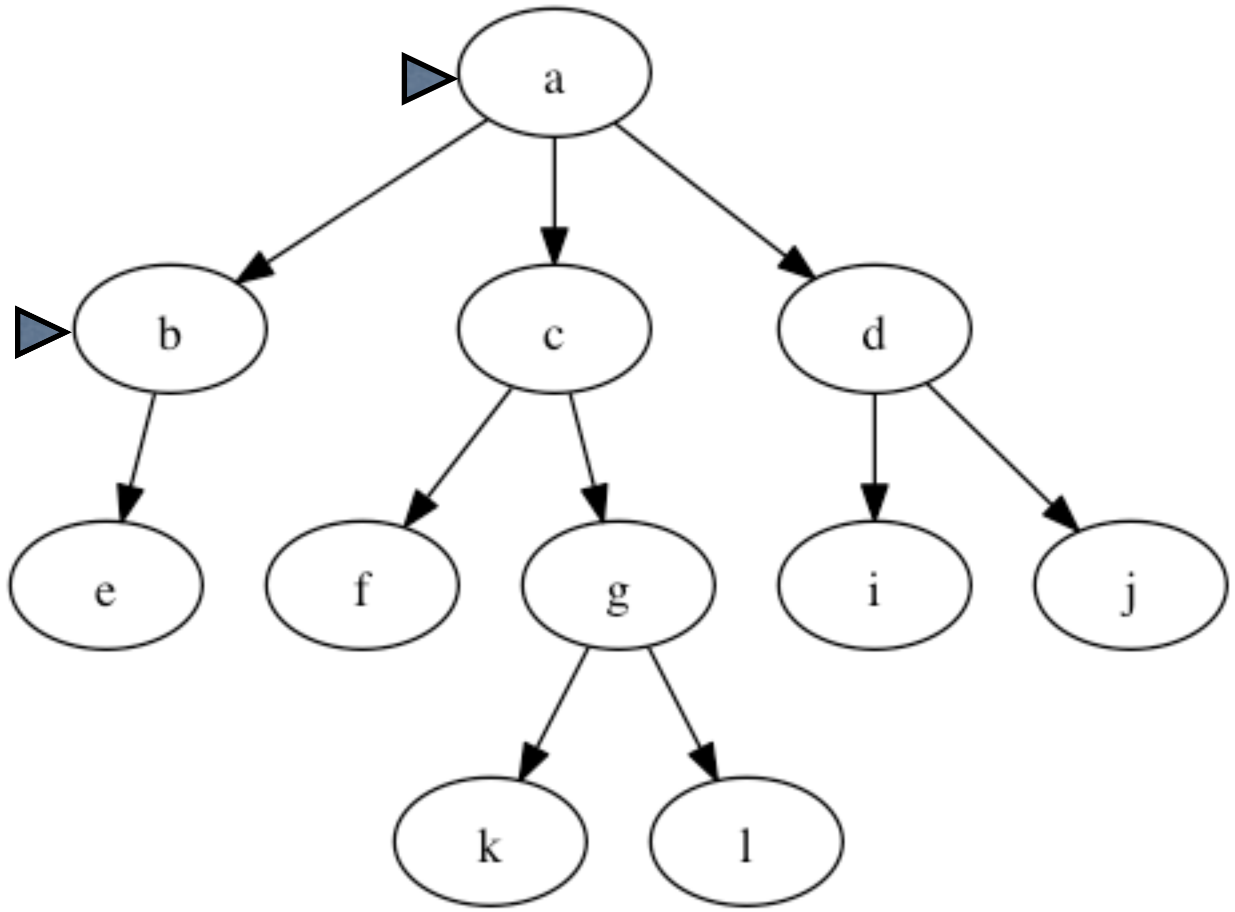
LOUDS

1110



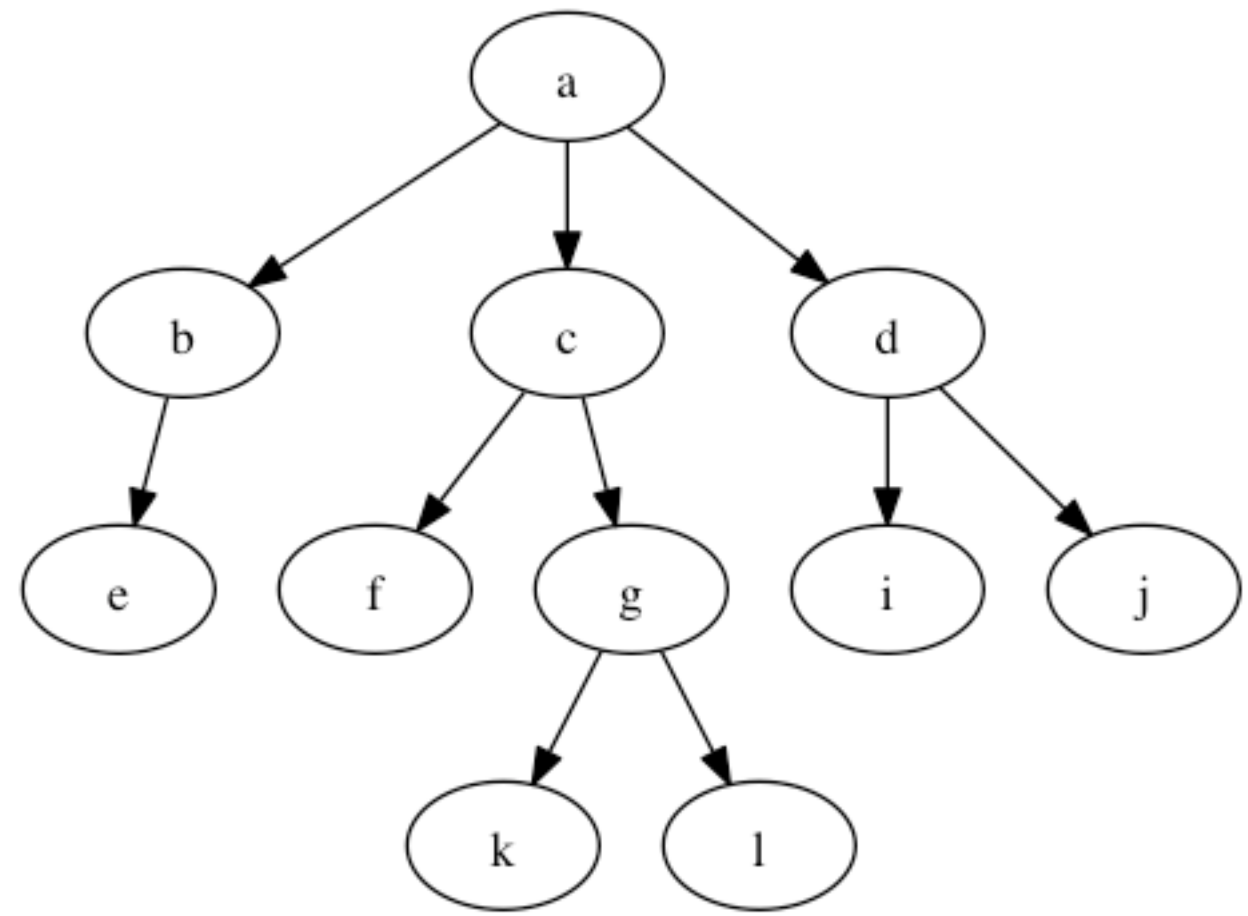
LOUDS

111010



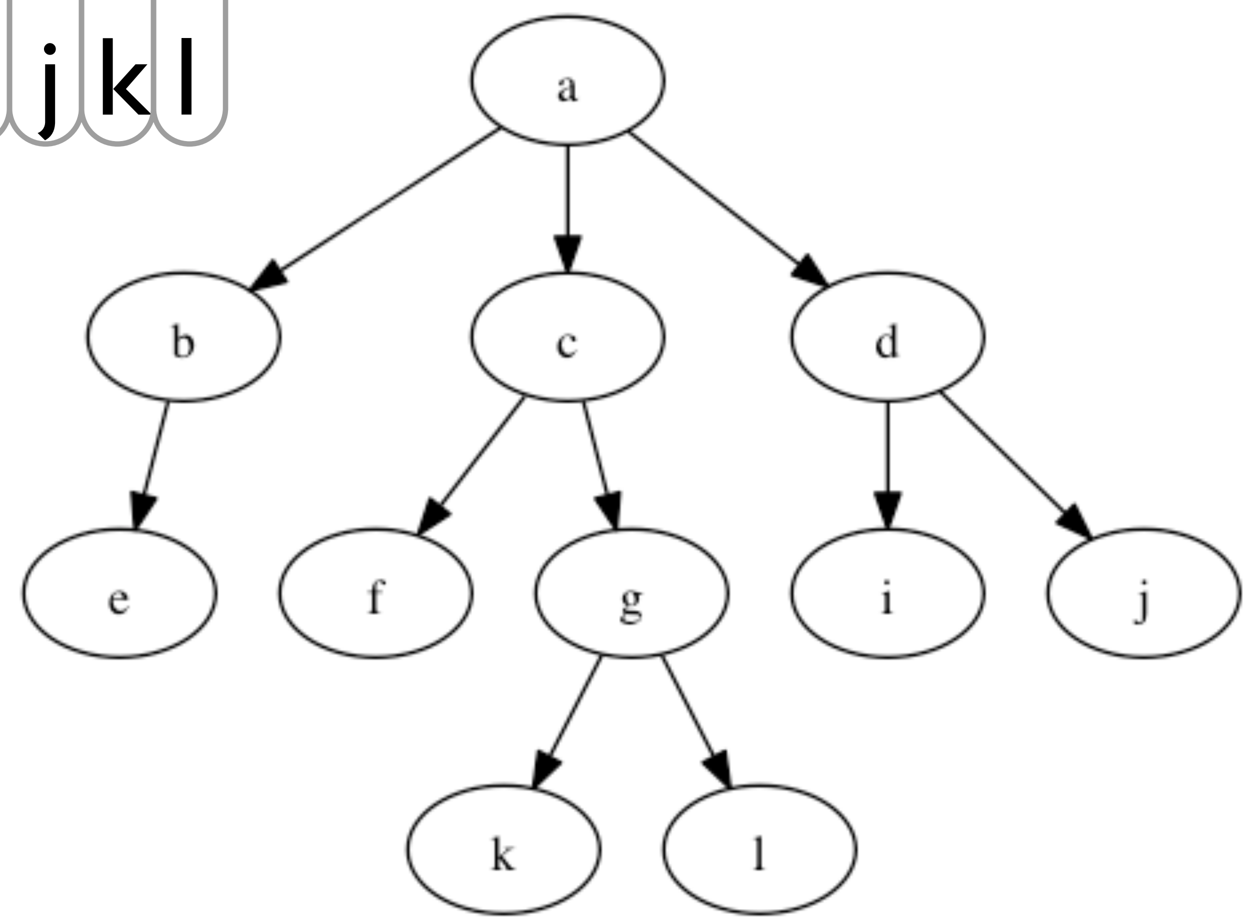
LOUDS

111010110110001100000



LOUDS

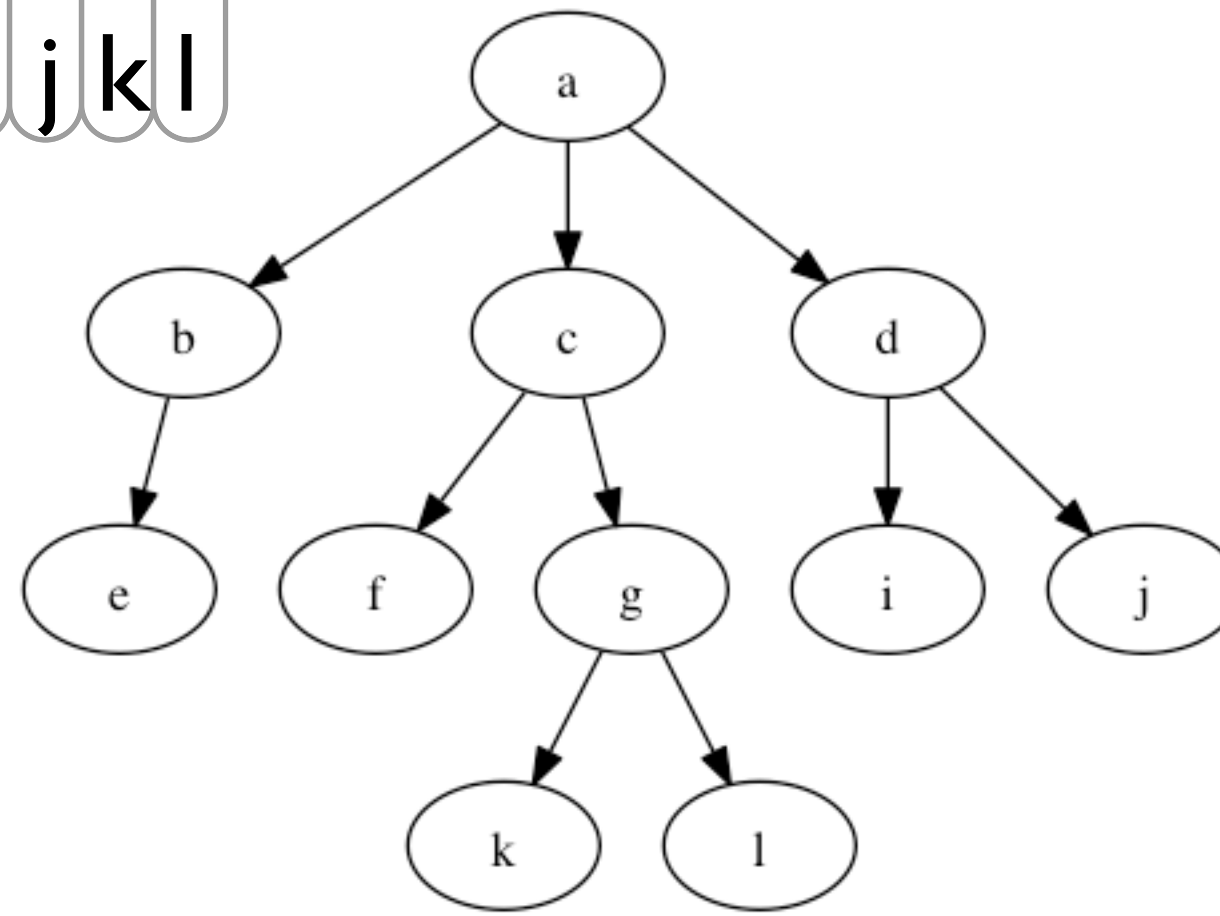
b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l



LOUDS

b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l

Where is the k-th node in BFS order?

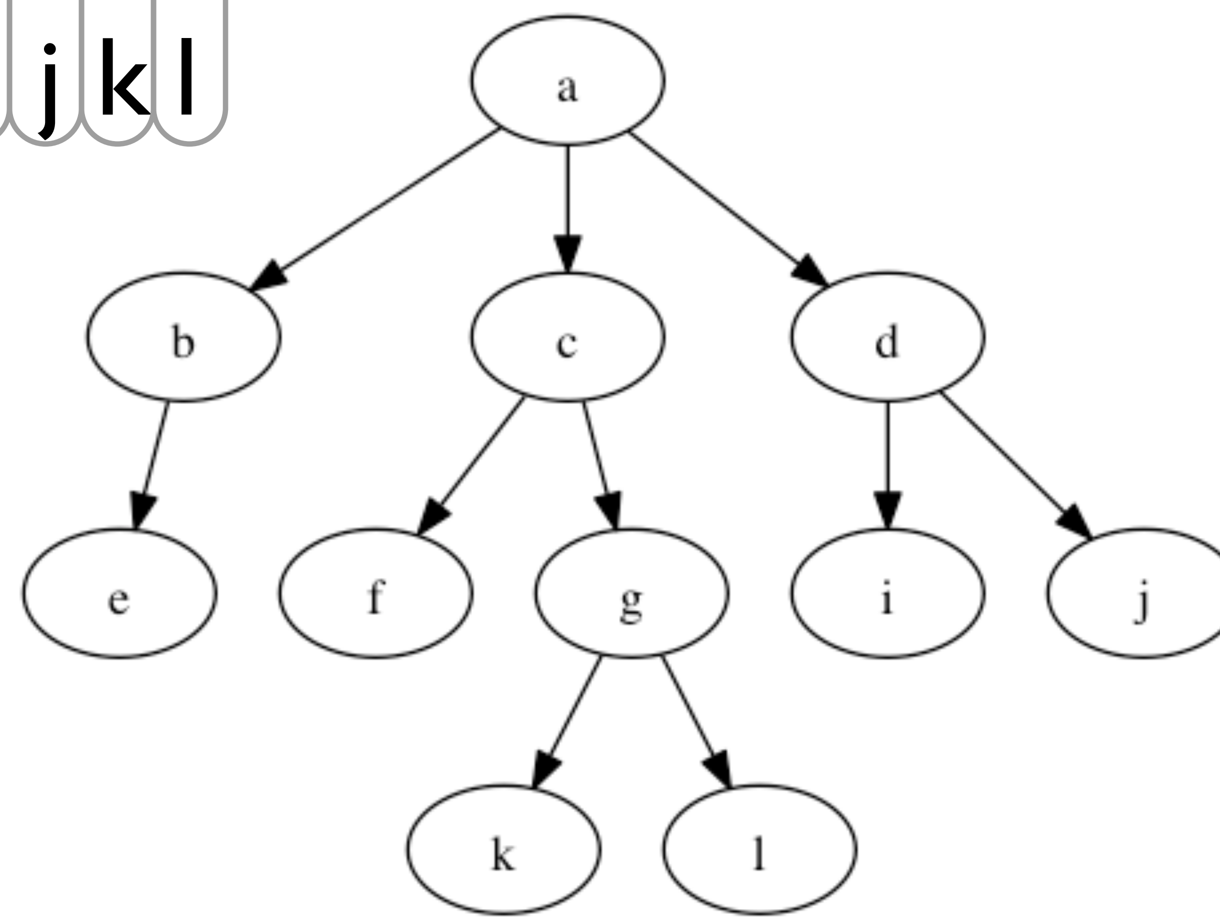


LOUDS

b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l

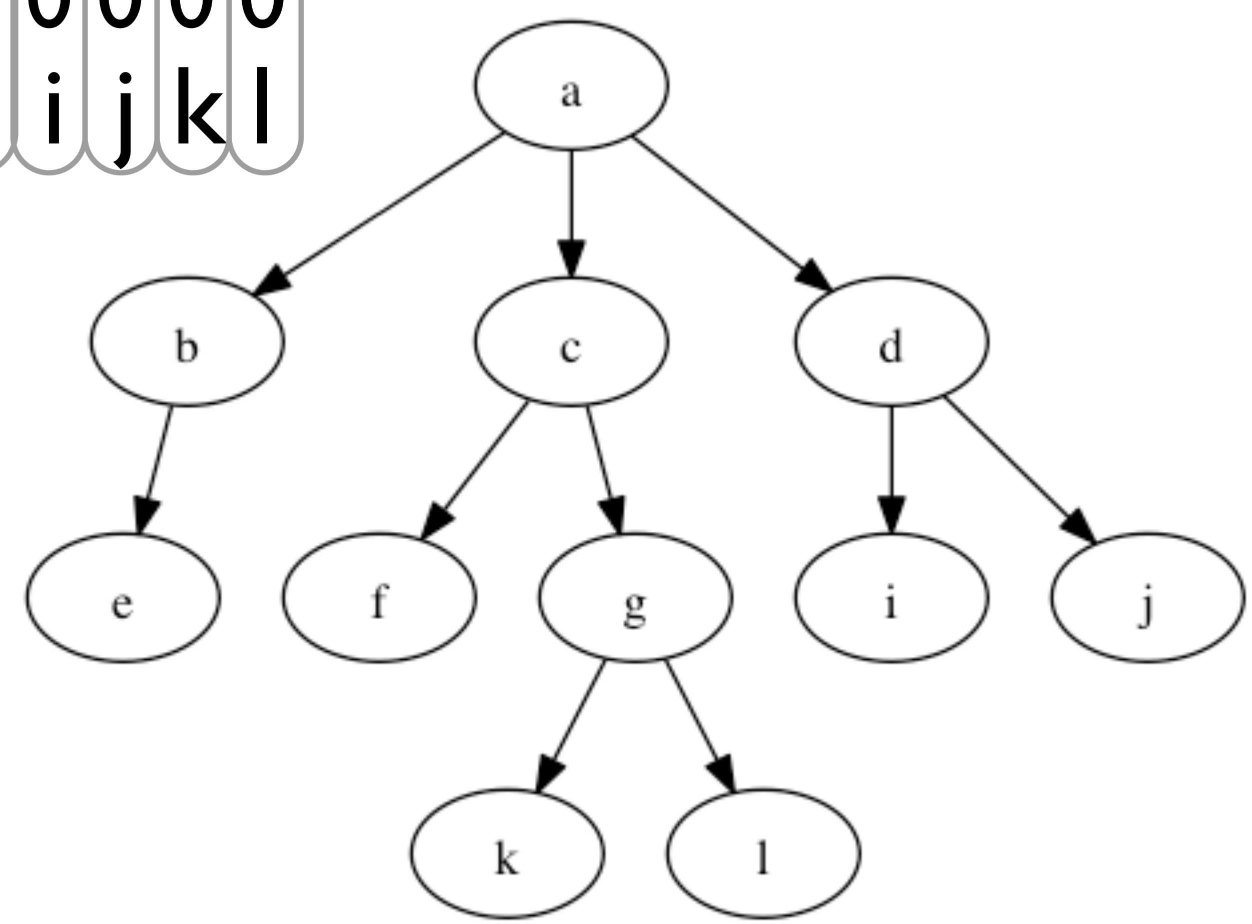
Where is the k-th node in BFS order?

select(0, k)



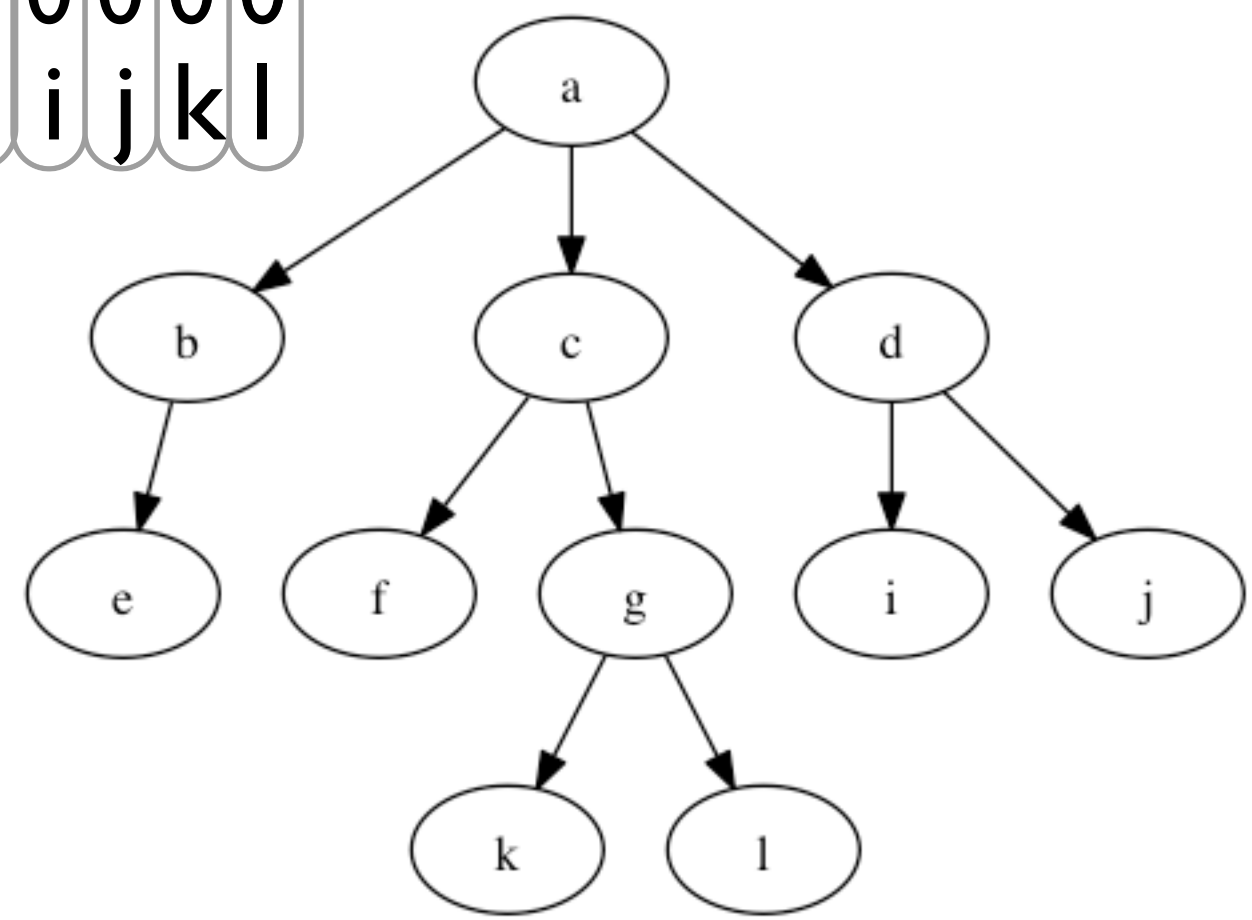
LOUDS

b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l

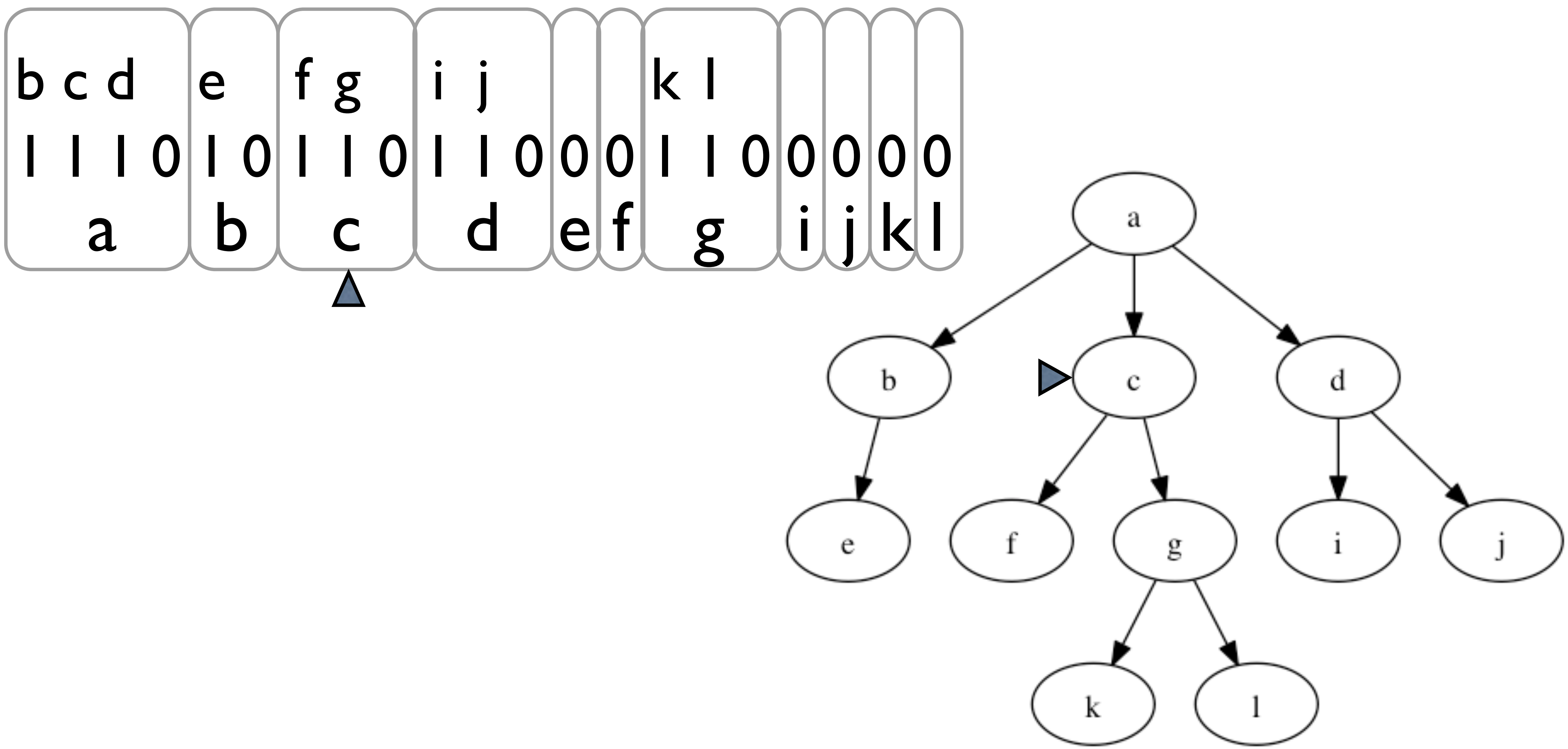


LOUDS (child)

b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l

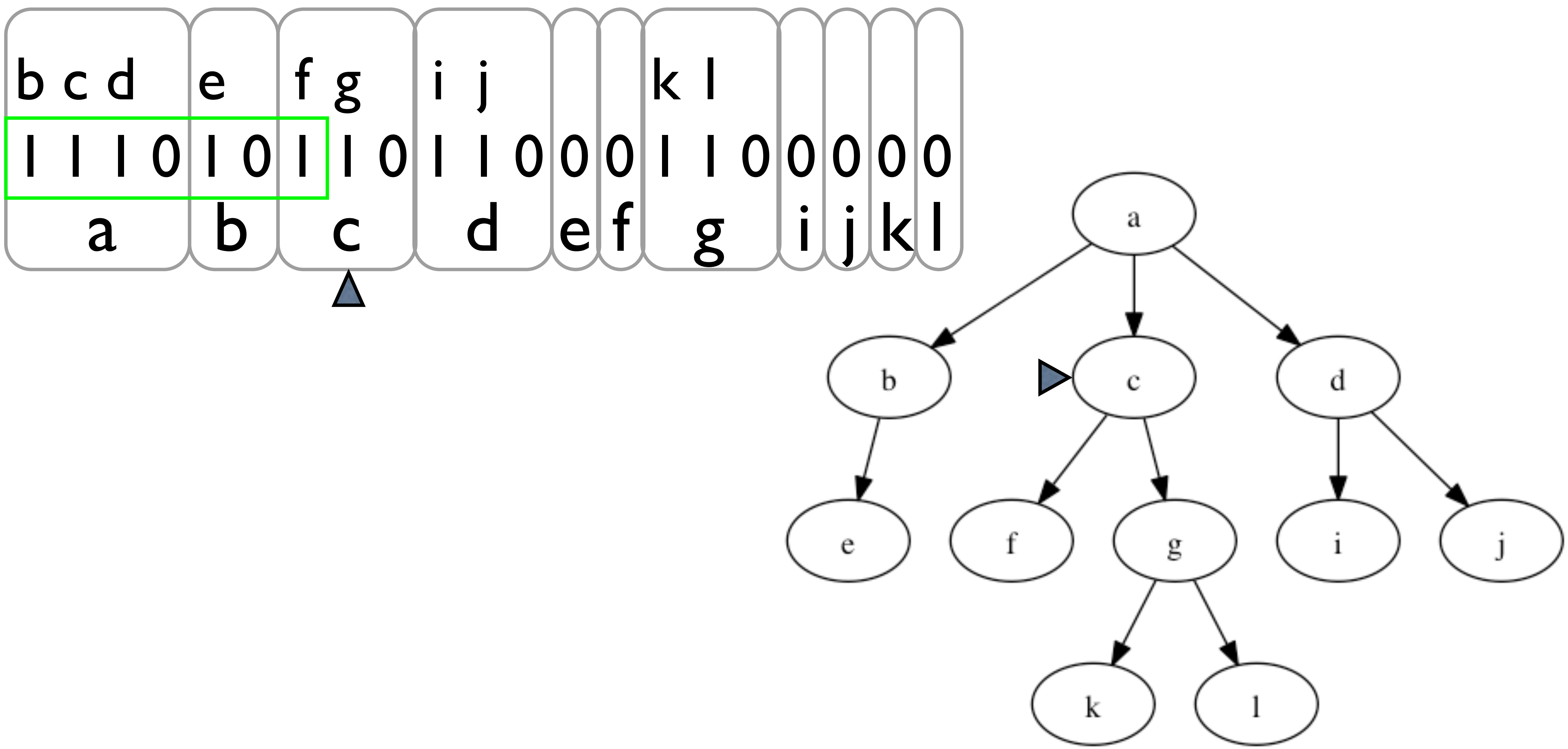


LOUDS (child)



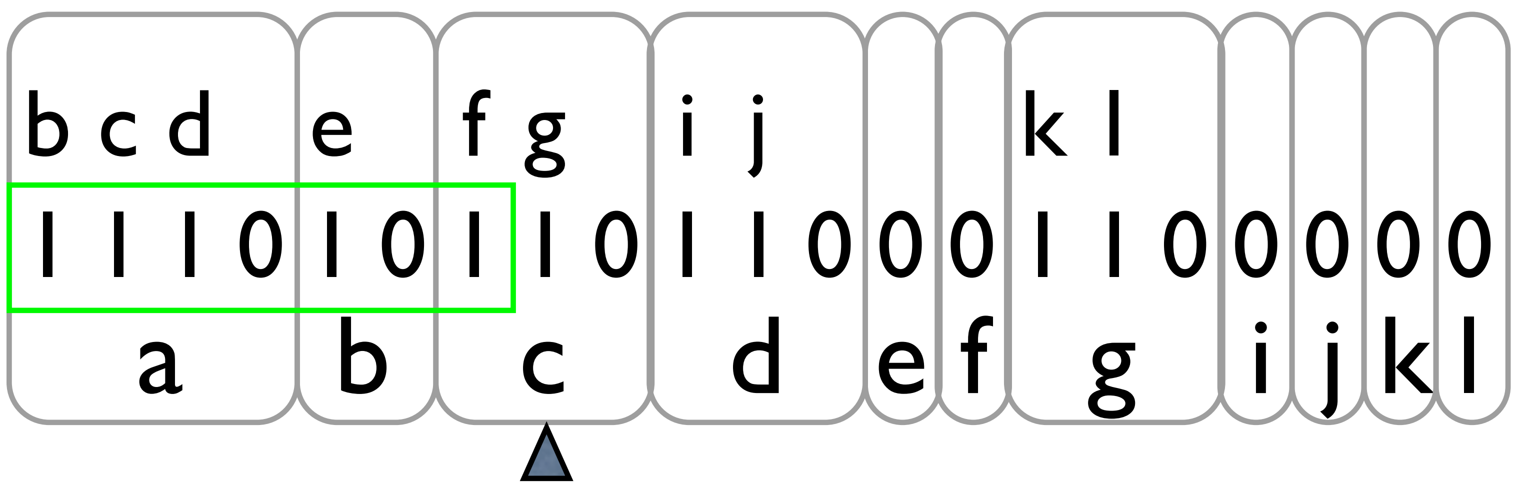
b c d
1 1 1 0
a
e
1 0
b
f g
1 1 0
c
i j
1 1 0
d
0
e
0
f
k l
1 1 0
g
0
i
0
j
0
k
0
l
a
b
c
d
e
f
g
i
j
k
l

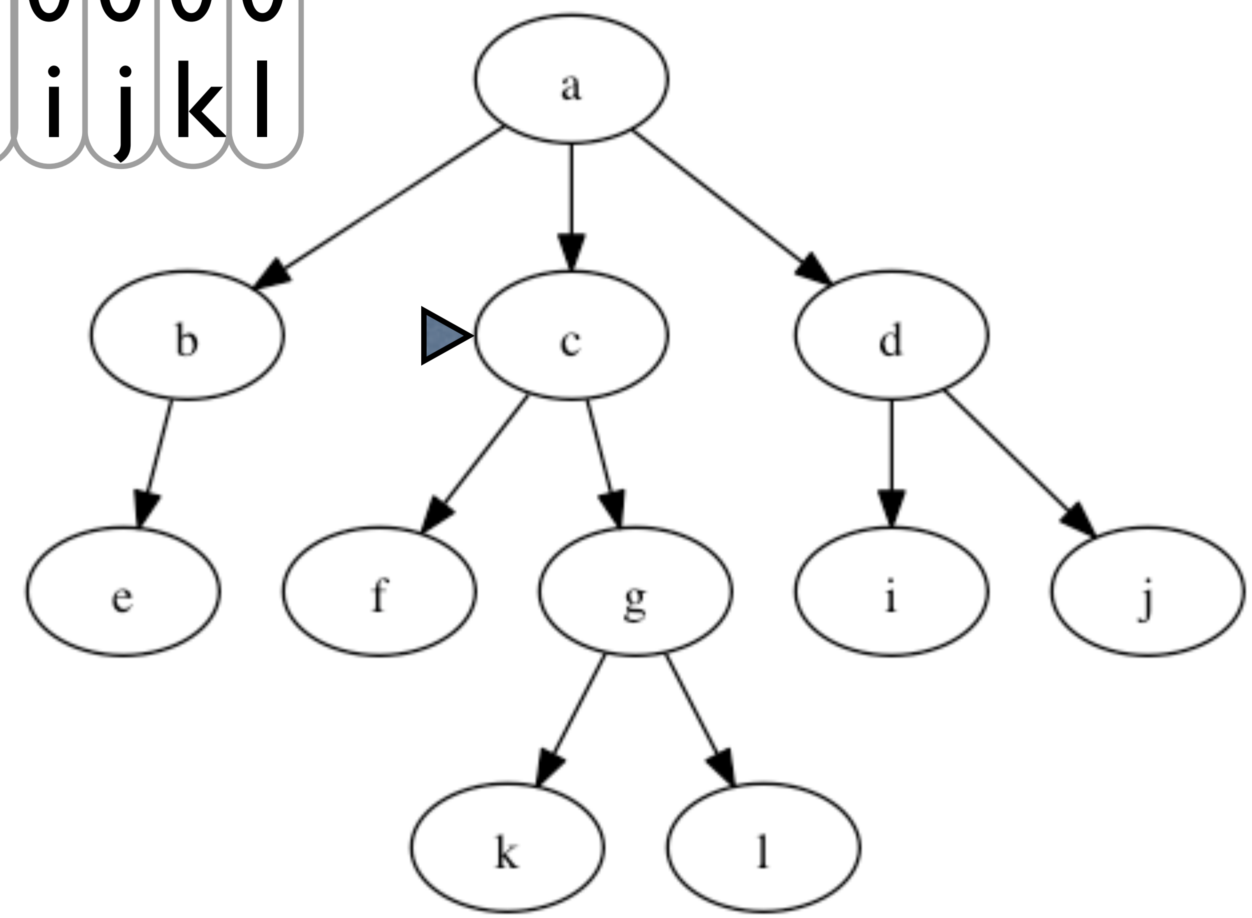
LOUDS (child)



b c d
e
f g
i j
k l
1 1 1 0 1 0 1 1 0 1 1 0 0 0 1 1 0 0 0 0 0
a b c d e f g i j k l
a
b
c
d
e
f
g
i
j
k
l

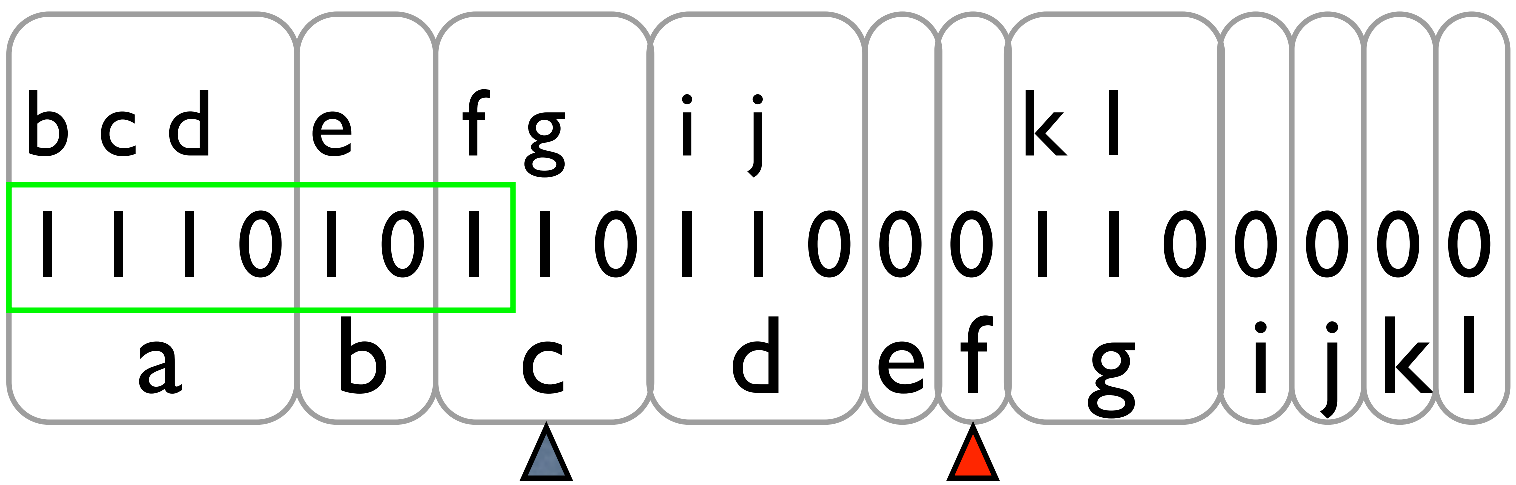
LOUDS (child)





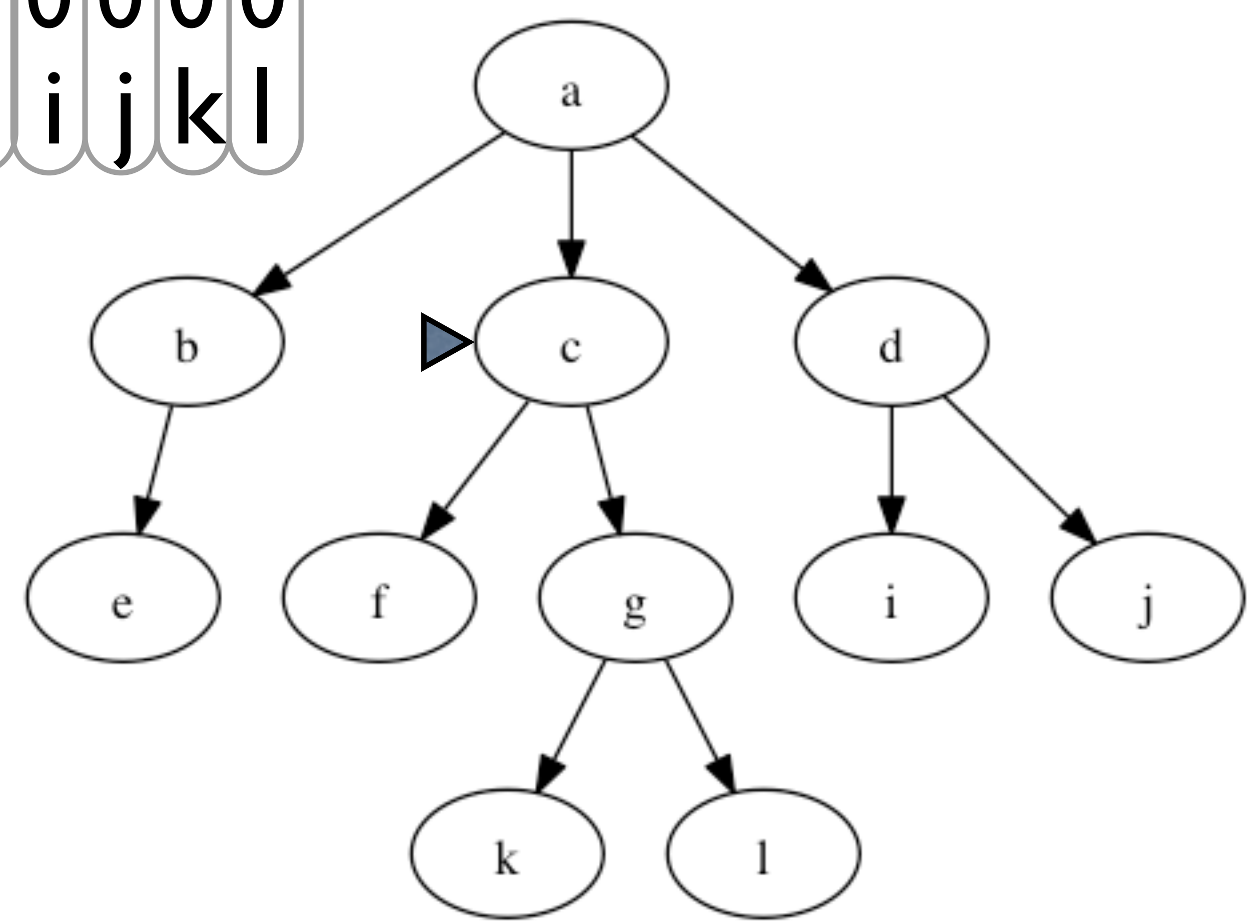
f is the 6th node

LOUDS (child)



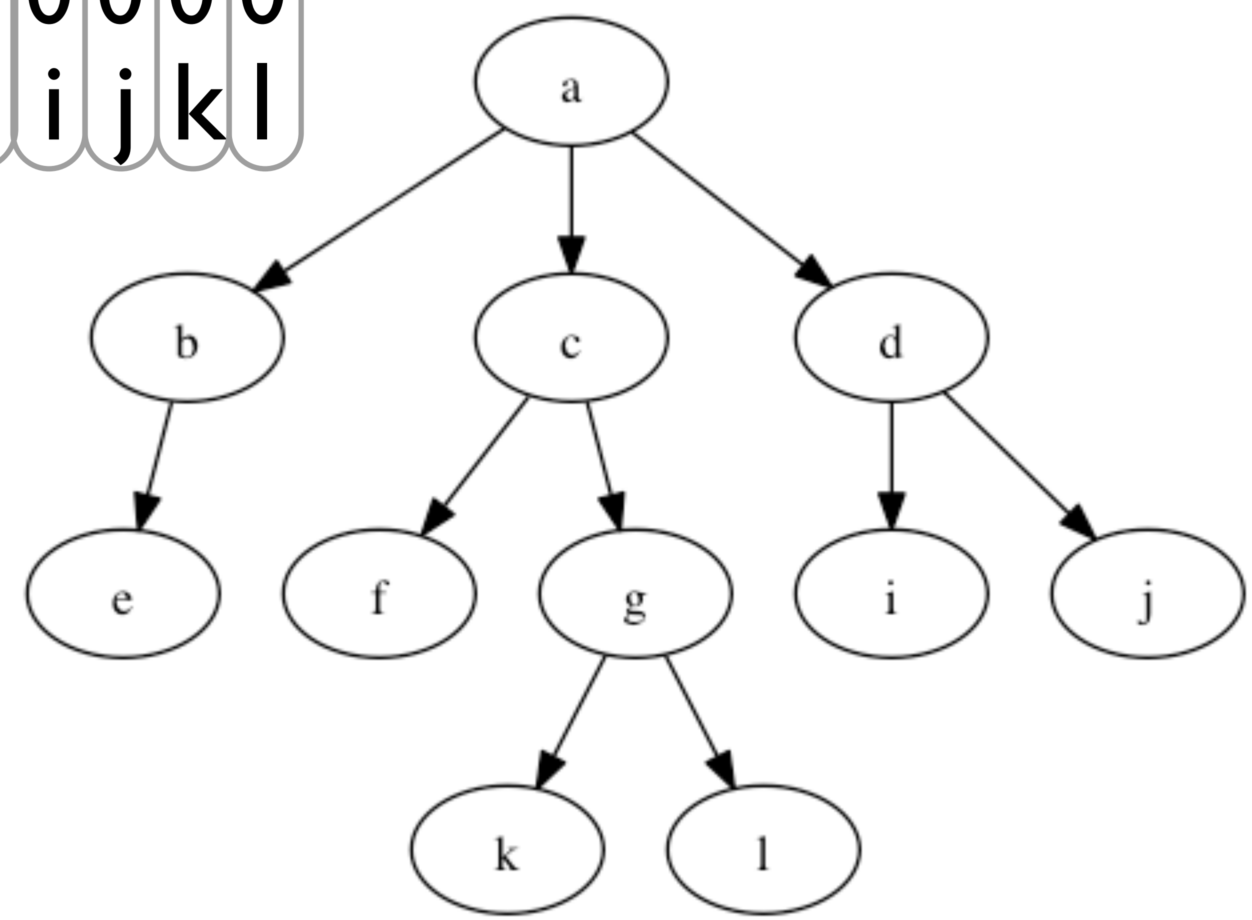
f is the 6th node

use select(0, 6)

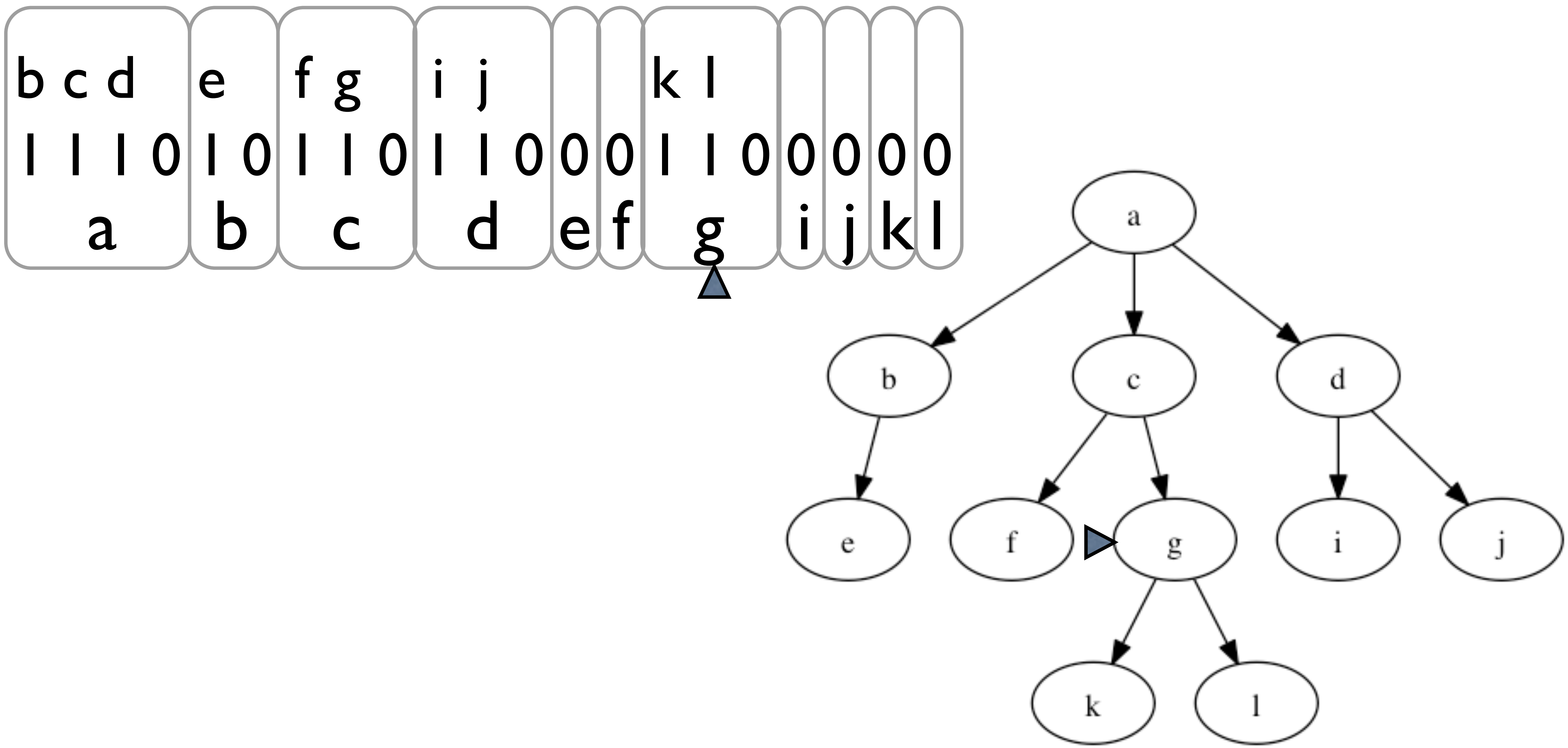


LOUDS (parent)

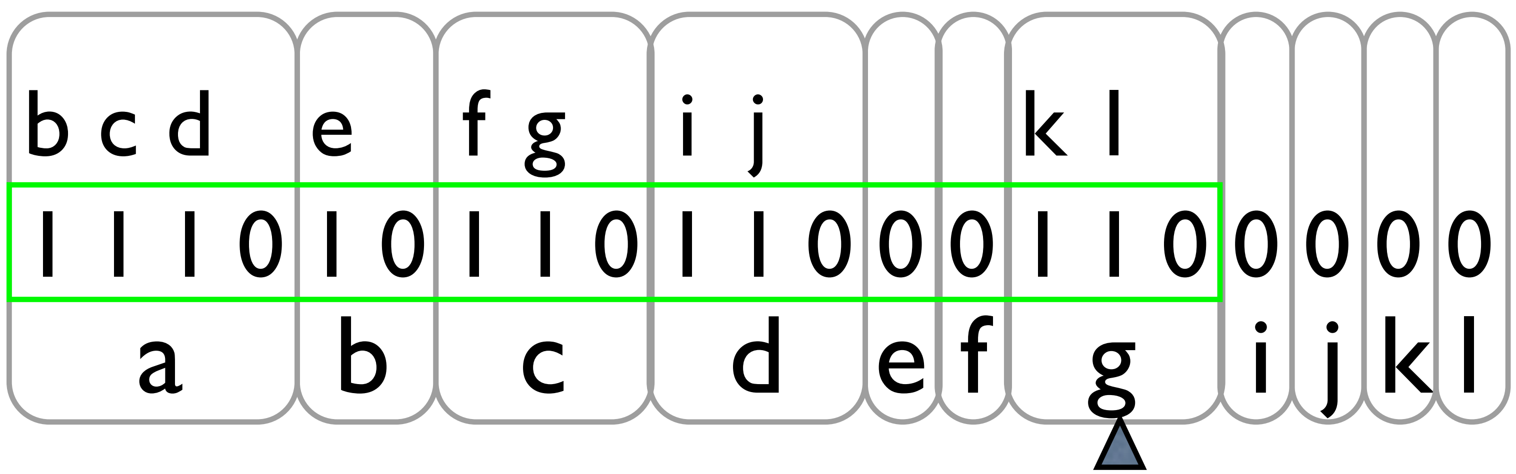
b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l

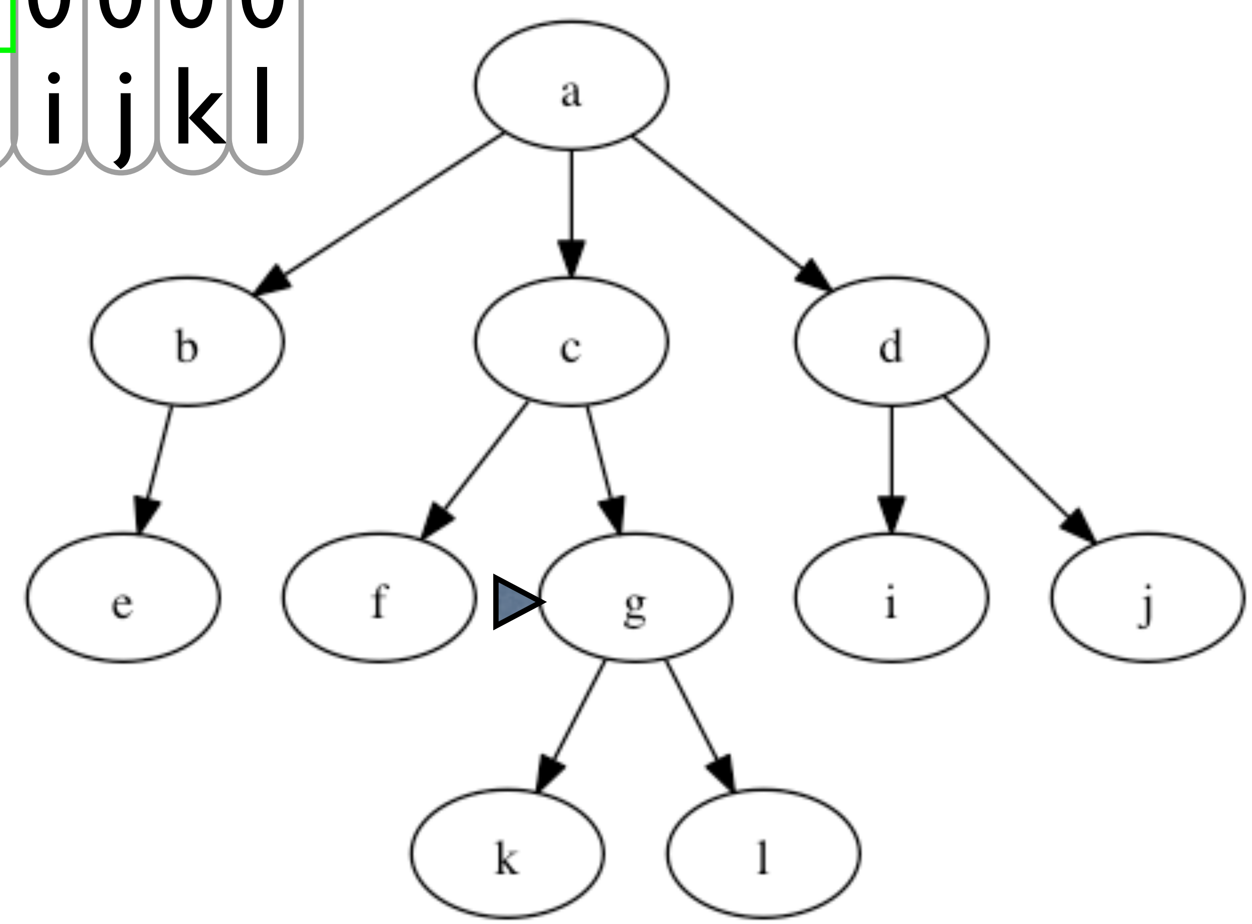


LOUDS (parent)

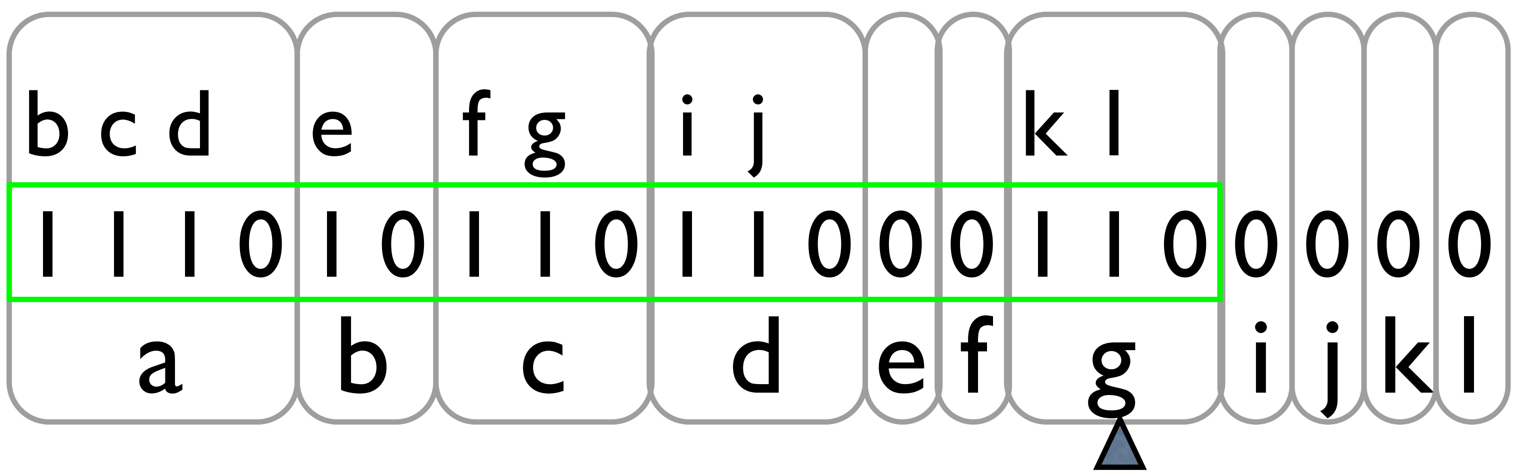


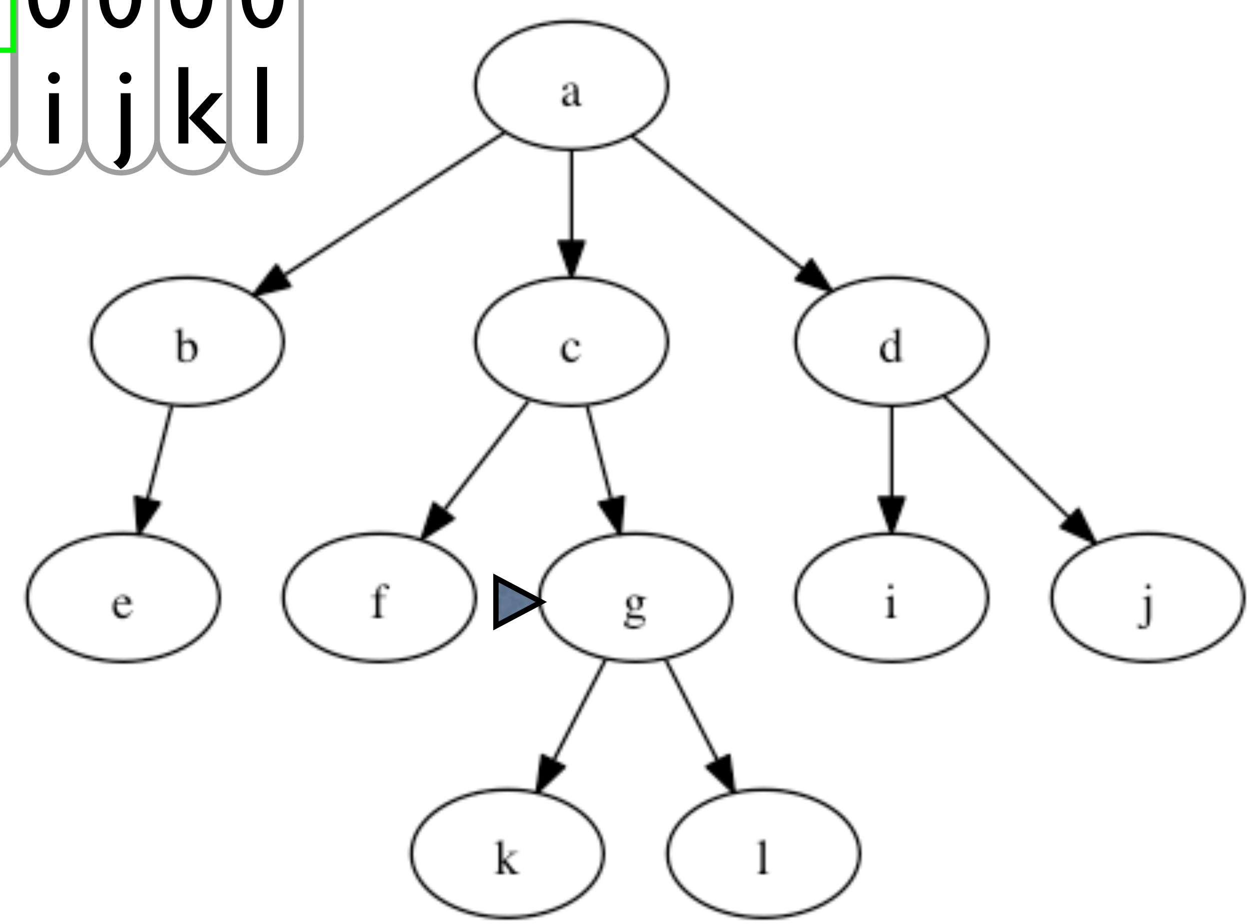
LOUDS (parent)





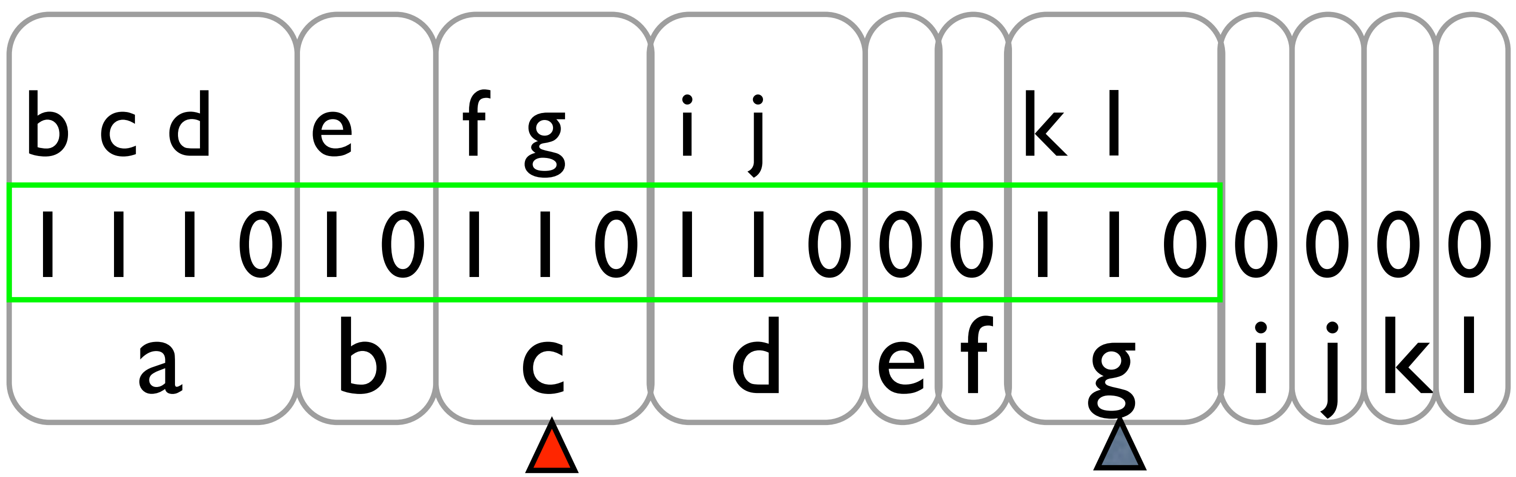
LOUDS (parent)

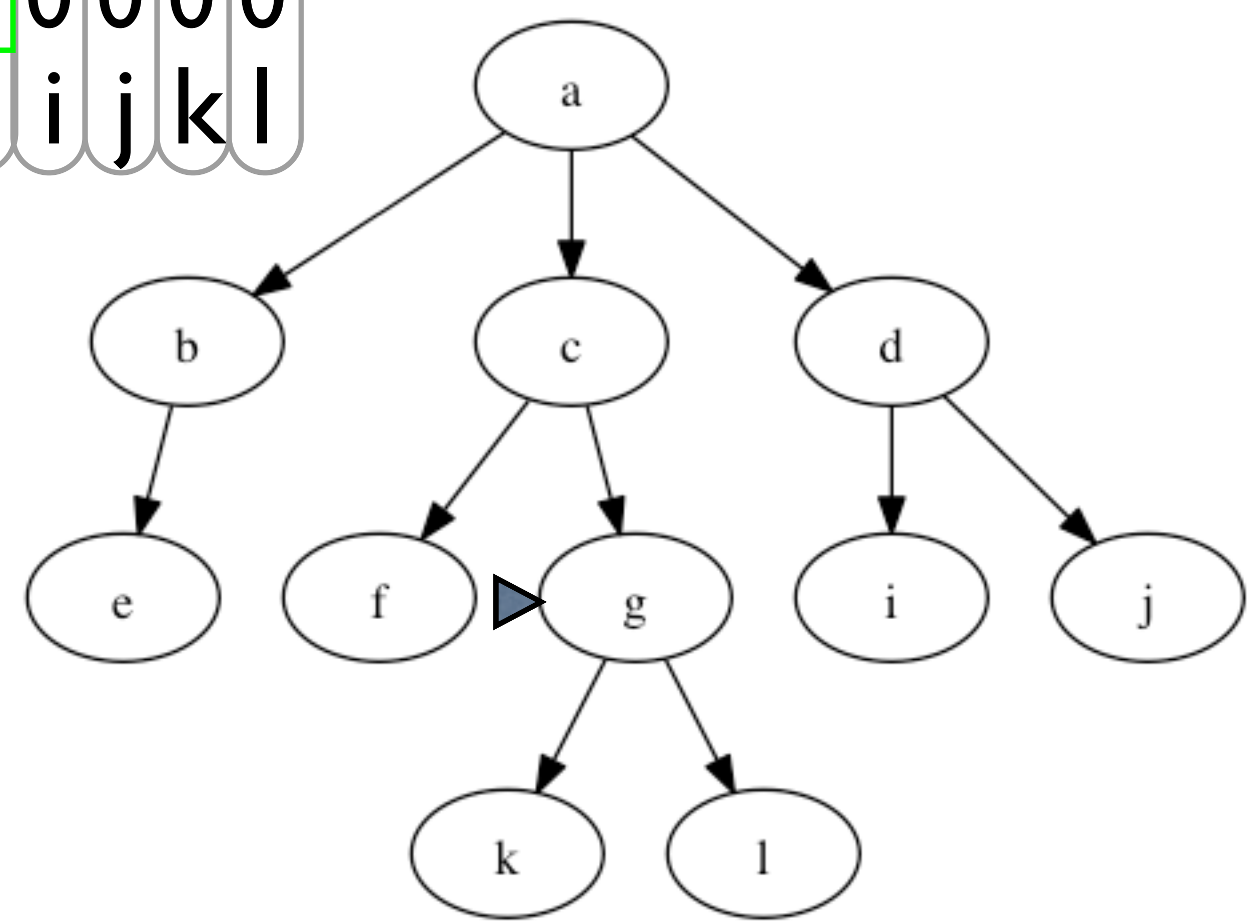




g is the 7th node

LOUDS (parent)



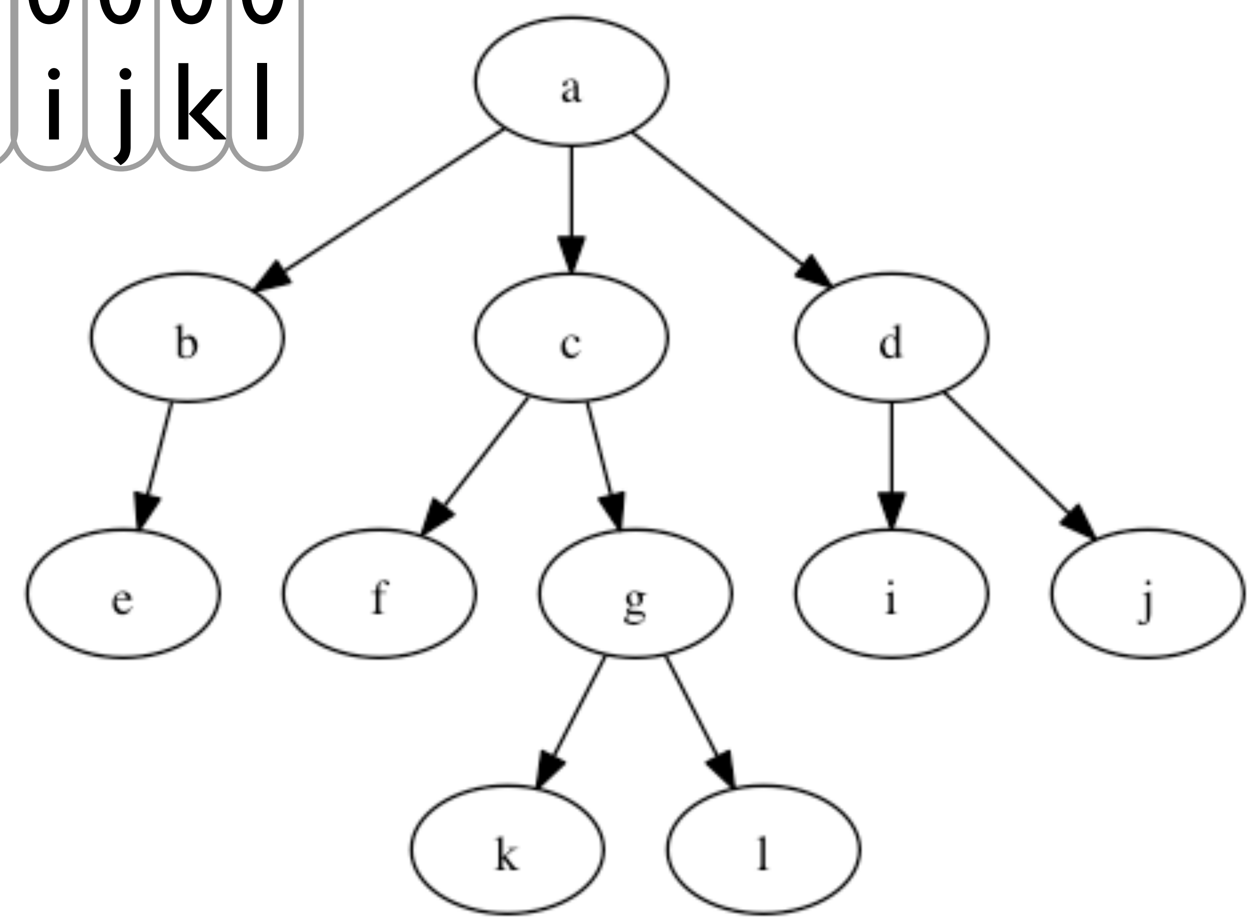


g is the 7th node

use select(1, 7 - 1)

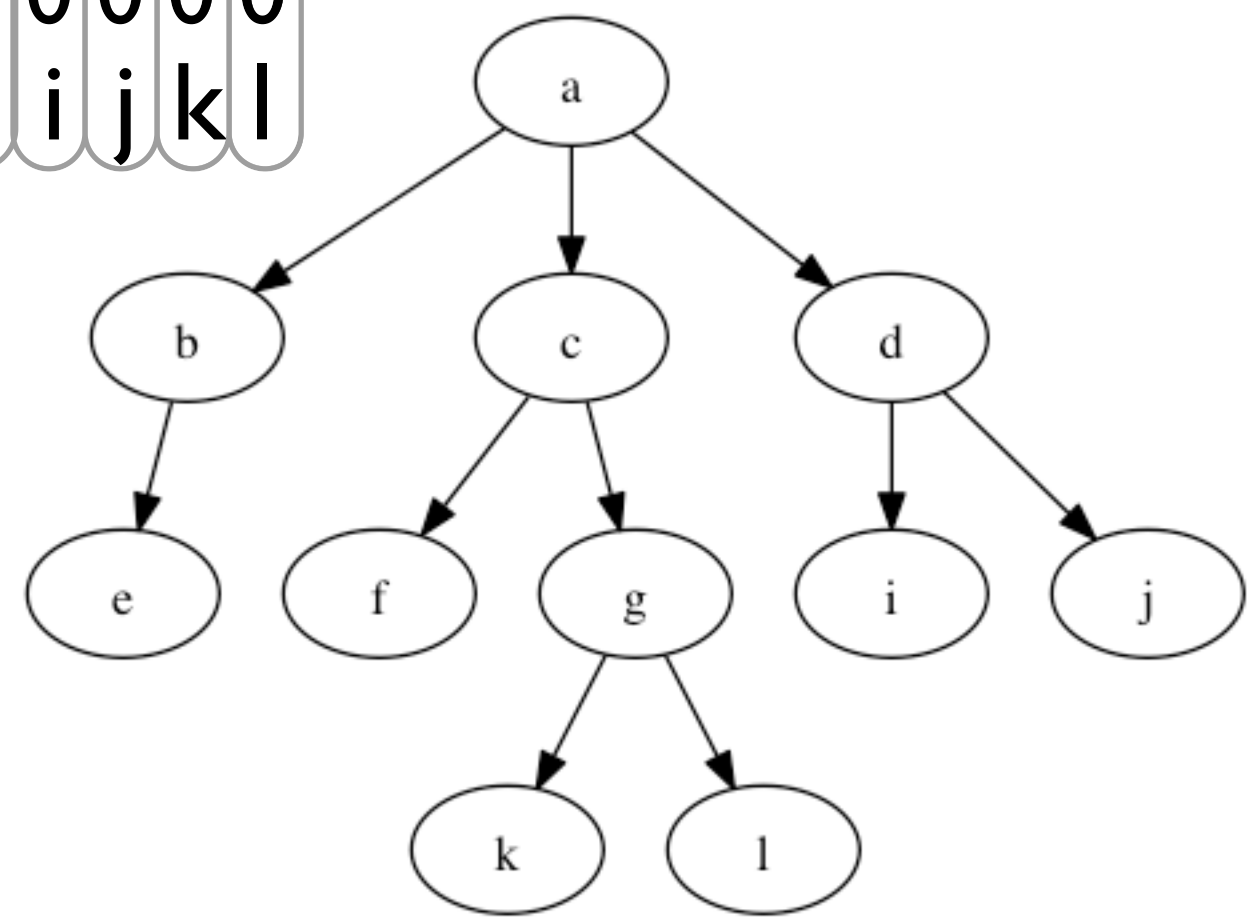
LOUDS (others)

b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l



LOUDS (others)

b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l



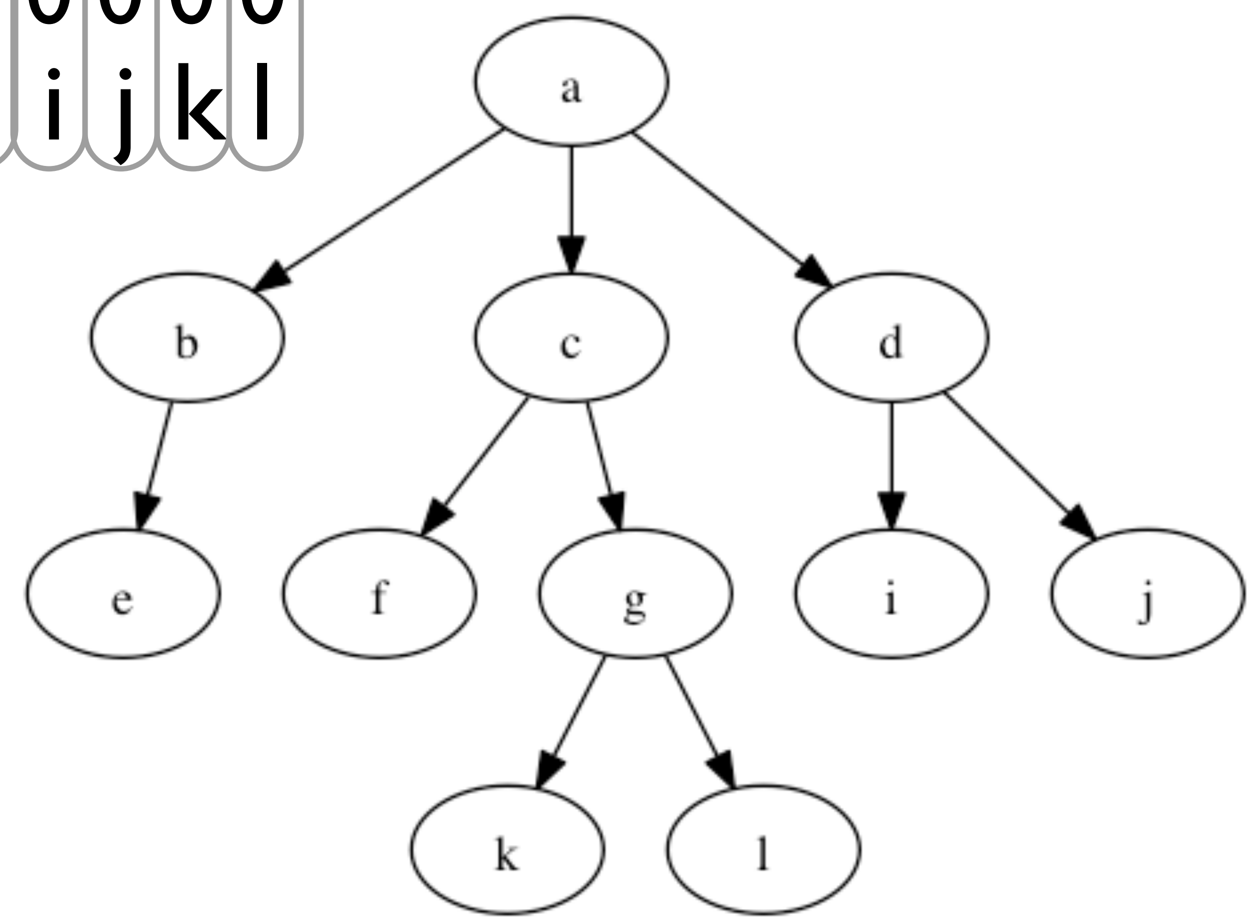
Left/right sibling

LOUDS (others)

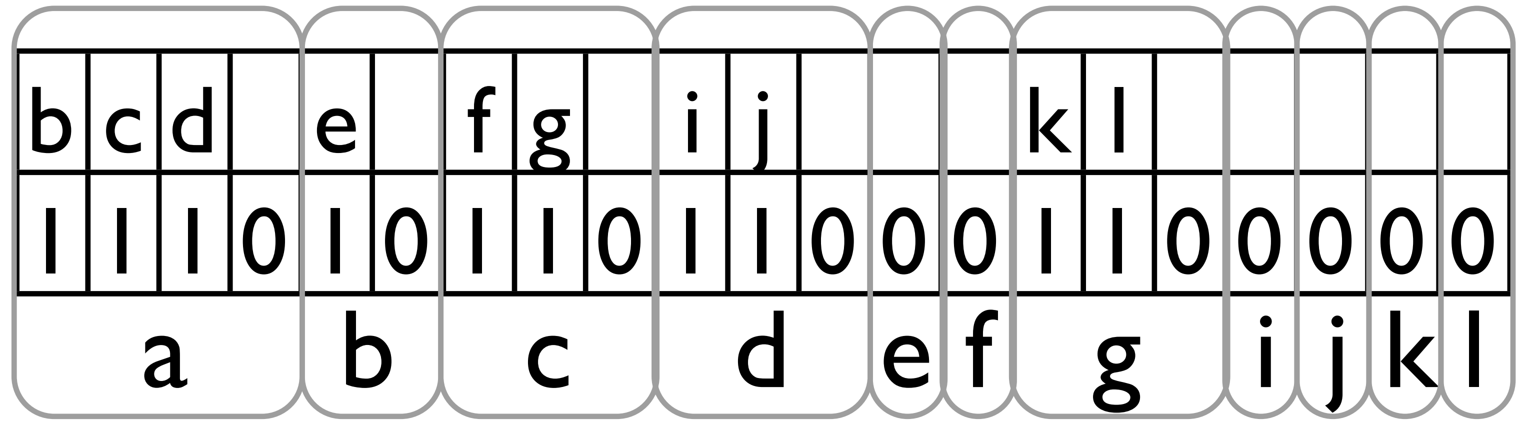
b c d	e	f g	i j			k l				
1 1 1 0	1 0	1 1 0	1 1 0	0	0	1 1 0	0	0	0	0
a	b	c	d	e	f	g	i	j	k	l

Left/right sibling

Degree



LOUDS (others)



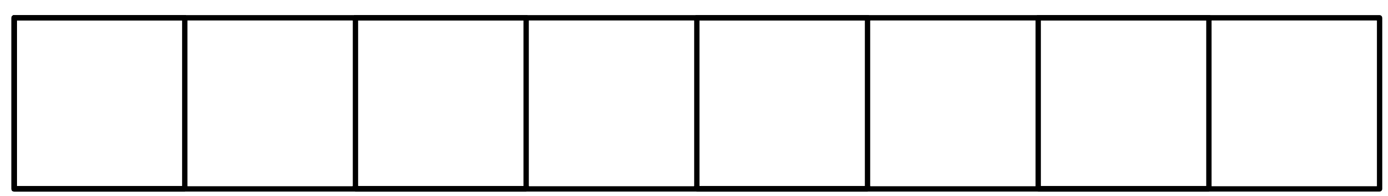

```
Tree *createTree() {
  vector<cds_word> v(21);
  v[0] = v[1] = v[2] = v[4] = v[6] = v[7] = 1;
  v[9] = v[10] = v[14] = v[15] = 1;
  Array *a = Array::Create(v);
  return new TreeLouds(new BitSequenceOneLevelRank(a, 20));
}

...
Tree * t = createTree();
assert(t->Parent(12) == 5);
assert(t->Child(5, 0) == 12);
assert(t->Degree(16) == 2);
... t->NextSibling(12)
... t->PrevSibling(12)
...
```

Trees

- BP: Geary, Raman, Raman & Rao
- DFUDS: Benoit et al. '05
- FF: Sadakane & Navarro '10
- Partitioning: Farzan & Munro '09

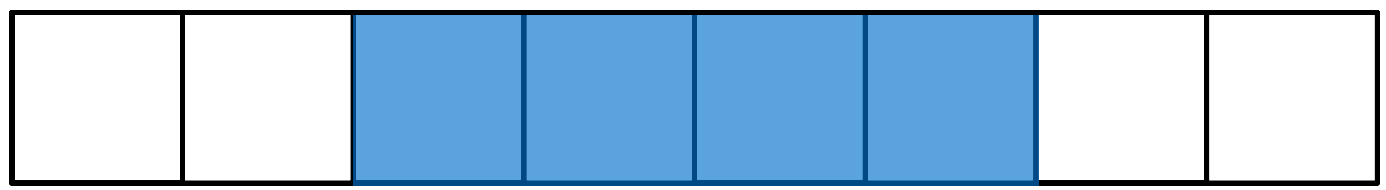
Trees

- There are some amazing results related to trees



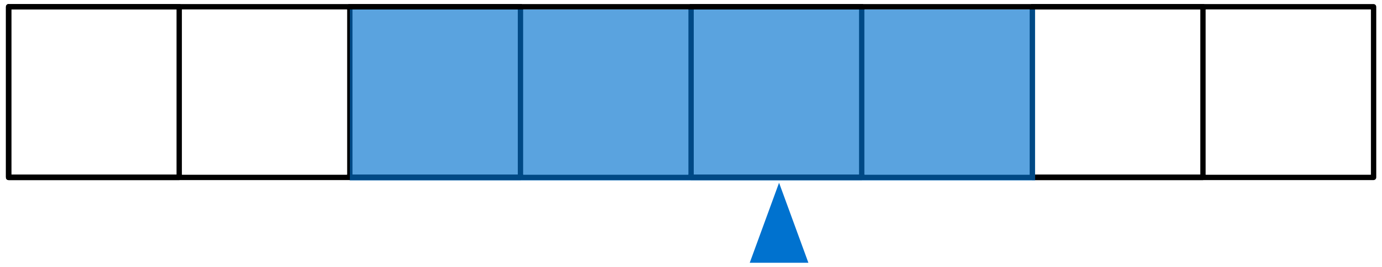
Trees

- There are some amazing results related to trees



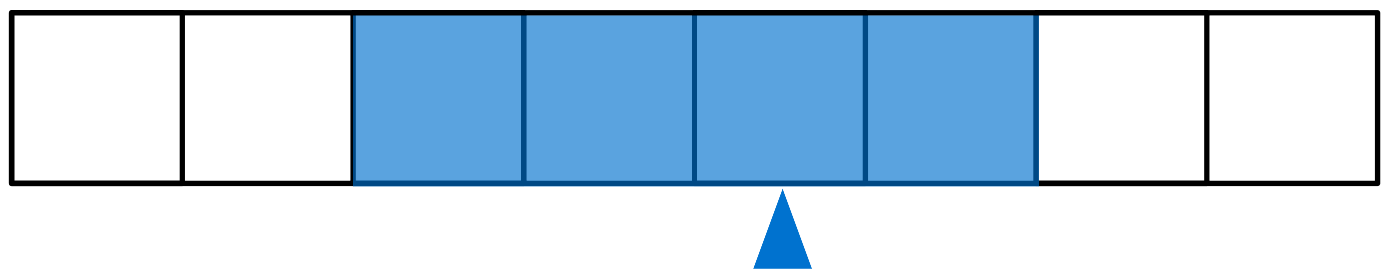
Trees

- There are some amazing results related to trees



Trees

- There are some amazing results related to trees



$2n + o(n)$ bits and constant time!

Sequences

Sequences

- rank(a, i): counts how many times does a occur up to position i
- select(a, j): finds the j-th occurrence of a
- access(i): retrieves the i-th element

Sequences

- rank(a, i): counts how many times does a occur up to position i
- select(a, j): finds the j-th occurrence of a
- access(i): retrieves the i-th element

alabaralalabarda

rank(r, 11) = 1

Wavelet Trees

- The best known structure for solving rank, select, and access on sequences
- Has many other applications we will not have time to cover
- Wavelet trees are awesome!

Wavelet Trees

access(8)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

access(8)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

access(8)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

access(8)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	*H*	*D*	*H*	*A*	*C*	*E*	*E*	*G*	*B*	*C*	*B*	*G*	*C*	*F*
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	*A*	*C*	*B*	*C*	*B*	*C*
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	*H*	*H*	*E*	*E*	*G*	*G*	*F*
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	*B*	*B*
A/B	0	1	1

C/D	3	6	11	14
	D	*C*	*C*	*C*
	1	0	0	0

	1	7	8	15
	E	*E*	*E*	*F*
E/F	0	0	0	1

	2	4	9	13
	H	*H*	*G*	*G*
G/H	1	1	0	0

Wavelet Trees

rank('E', 11)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

rank('E', 11)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	*H*	*D*	*H*	*A*	*C*	*E*	*E*	*G*	*B*	*C*	*B*	*G*	*C*	*F*
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	*A*	*C*	*B*	*C*	*B*	*C*
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	*H*	*H*	*E*	*E*	*G*	*G*	*F*
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	*B*	*B*
A/B	0	1	1

C/D	3	6	11	14
	D	*C*	*C*	*C*
	1	0	0	0

	1	7	8	15
	E	*E*	*E*	*F*
E/F	0	0	0	1

	2	4	9	13
	H	*H*	*G*	*G*
G/H	1	1	0	0

Wavelet Trees

rank('E', 11)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

rank('E', 11)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

select('E', 3)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

select('E', 3)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

select('E', 3)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

select('E', 3)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

- Each level has n bits
- There are $\lceil \log \sigma \rceil$ levels
- All queries cost $O(\log \sigma)$ time
- The total space is $n \log \sigma + o(n \log \sigma)$ bits

Wavelet Trees

- We can also compress the sequence, changing the shape of the tree.
- Any encoding works.
- With Huffman shape we get

$$n(H_0(S)+1)+o(n(H_0+1))$$

Wavelet Trees

- What about compressing the bitmaps?
- If we use RRR, we also get close to H_0
- It works really well on sequences with runs, like the BWT.

Wavelet Trees

- One problem: the tree
- We are spending $O(\sigma \log \sigma)$ bits on pointers!

Wavelet Trees

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees

- We now need $n \log \sigma(1 + o(1))$ bits
- We don't waste $O(\sigma \log \sigma)$ bits in pointers
- To move from one level to another we perform 2 rank queries

Wavelet Trees

```
Array *array = CreateRandomSequence(len, sigma);
WaveletTreeNoPtrs *seq = new WaveletTreeNoPtrs(array,
                                  new BitSequenceBuilderOneLevelRank(20),
                                  new MapperNone());
seq->Rank(65, 10);
seq->Select('A', 5);
seq->Access(10);
...
```

Wavelet Matrix

- Can we reduce the number of operations when we move from one level to the other?

Wavelet Matrix

- Can we reduce the number of operations when we move from one level to the other?

YES! Go to our talk on Wednesday :-)

Wavelet Trees

- Can we give Huffman shape to the wavelet tree without pointers?

Wavelet Trees

- Can we give Huffman shape to the wavelet tree without pointers?

YES! Go to Alberto's talk tomorrow :-)

Permutations

Representing a permutation of size n requires $n \log n$ bits

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

Permutations

Representing a permutation of size n requires $n \log n$ bits

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

What if we want to compute $\pi^{-1}(i)$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

$\pi(i)$ is easy

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

$\pi(i)$ is easy

$\pi^{-1}(i)$ is also easy, if we spend $n \log n$ extra bits

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

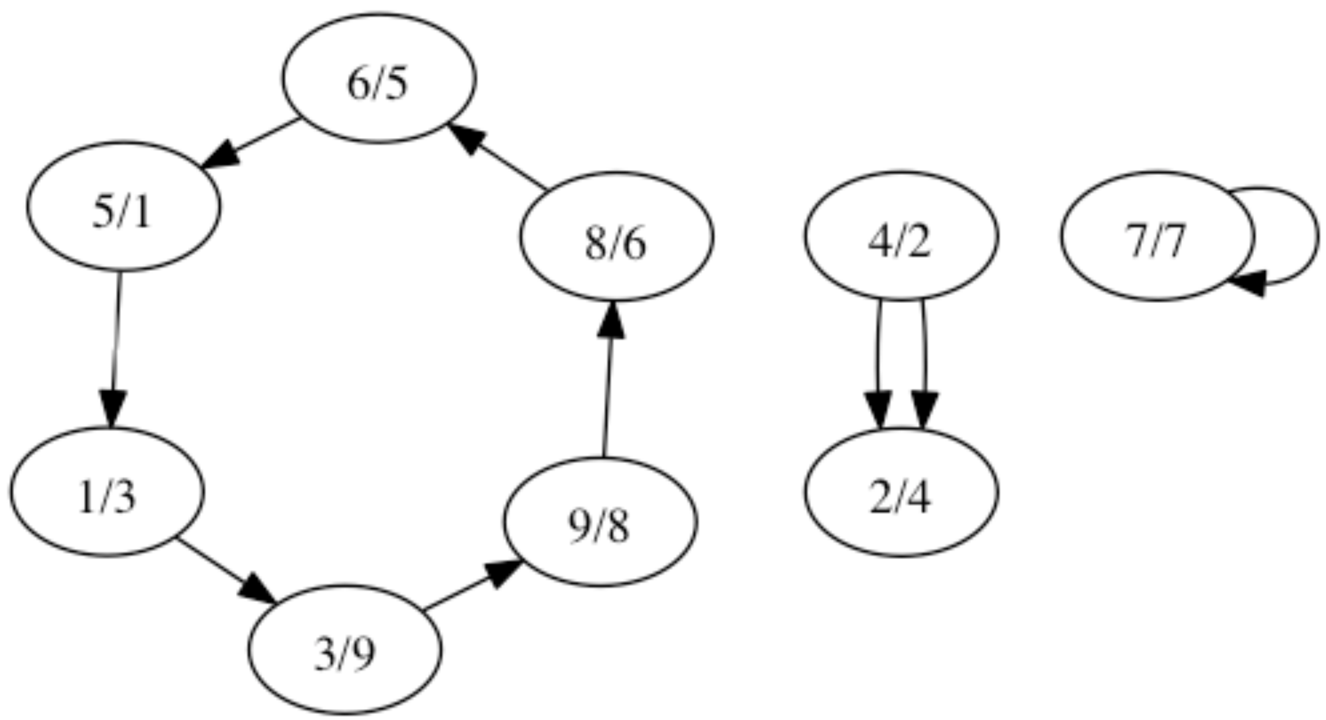
$\pi(i)$ is easy

$\pi^{-1}(i)$ is also easy, if we spend $n \log n$ extra bits

Wavelet trees solve this in $O(\log n)$

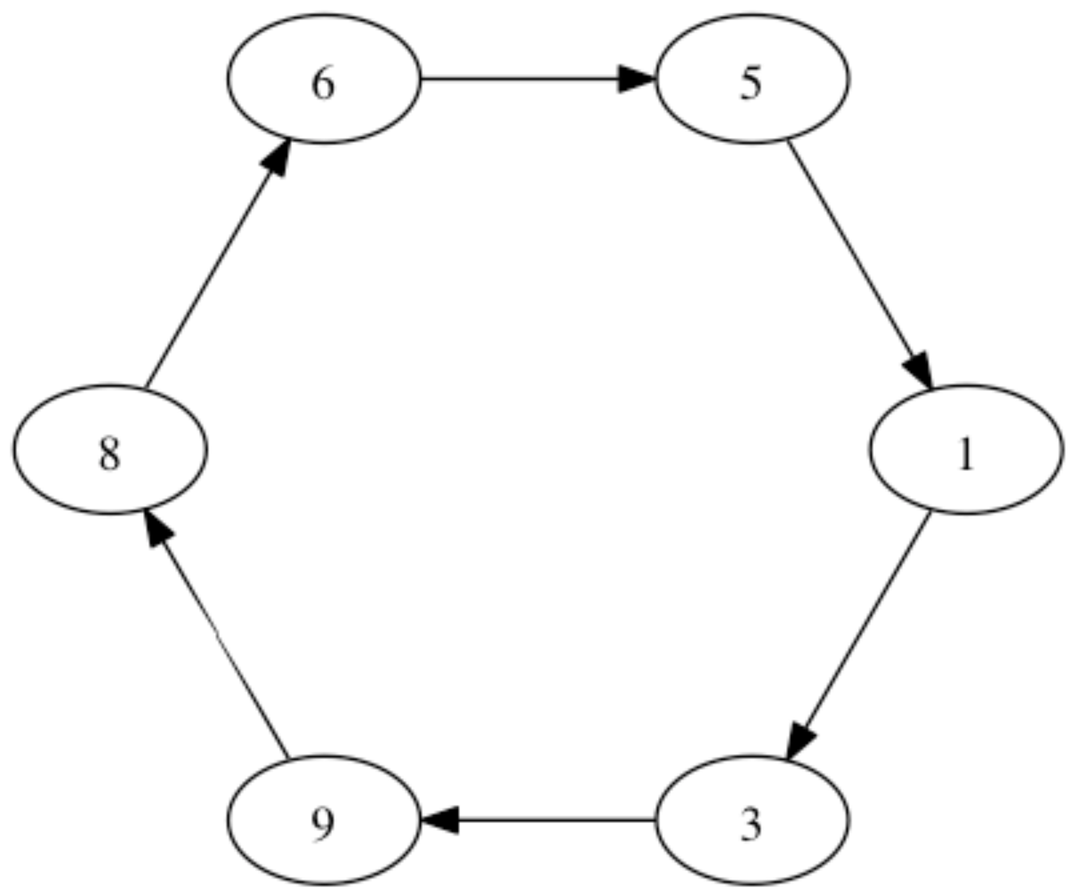
Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$



Permutations

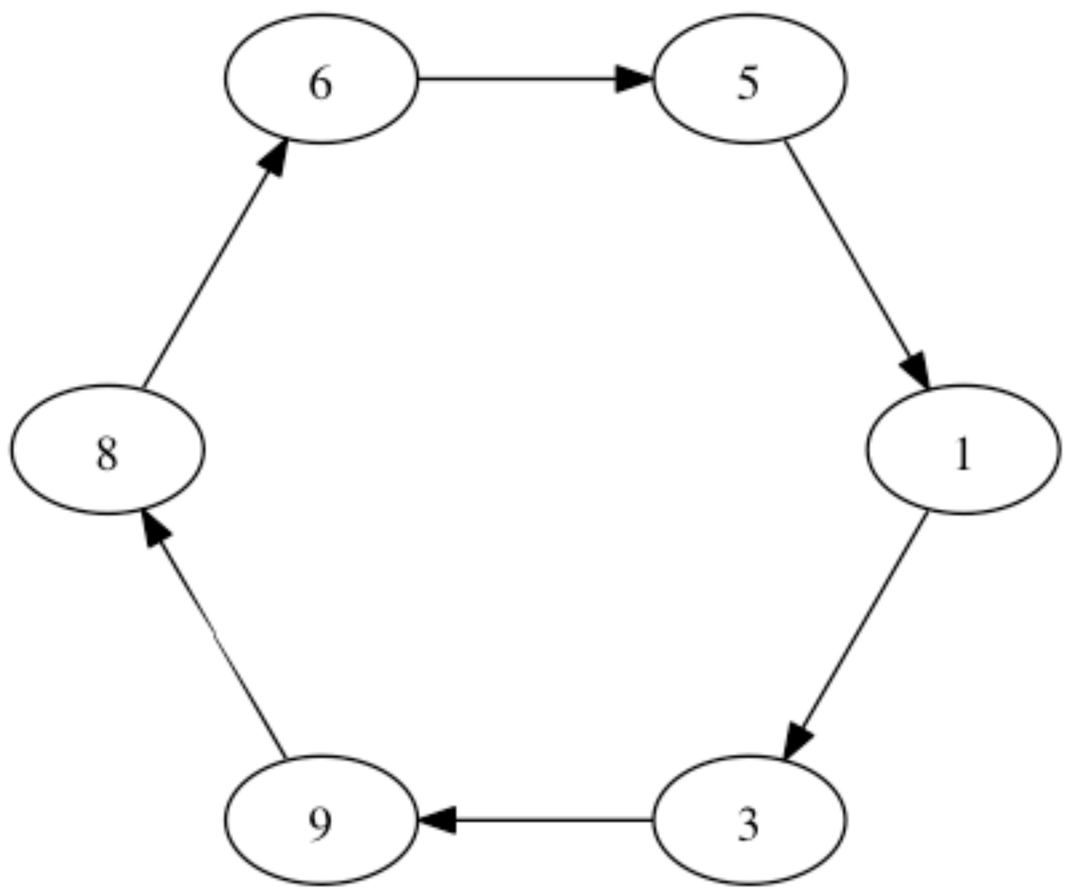
$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

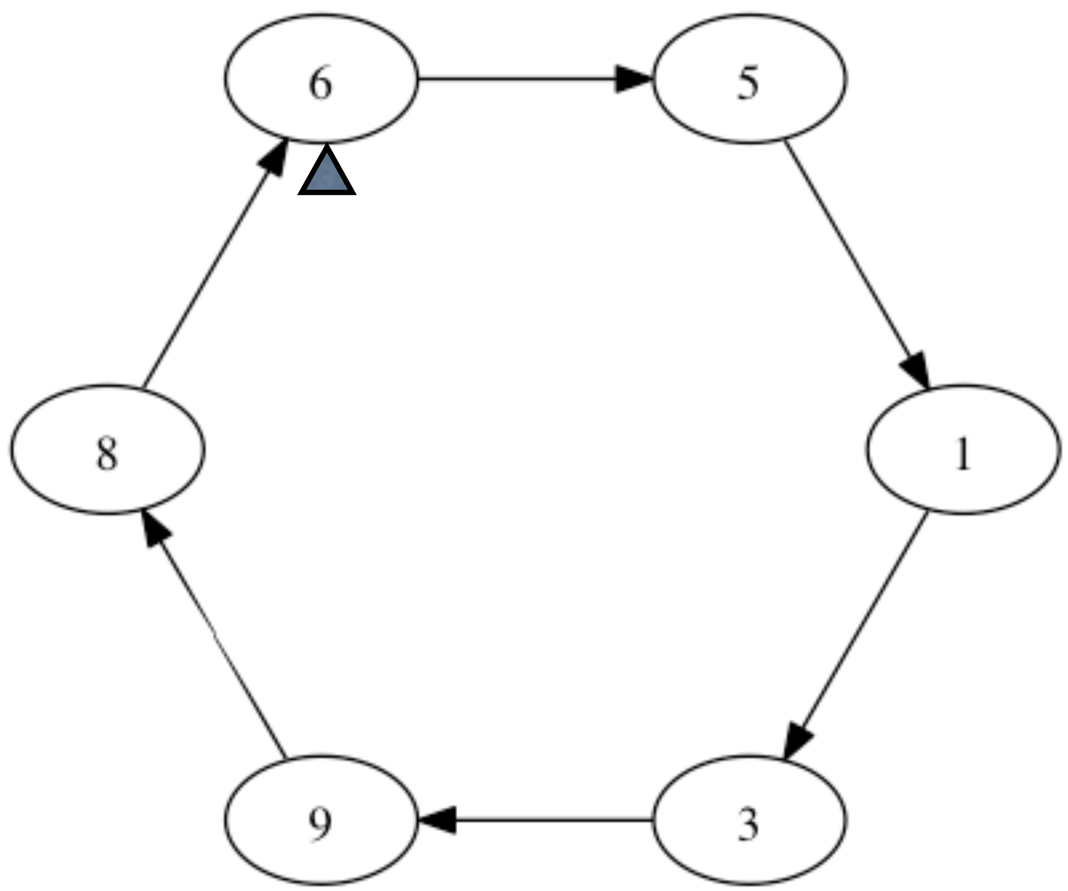
$\pi^{-1}(8)$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

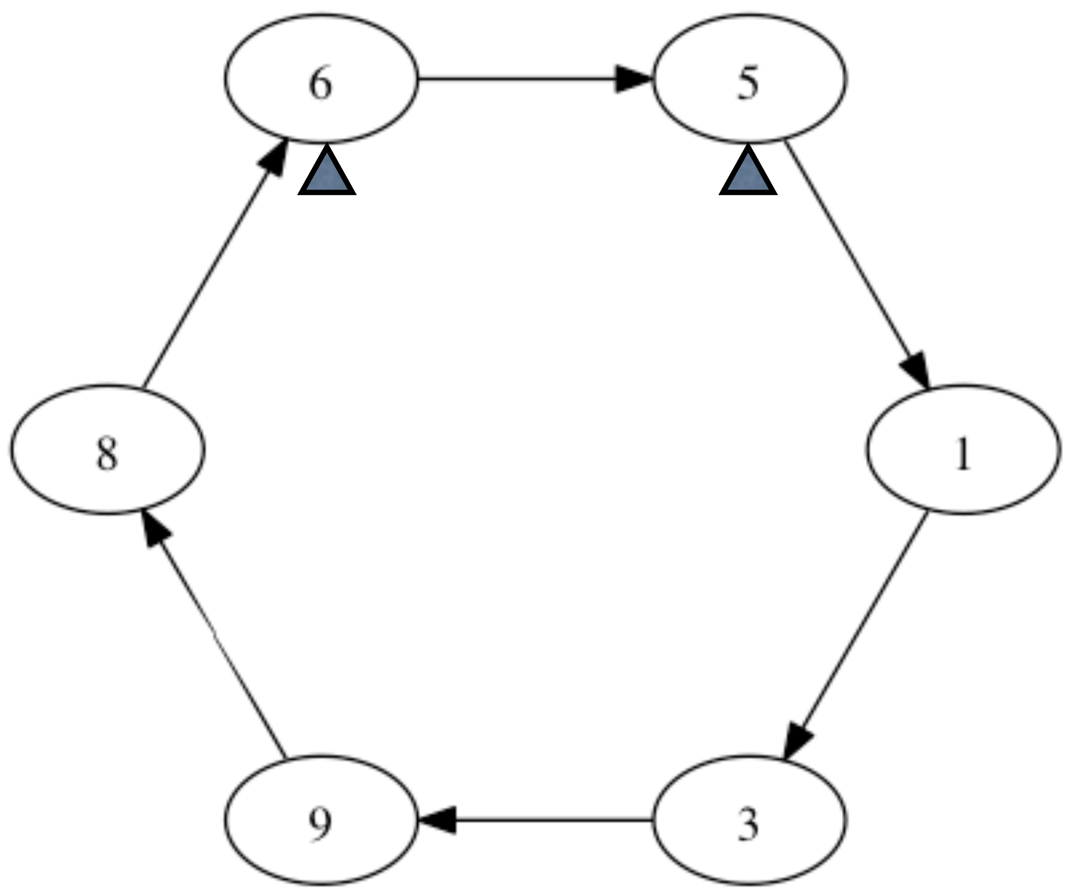
$\pi^{-1}(8)$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

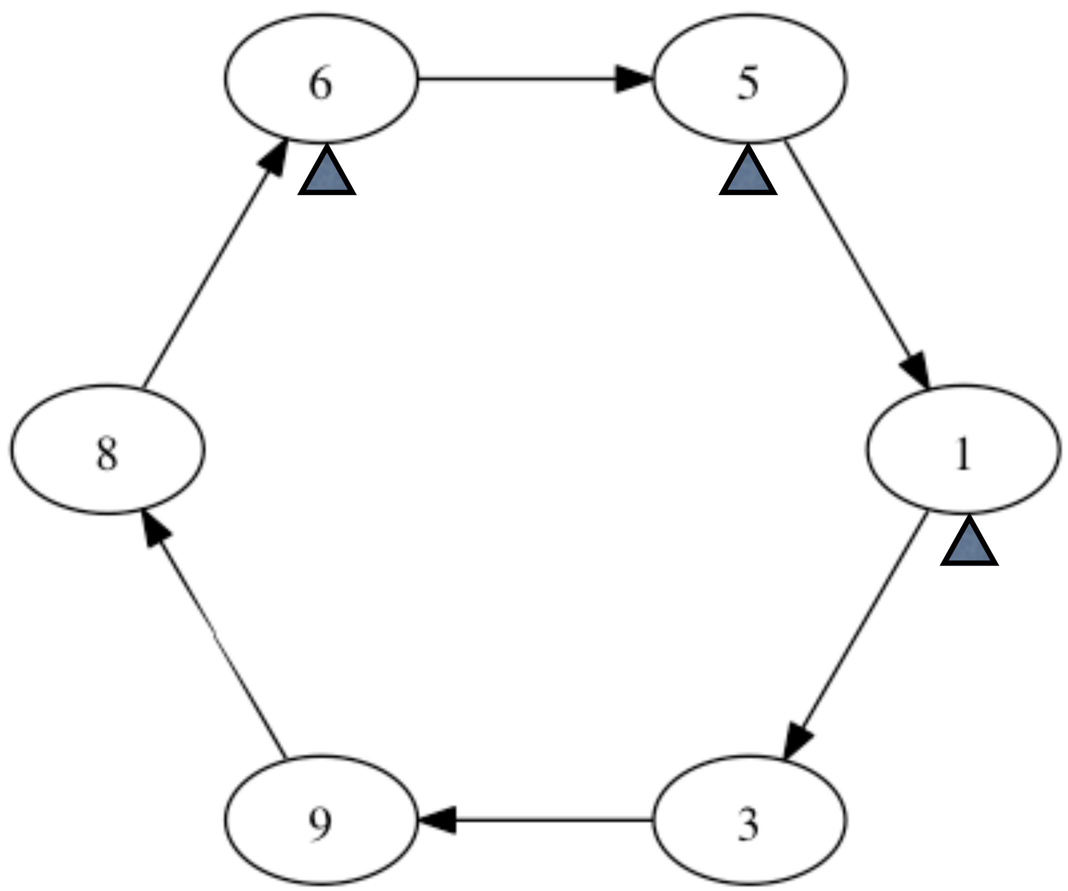
$\pi^{-1}(8)$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

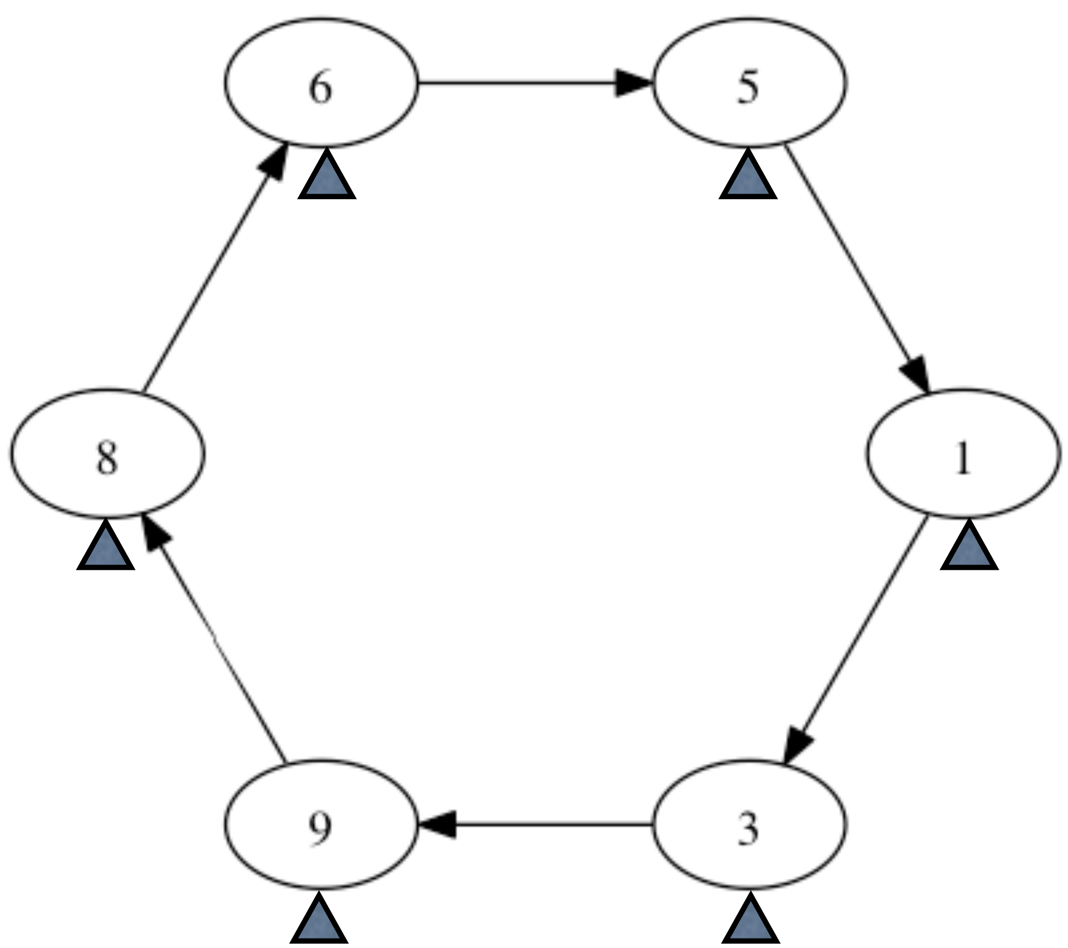
$\pi^{-1}(8)$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

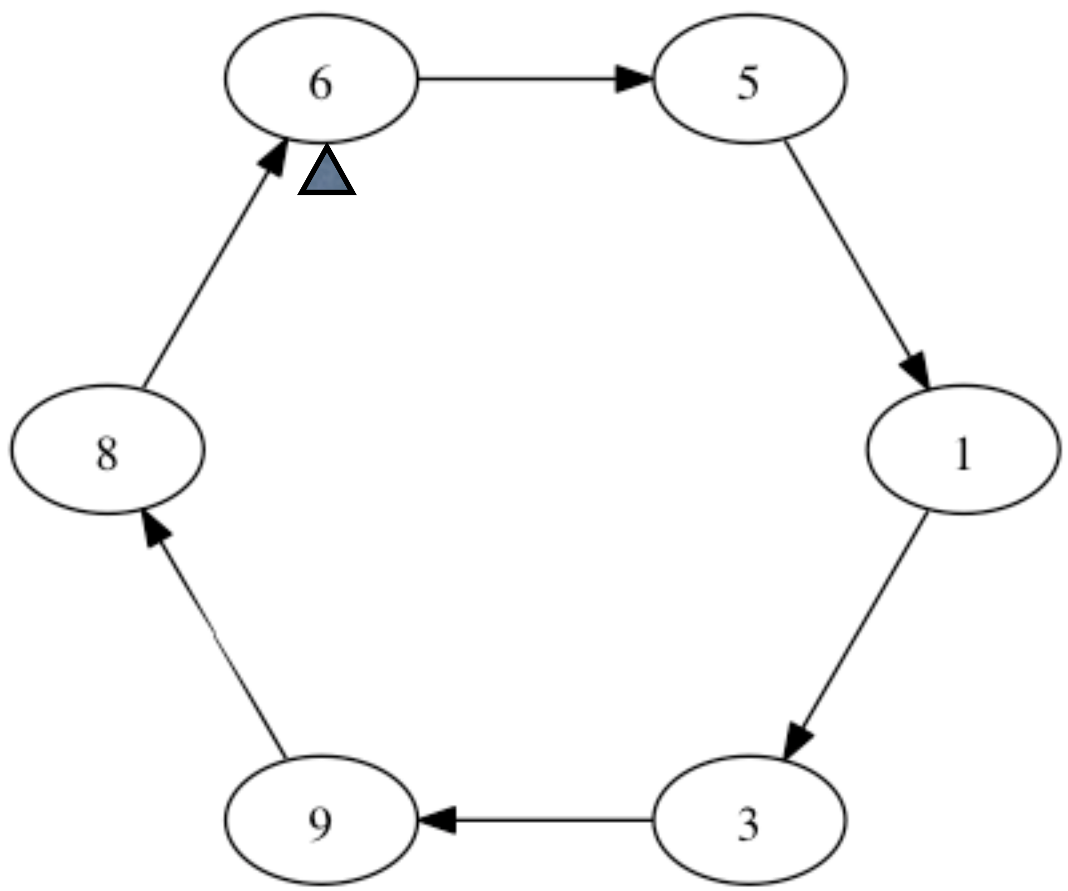
$\pi^{-1}(8)$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

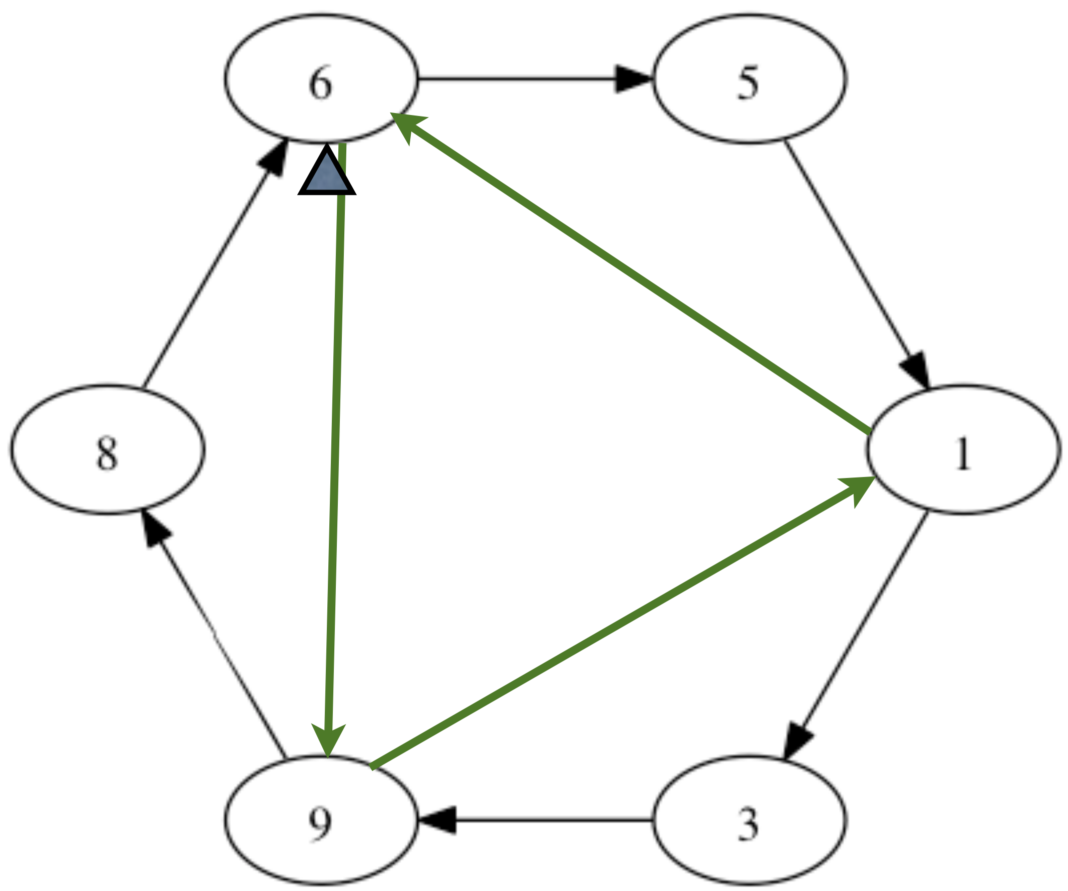
$\pi^{-1}(8)$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

$\pi^{-1}(8)$



Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

0 0 1 0 1 0 0 1 0

$$P = [5, 8, 3]$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

0 0 1 0 1 0 0 1 0

$$P = [5, 8, 3]$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

0 0 1 0 1 0 0 1 0

$$P = [5, 8, 3]$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

0 0 1 0 1 0 0 1 0

$$P = [5, 8, 3]$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8] \qquad n \log n + O(n)$$

0 0 1 0 1 0 0 1 0

$$P = [5, 8, 3]$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8] \qquad n \log n + O(n)$$

0 0 1 0 1 0 0 1 0 $\qquad n + o(n)$

$$P = [5, 8, 3]$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8] \qquad n \log n + O(n)$$

0 0 1 0 1 0 0 1 0 $\qquad n + o(n)$

$$P = [5, 8, 3] \qquad \leq n/t \log n + O(n)$$

Permutations

$$\pi = [3, 4, 9, 2, 1, 5, 7, 6, 8]$$

$$0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0$$

$$P = [5, 8, 3]$$

$$n \log n + O(n)$$

$$n + o(n)$$

$$\leq n/t \log n + O(n)$$

$$n \log n + \frac{n \log n}{t} + O(n)$$

Permutations

- We can trade space for time in computing π^{-1}
- For $t = \log\log n$ we get $n \log n + o(n \log n)$ bits

Permutations

```
cds_word a[] = {1,2,3,4,5,6,7,8,9,0};
Array *perm_a = Array::Create((cds_word*)a, 0, 9);
PermutationMRRR *perm = new PermutationMRRR(perm_a, 3);
for (cds_word i = 0; i < perm->GetLength(); i++) {
  cds_word expected = (i+1) % 10;
  ASSERT_EQ(expected, perm->Access(i));
  expected = (i + 10 -1) % 10;
  ASSERT_EQ(expected, perm->Reverse(i));
}
```

GMR

- Last time we didn't know how to represent permutations
- Lets focus on sequences of length $O(\sigma)$

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

access(5)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

access(5)

GMR

$$S = [abcabbca]$$

access(5)

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

$$\pi^{-1}(5)$$

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

$\pi^{-1}(5)$

access(5)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

$\pi^{-1}(5)$

access(5)=b

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

$\pi^{-1}(5)$

access(5)=b

$O(\log\log\sigma)$ time

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

select(b, 3)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

select(b, 3)

select(1, 2)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

select(b, 3)

select(1, 2)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

select(b, 3)

select(1, 2)

GMR

$$S = [abcabbbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

select(b, 3)

select(1, 2)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

select(1, 2)

select(b, 3) =6

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

select(b, 3) =6

select(1, 2)

$O(1)$ time

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

rank(b, 5)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

rank(b, 5)

select(1, 2)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

rank(b, 5)

select(1, 2)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 $\boxed{1\ 0\ 0}$ 1 0

a b c

rank(b, 5)

select(1, 2)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

rank(b, 5) =2

select(1, 2)

GMR

$$S = [abcabbca]$$

$$\Pi = [1, 4, 8, 2, 5, 6, 3, 7]$$

1 0 0 1 0 0 1 0

a b c

rank(b, 5) =2

select(1, 2)

$O(\log \log \sigma)$ time

GMR

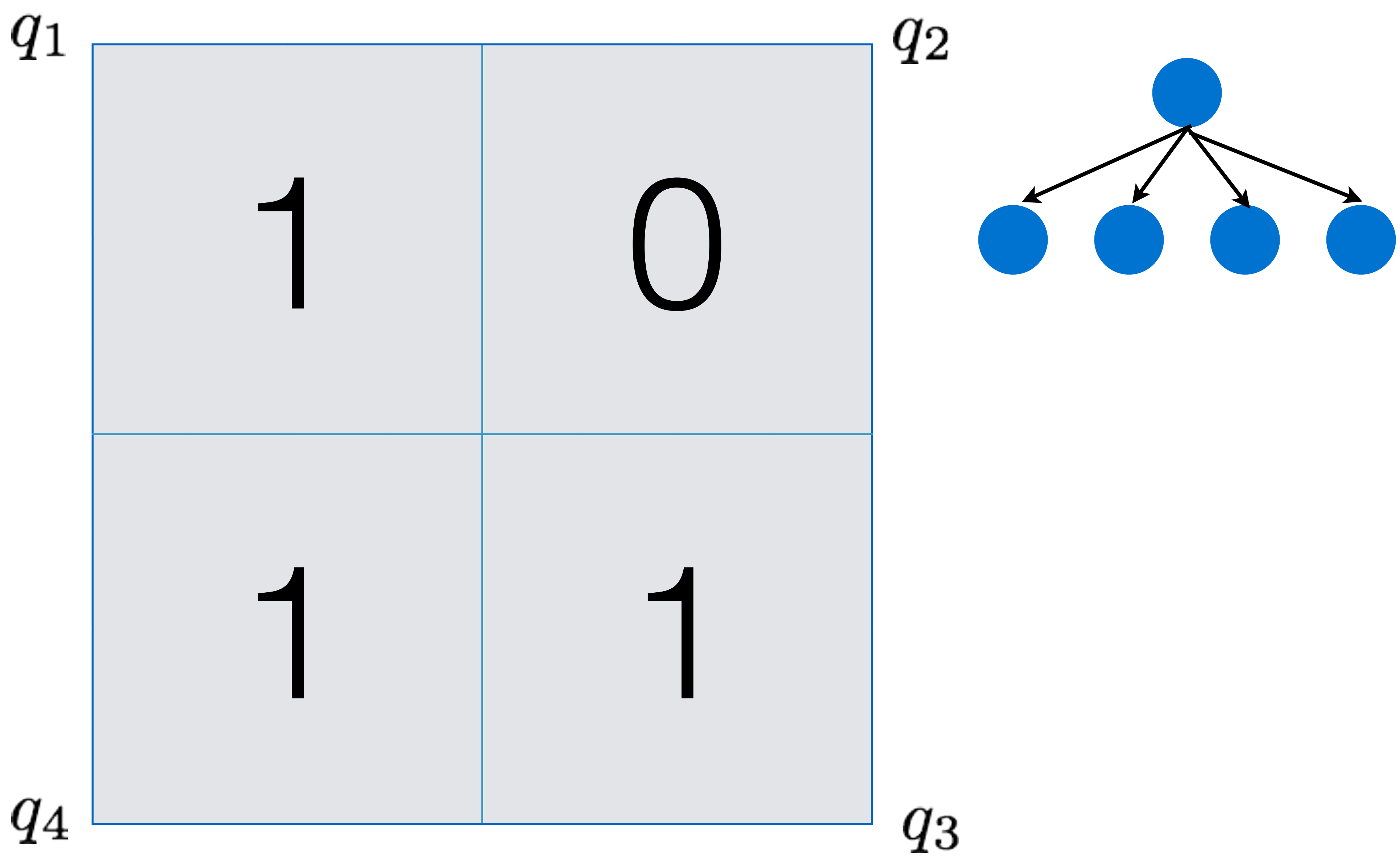
- It is possible to extend this to sequences of arbitrary length
- The resulting space is $n \log \sigma + n \cdot o(\log \sigma)$
- Times remain the same as for short sequences

Applications

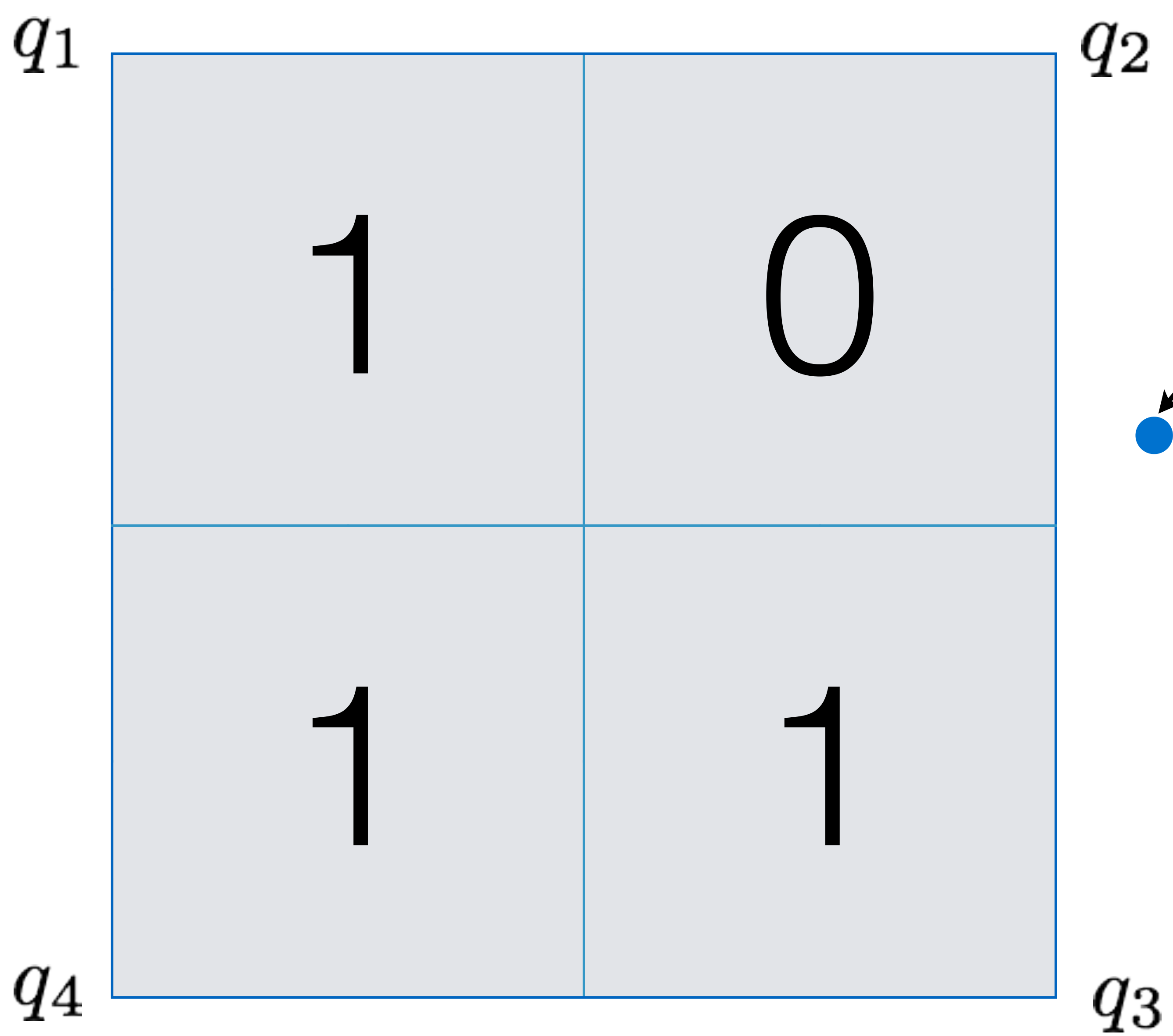
k2-tree

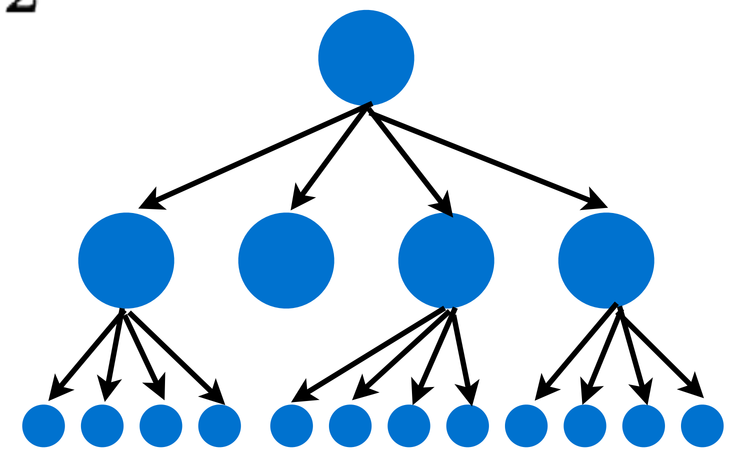
- A space-efficient representation for graphs
- The main idea behind it is to reduce the graph to a tree
- We will discuss a simple version, assuming we represent the graph as a quad-tree

k2-tree

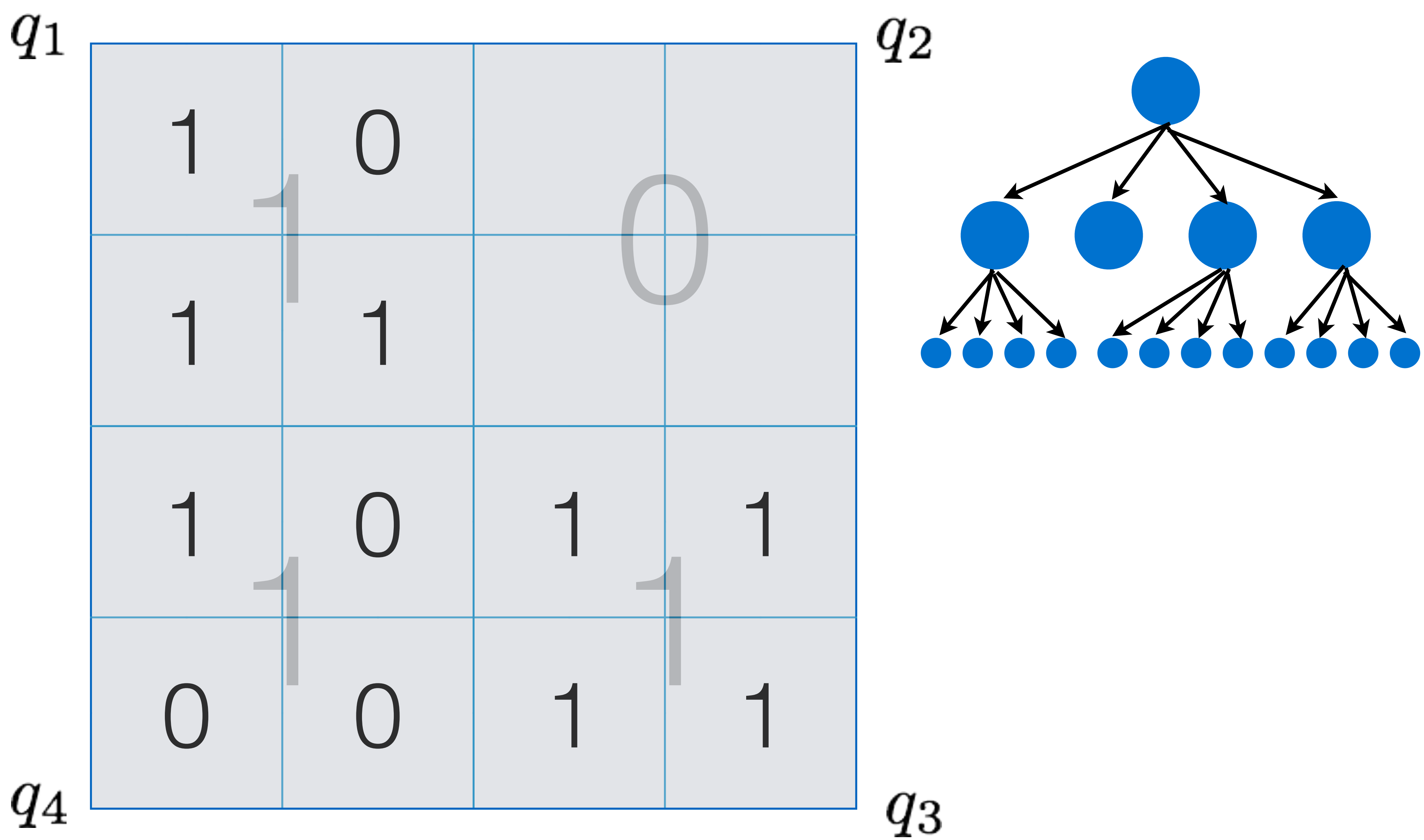


k2-tree

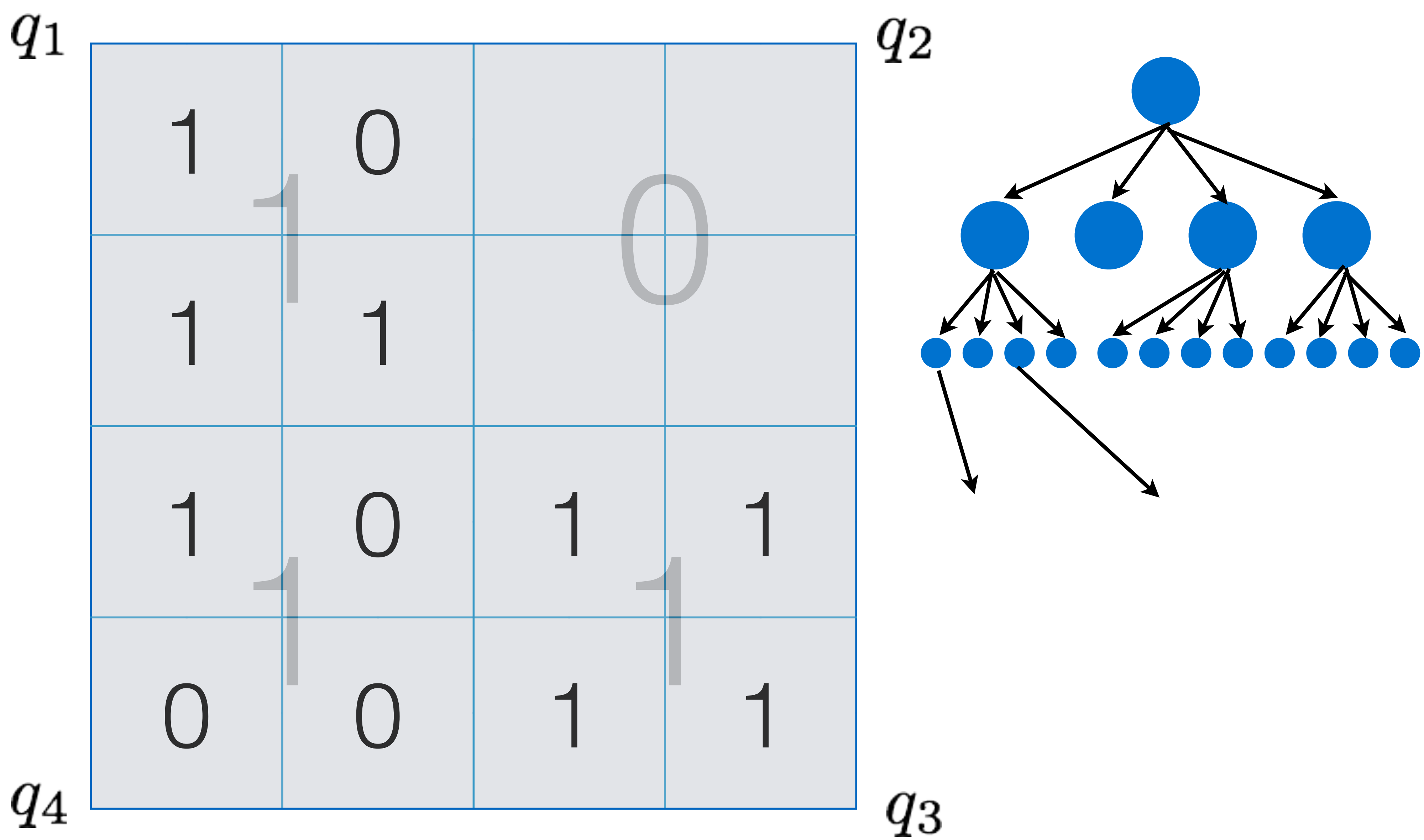




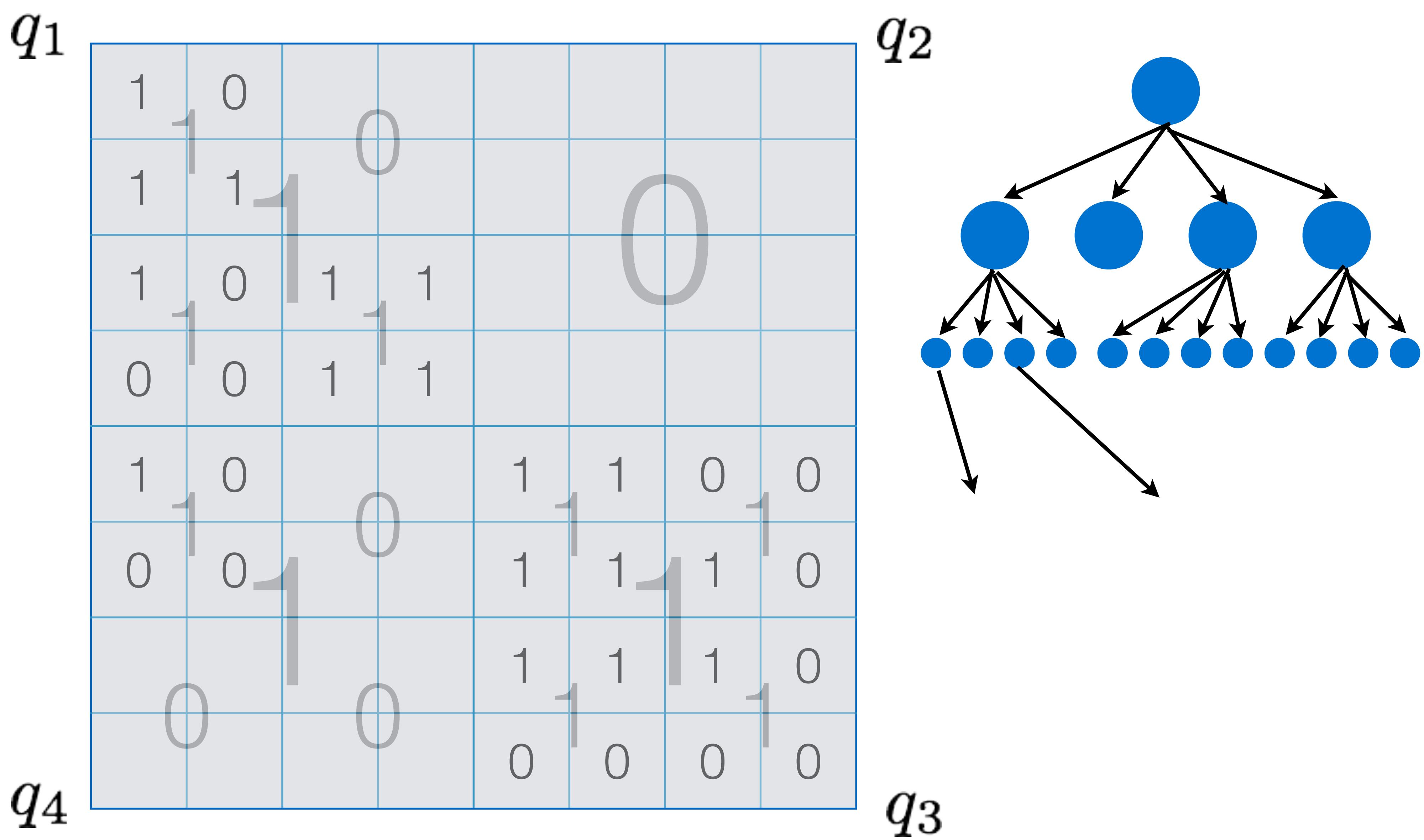
k2-tree



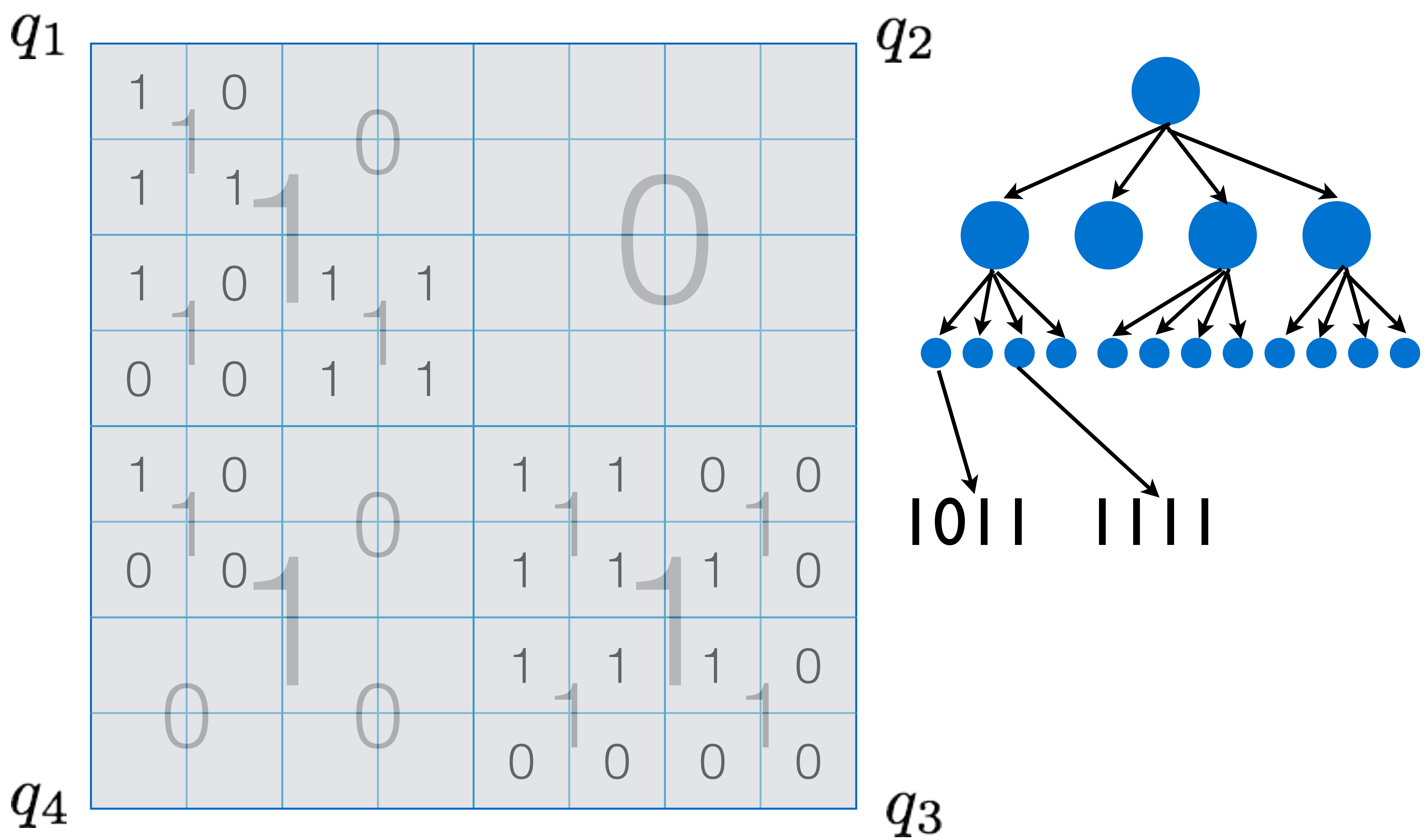
k2-tree



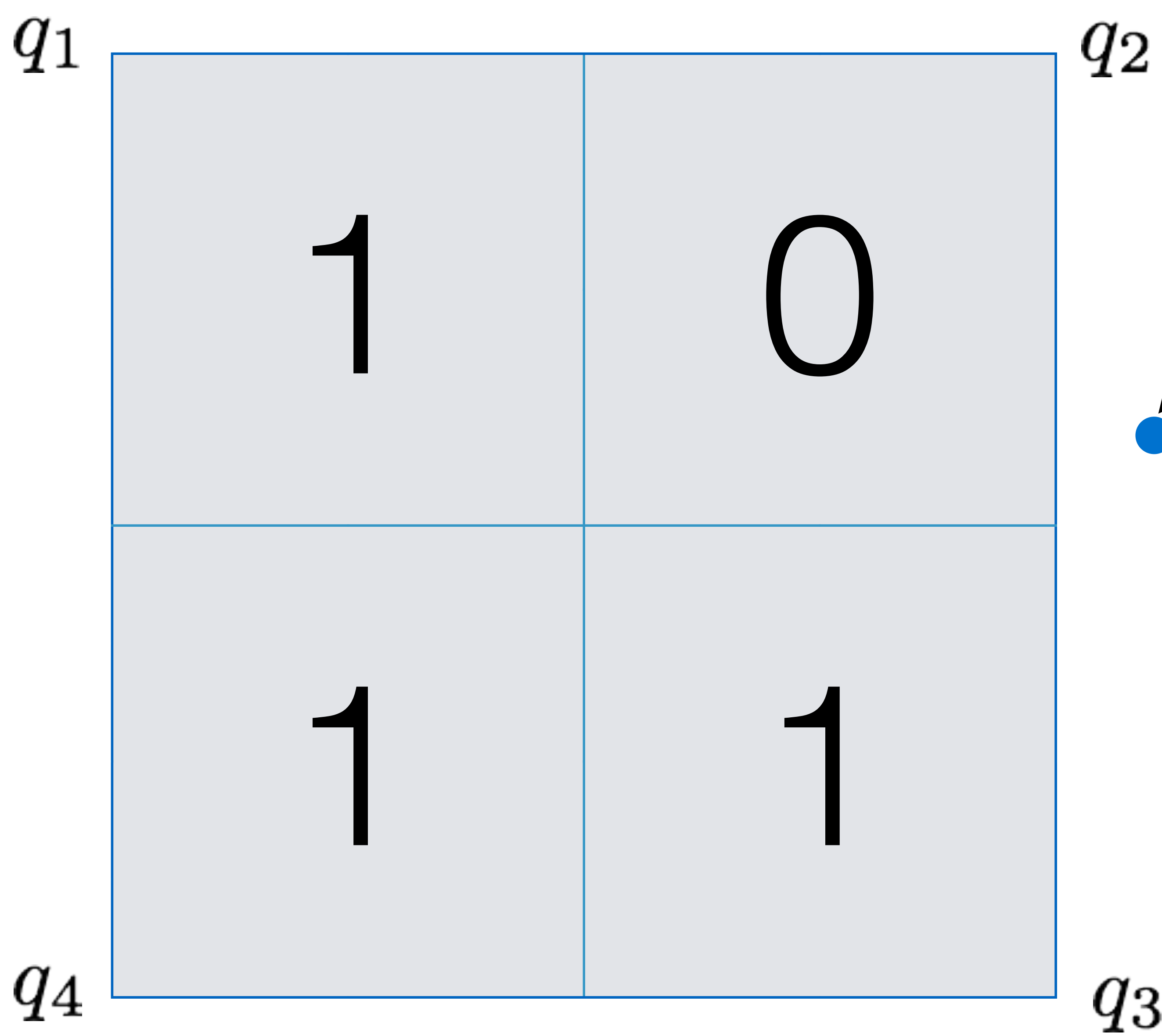
k2-tree

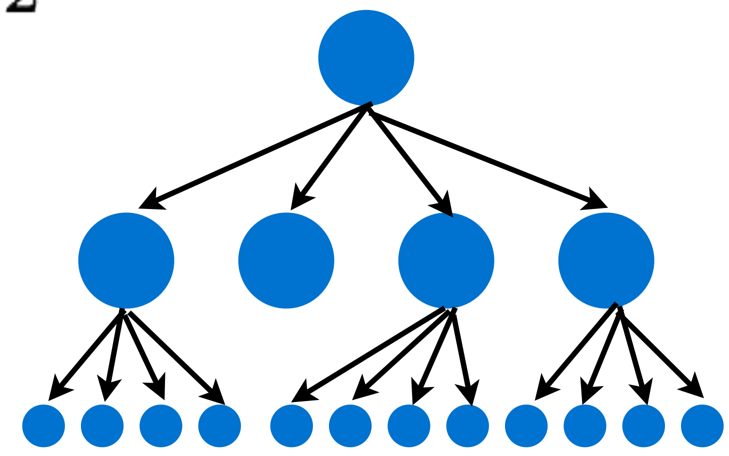


k2-tree

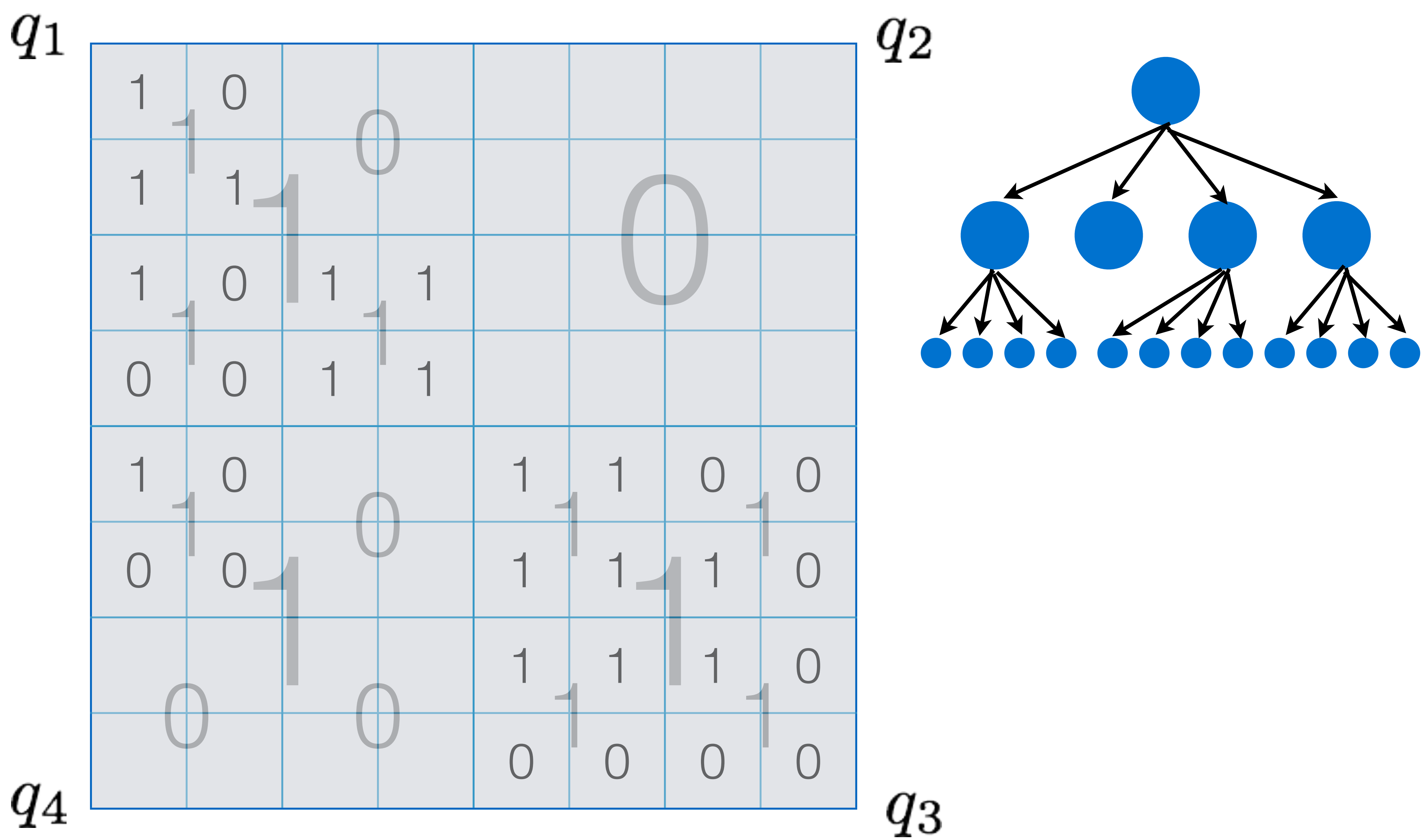


k2-tree

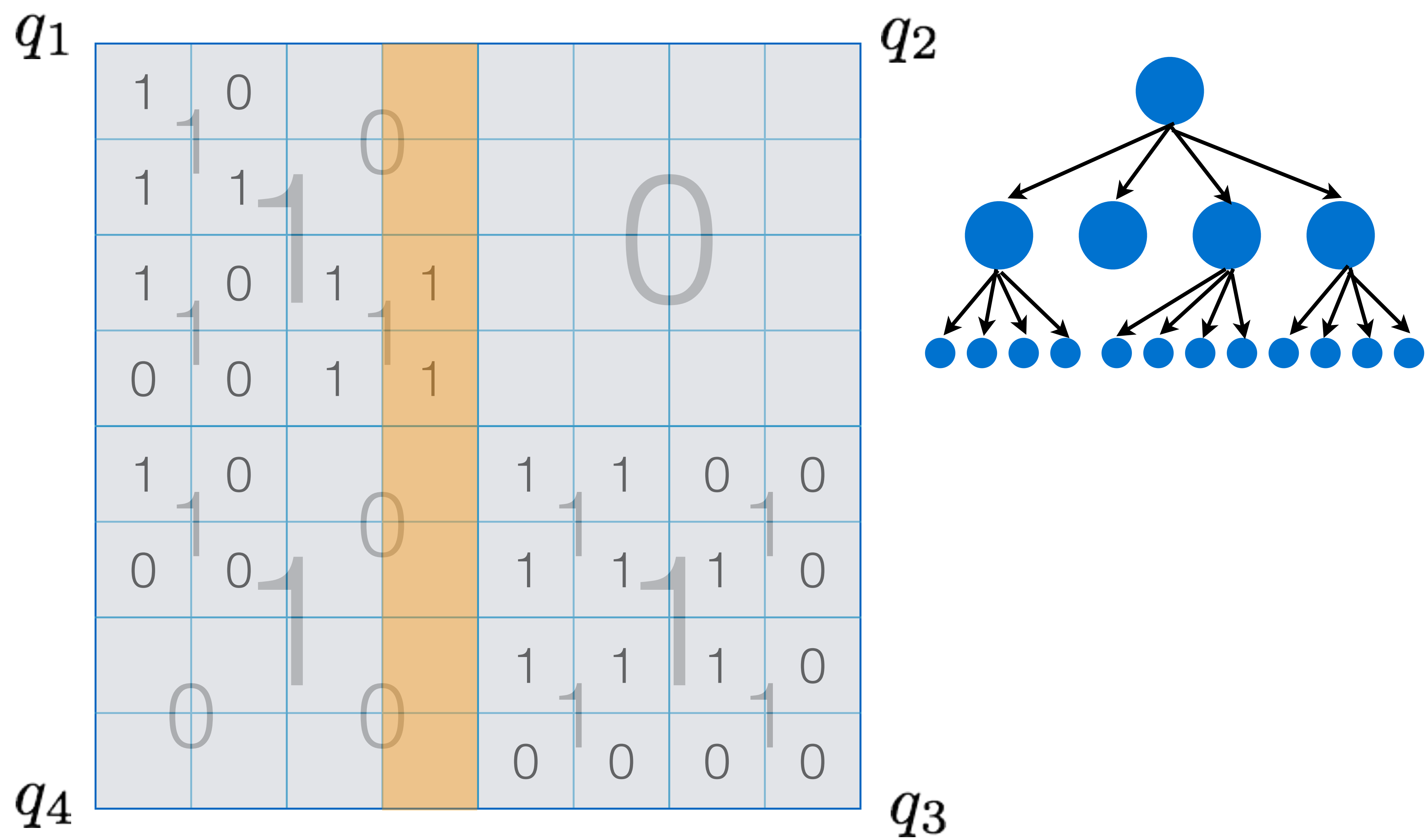




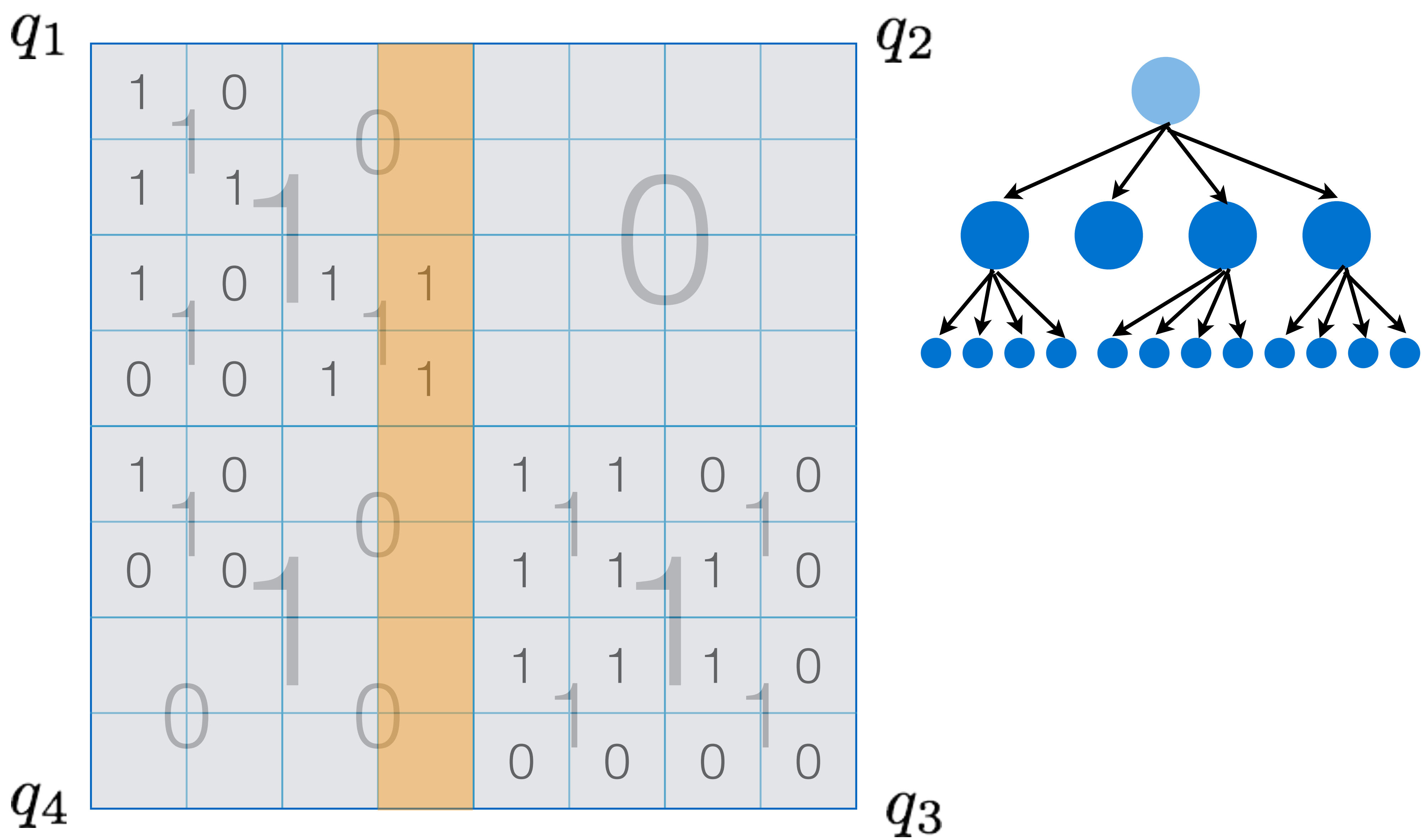
k2-tree



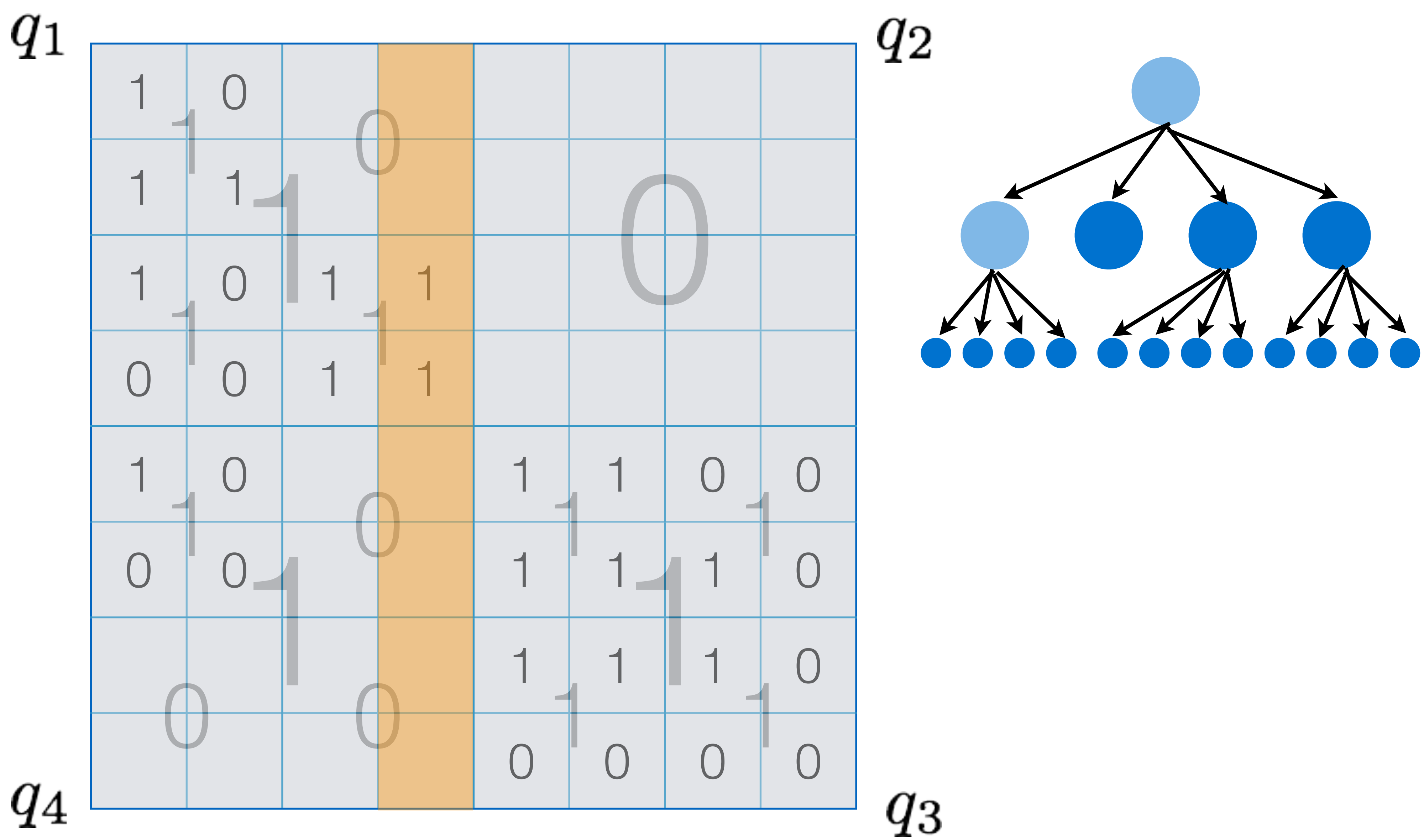
k2-tree



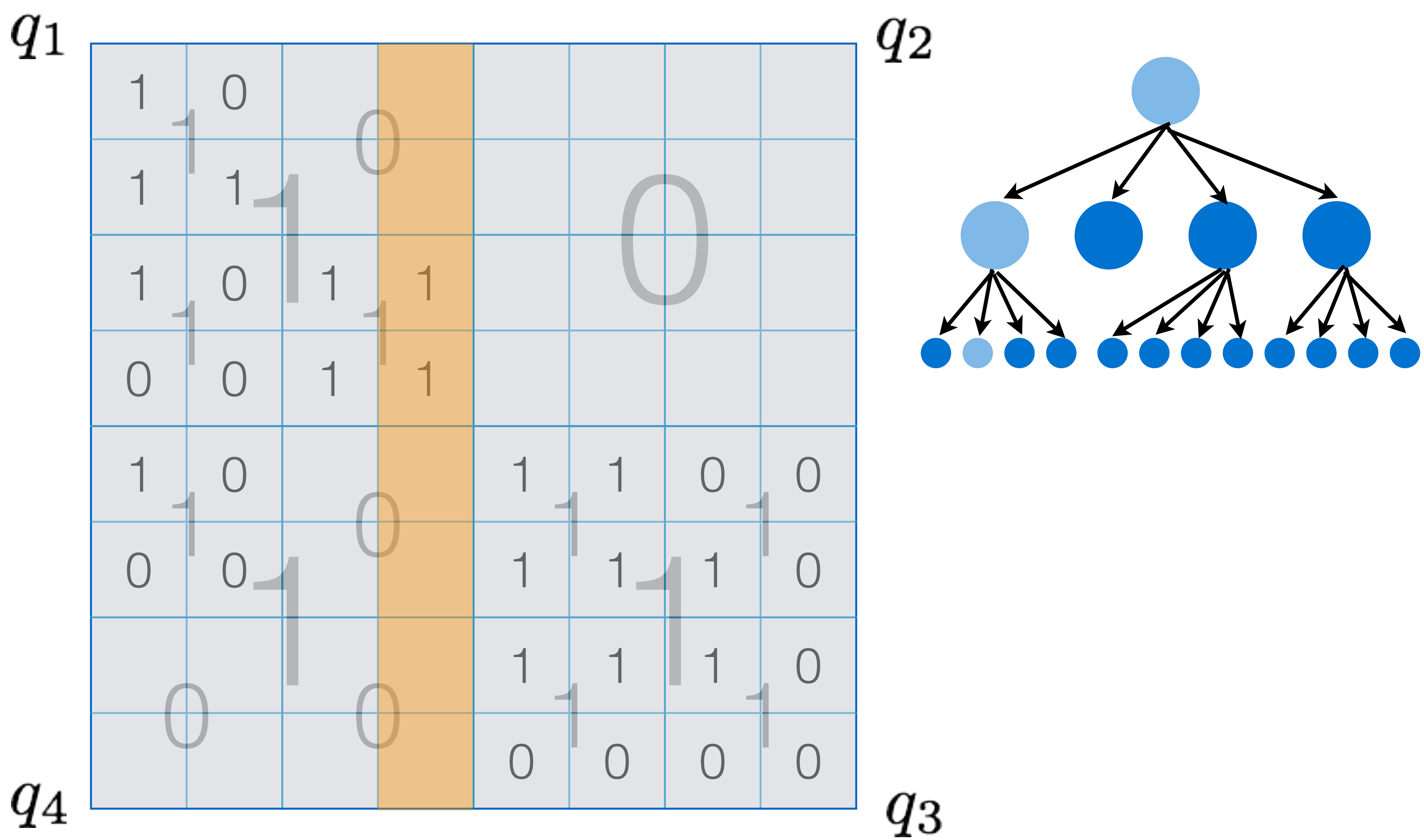
k2-tree



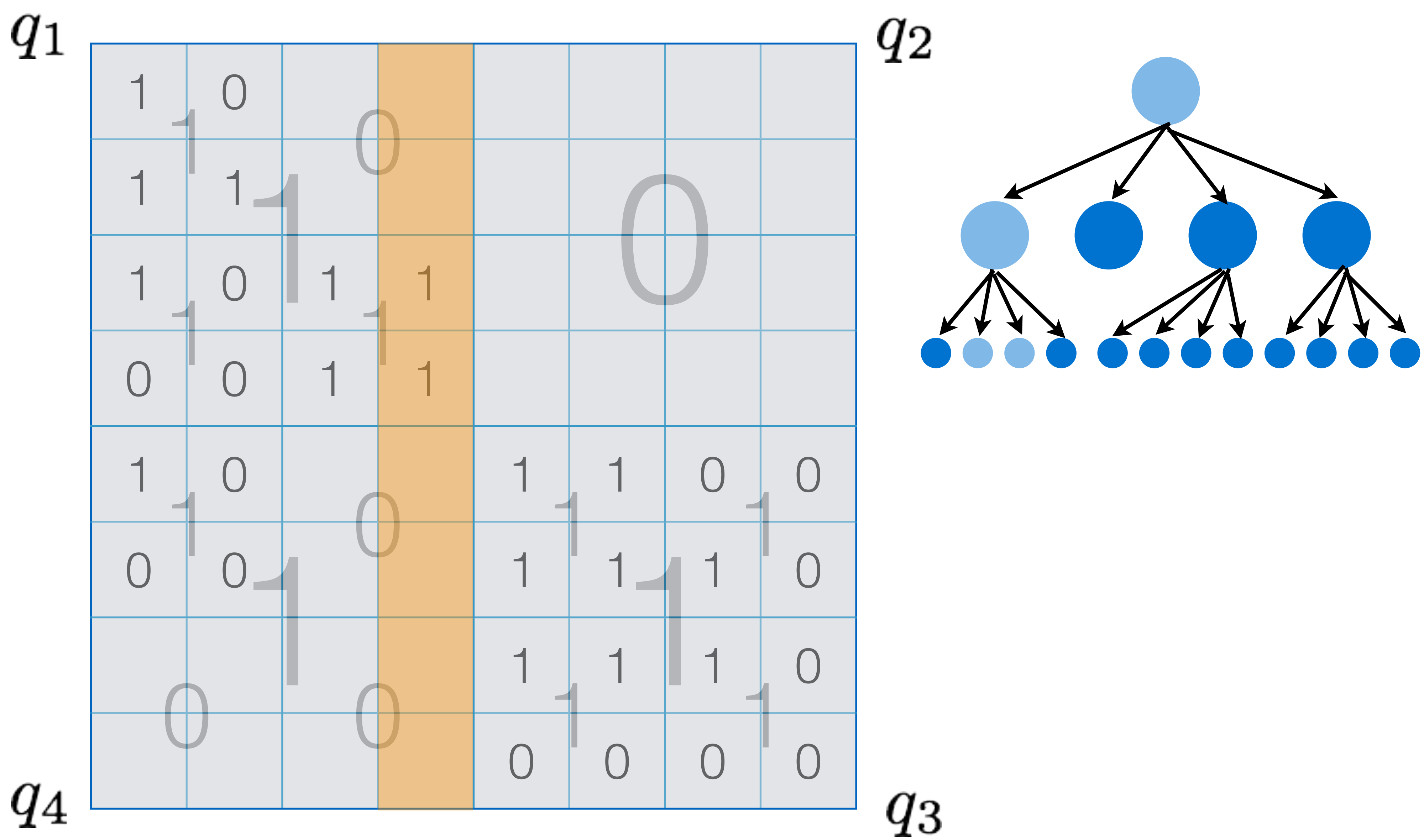
k2-tree



k2-tree

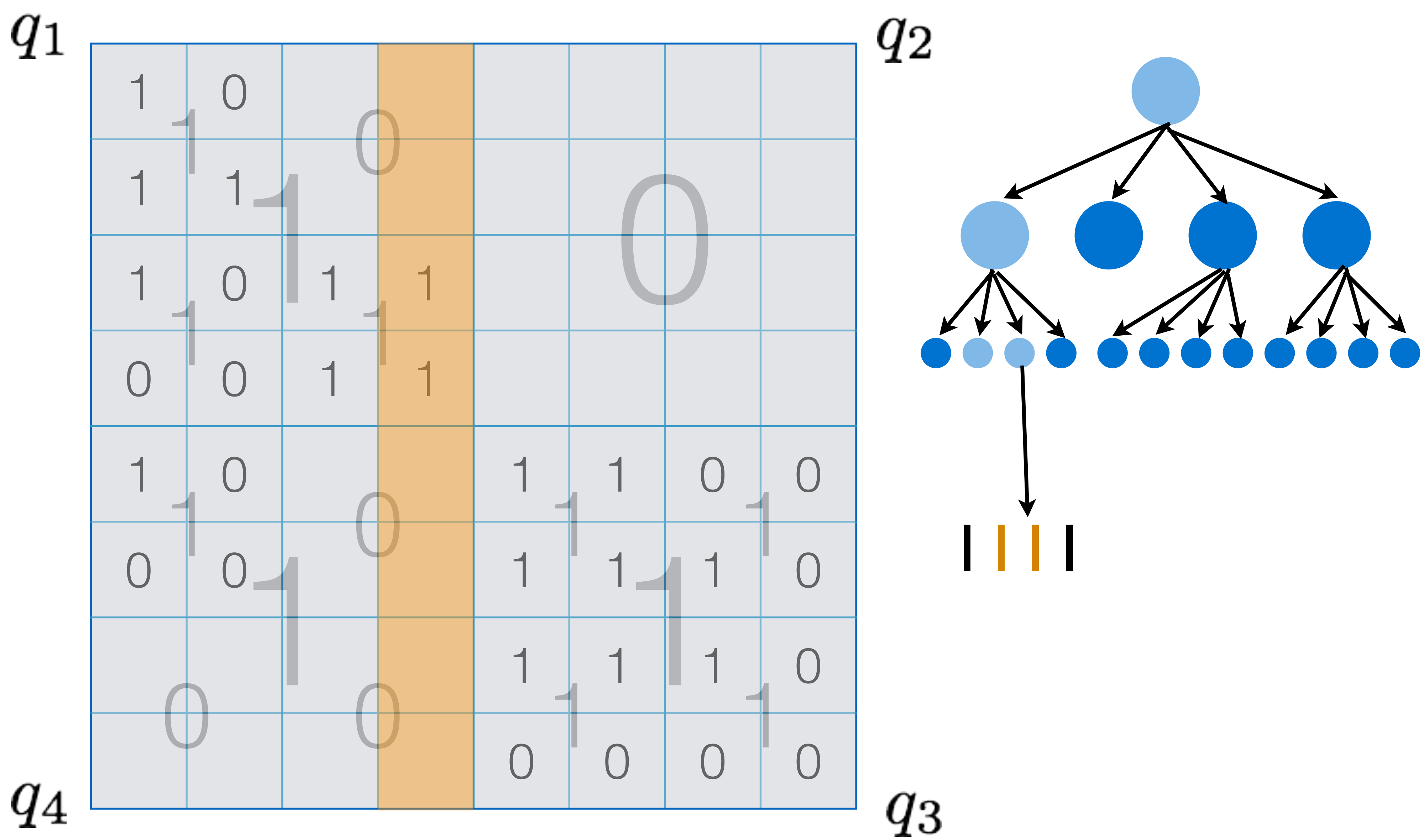


k2-tree

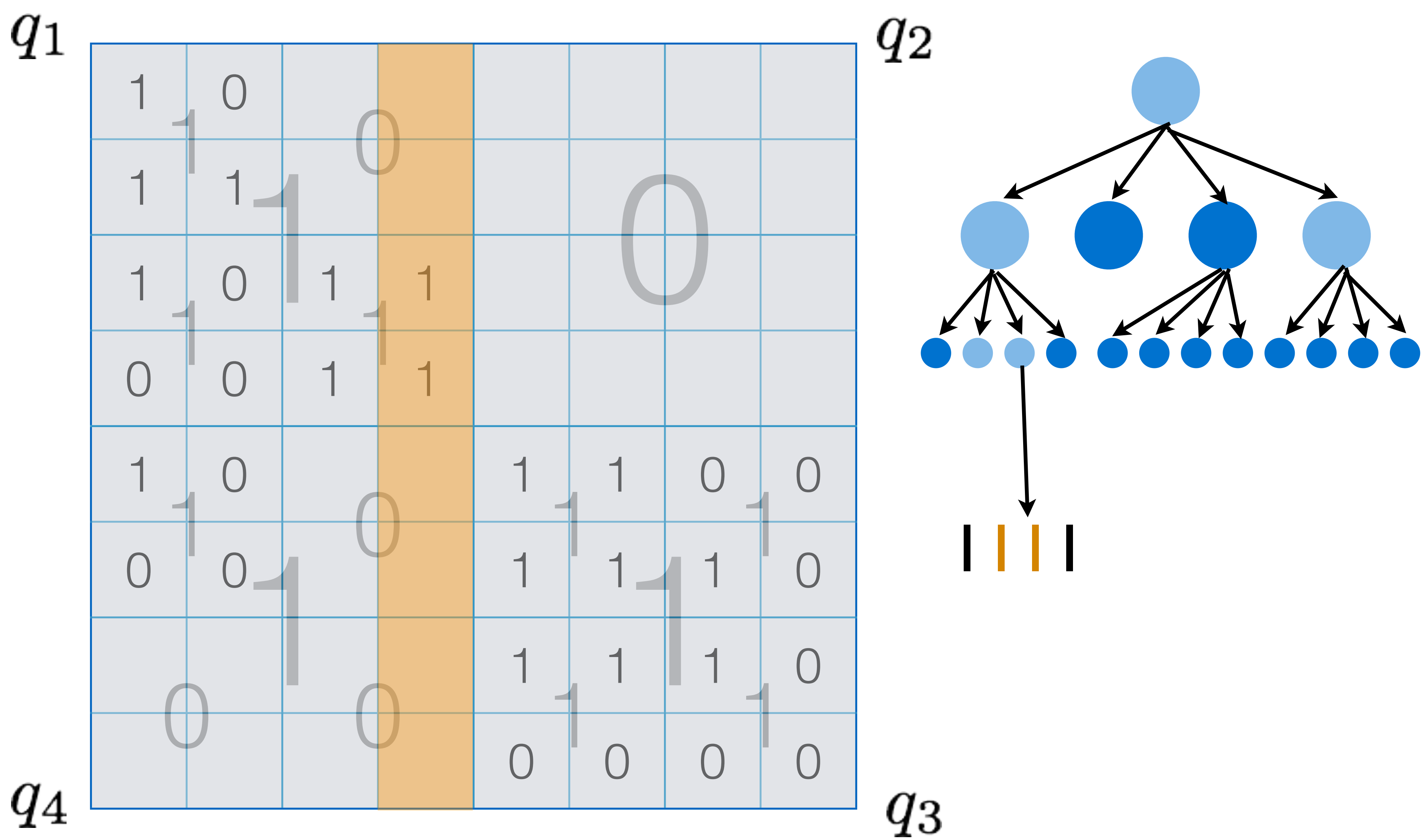


q_1
q_2
q_3
q_4
1 0
1 1
1 0 1 1
0 0 1 1
1 0 1 1 0 0
0 0 1 1 1 0
1 1 1 0
0 0 0 0
1
0
1
0
1
1
1
0
1
0
0
1
1
1
1

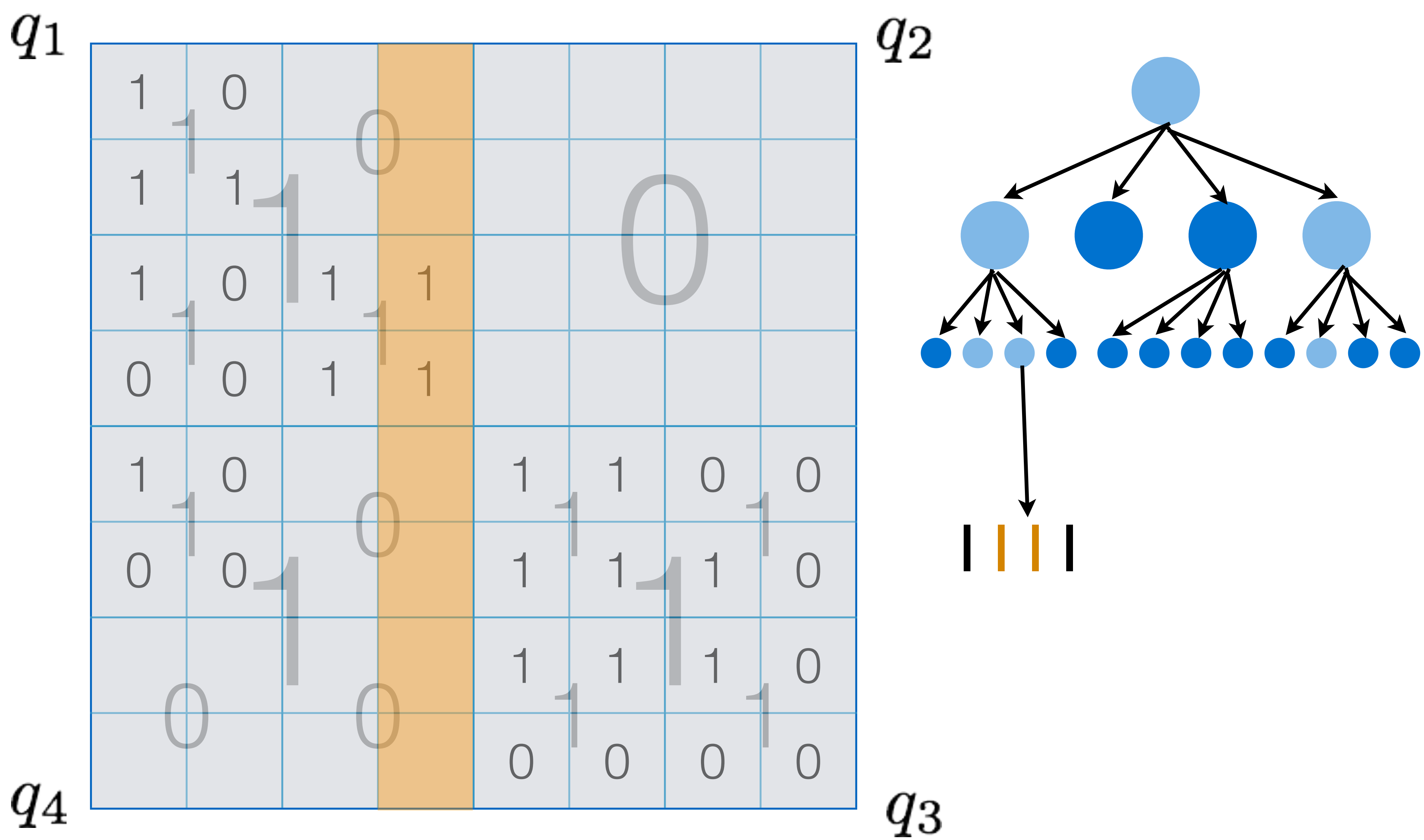
k2-tree



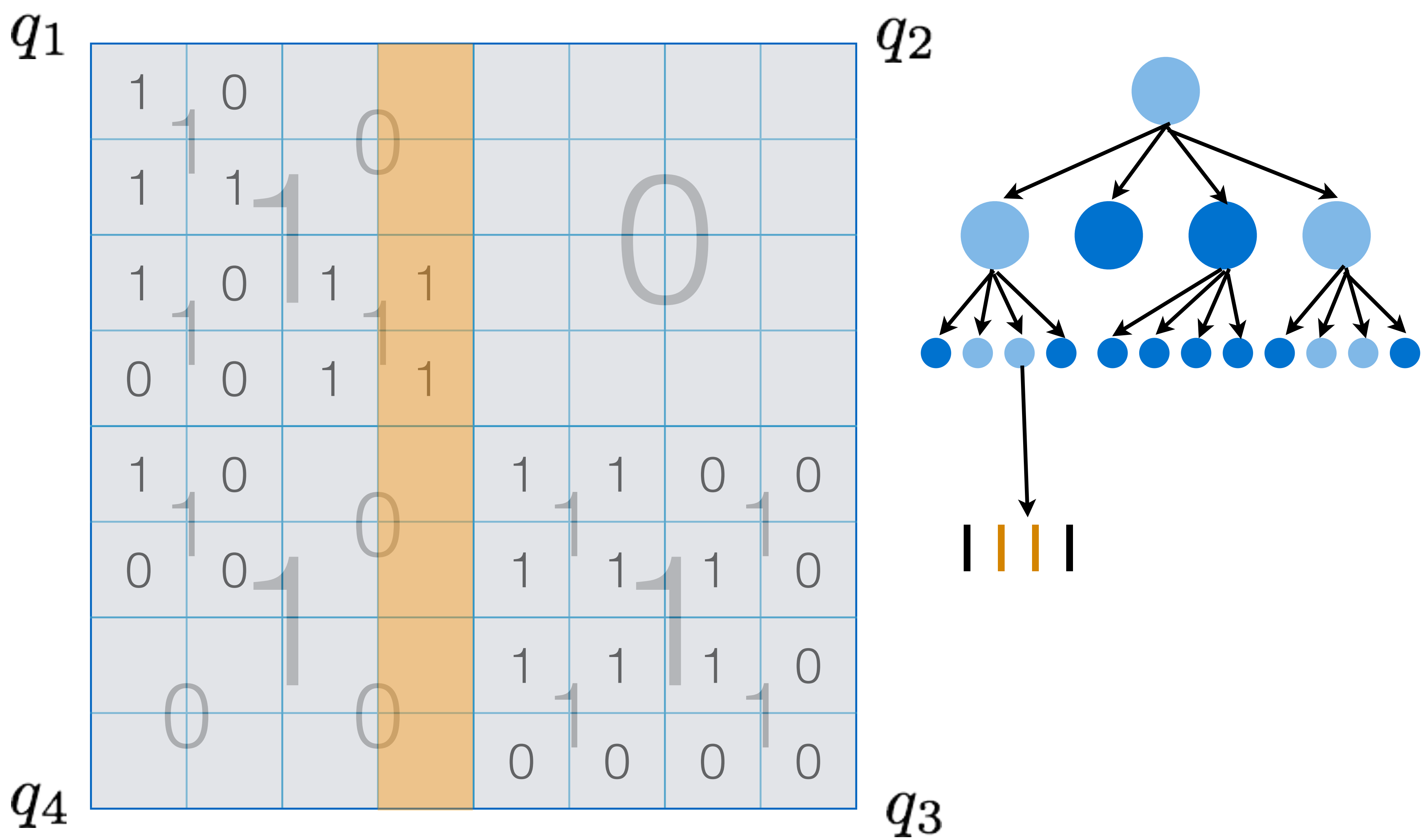
k2-tree



k2-tree



k2-tree



k2trees

- Each node has either 4 or no children
- We can adapt the simple representation for binary trees we saw at the beginning
- This means, we need 1 bit per node!
- But how much is that compared to the information theoretical minimum?

k2tree

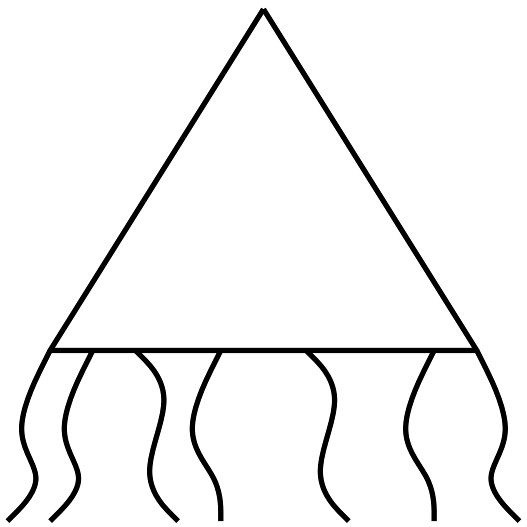
- There are $\binom{n^2}{m}$ graphs with m edges
- The number of bits we need to represent such an object is:

$$\log \binom{n^2}{m}$$

- This is roughly $\mathcal{H} = m \log \frac{n^2}{m} + O(m)$

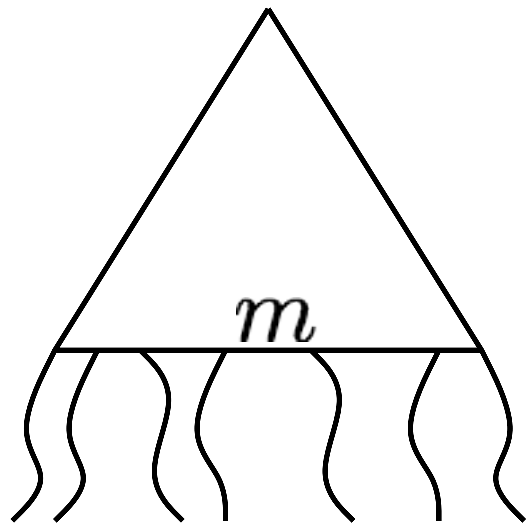
k2tree

- Intuition: the worst that could happen
  - All elements split their path as soon as possible
  - We spend 4 bits per element per level after they split



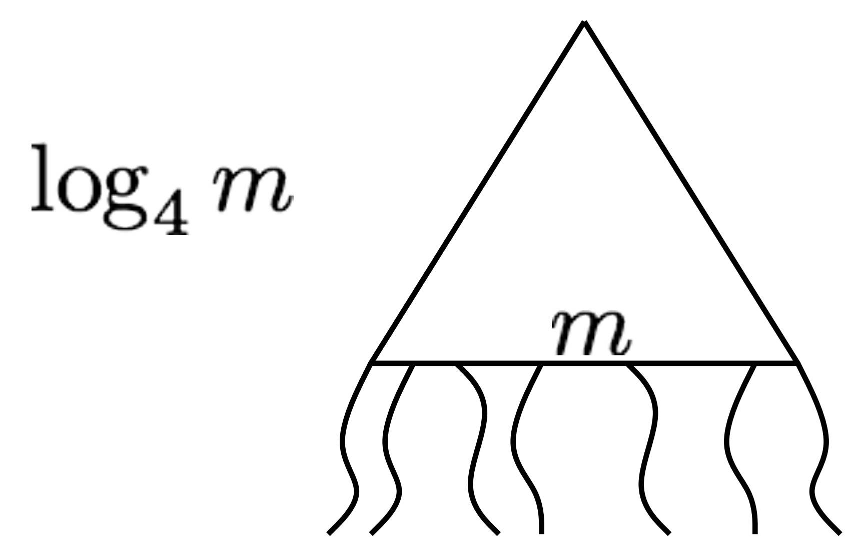
k2tree

- Intuition: the worst that could happen
  - All elements split their path as soon as possible
  - We spend 4 bits per element per level after they split



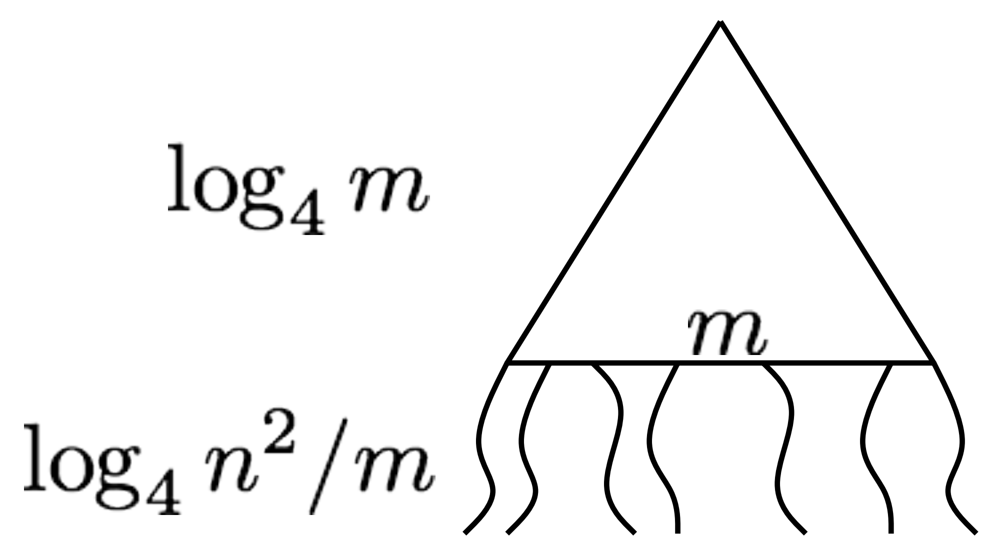
k2tree

- Intuition: the worst that could happen
  - All elements split their path as soon as possible
  - We spend 4 bits per element per level after they split



k2tree

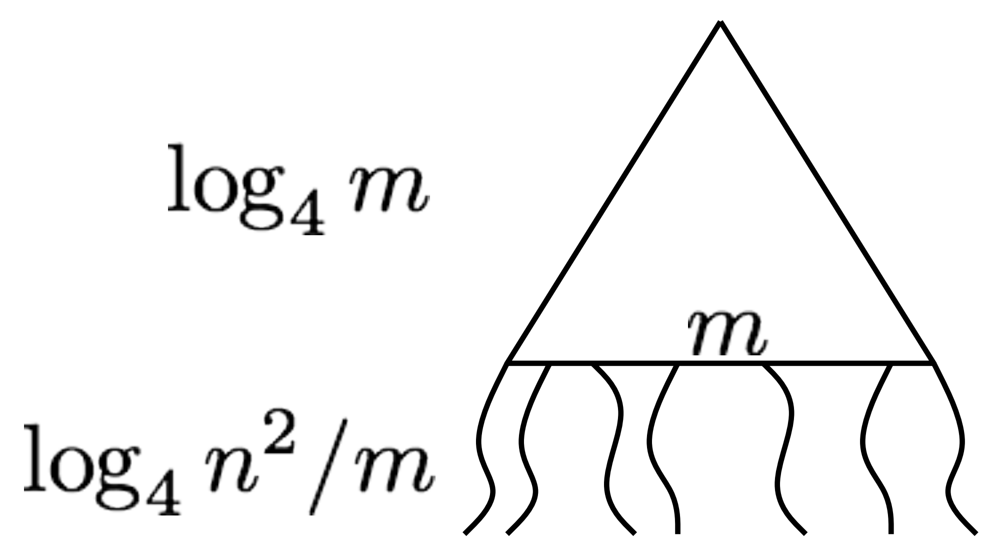
- Intuition: the worst that could happen
  - All elements split their path as soon as possible
  - We spend 4 bits per element per level after they split



k2tree

- Intuition: the worst that could happen
  - All elements split their path as soon as possible
  - We spend 4 bits per element per level after they split

$$4m \log_4 \frac{n^2}{m} + O(m)$$

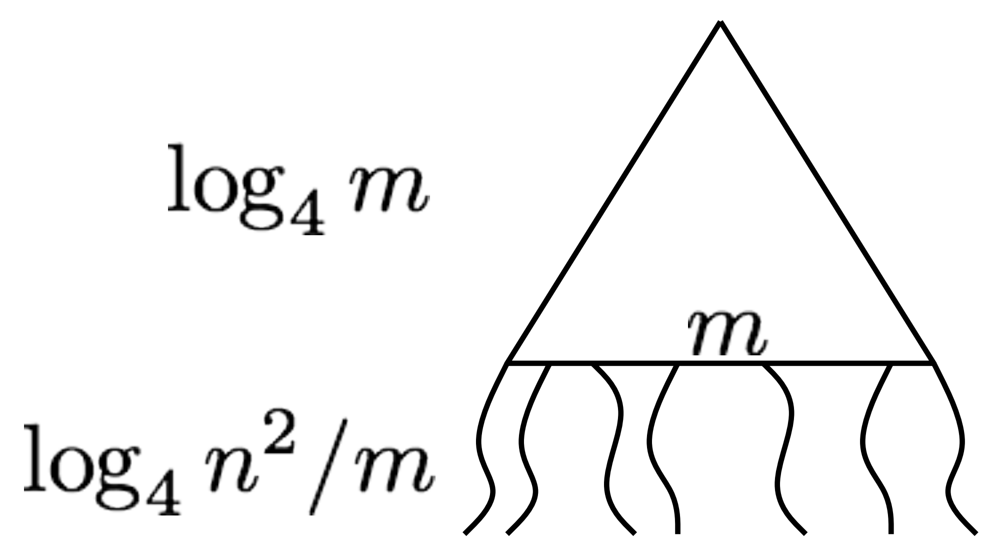


k2tree

- Intuition: the worst that could happen
  - All elements split their path as soon as possible
  - We spend 4 bits per element per level after they split

$$4m \log_4 \frac{n^2}{m} + O(m)$$

$$2\mathcal{H} + O(m)$$



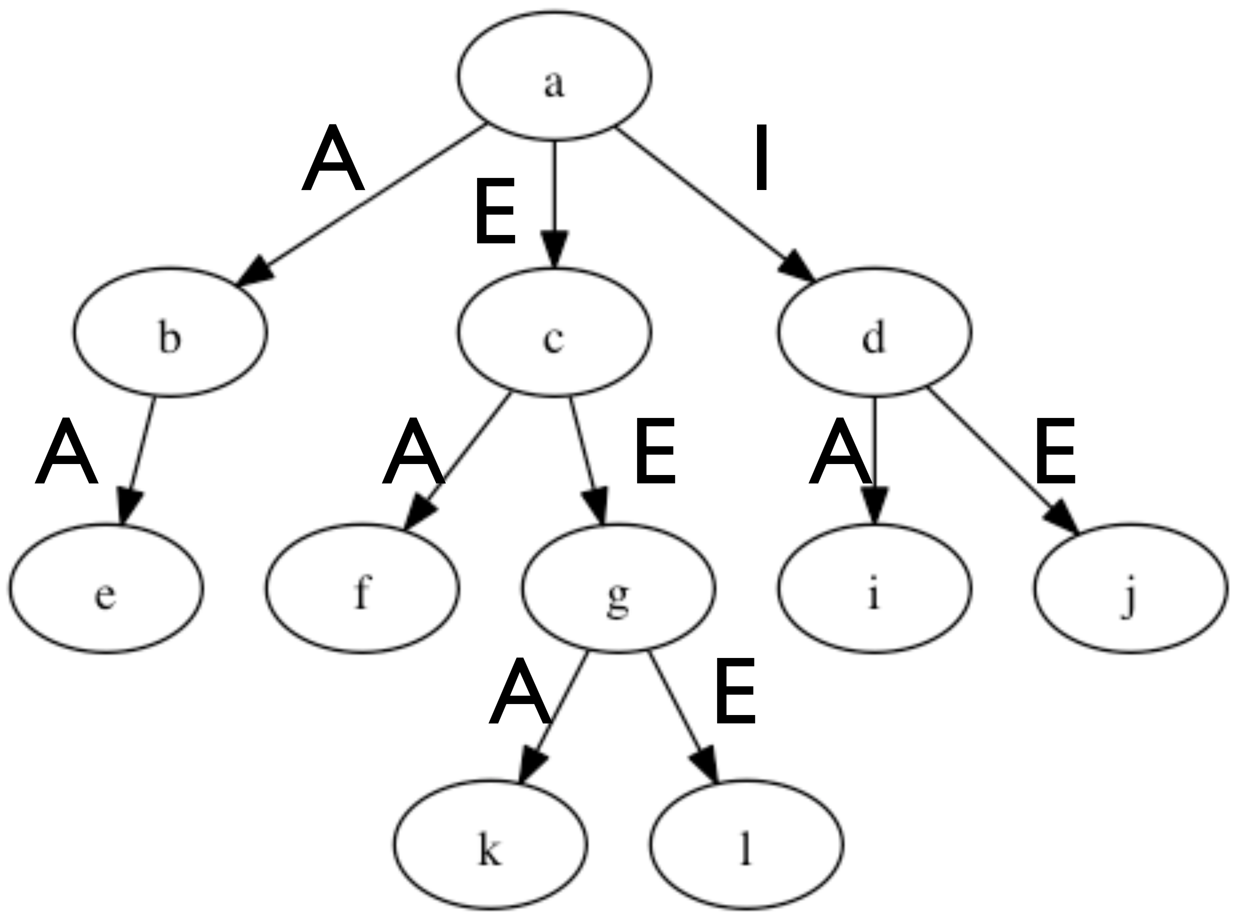
A Simple Trie

A simple trie

- A Trie is simply a tree symbols in the edges
- Very good option for representing string dictionaries

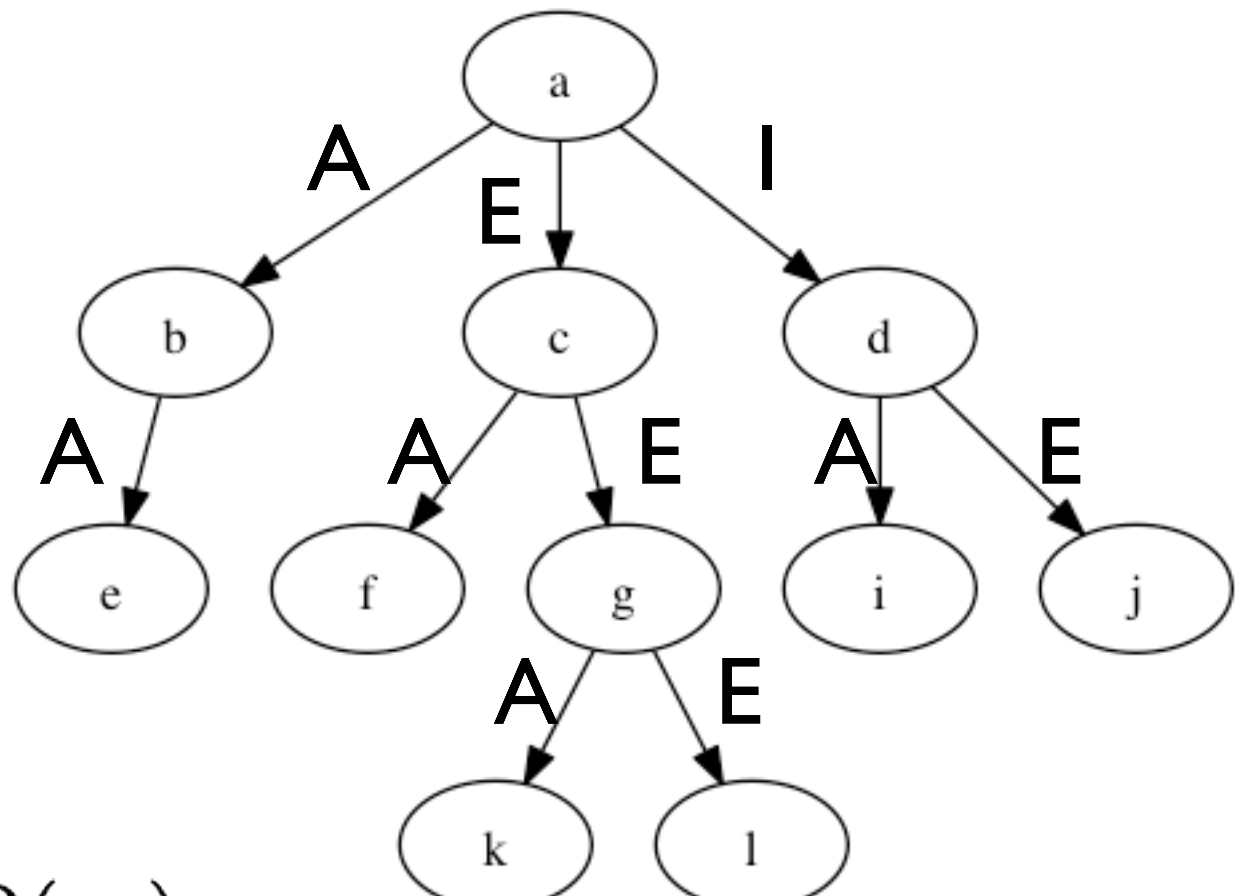
A simple trie

- A Trie is simply a tree symbols in the edges



A simple trie

- A Trie is simply a tree symbols in the edges



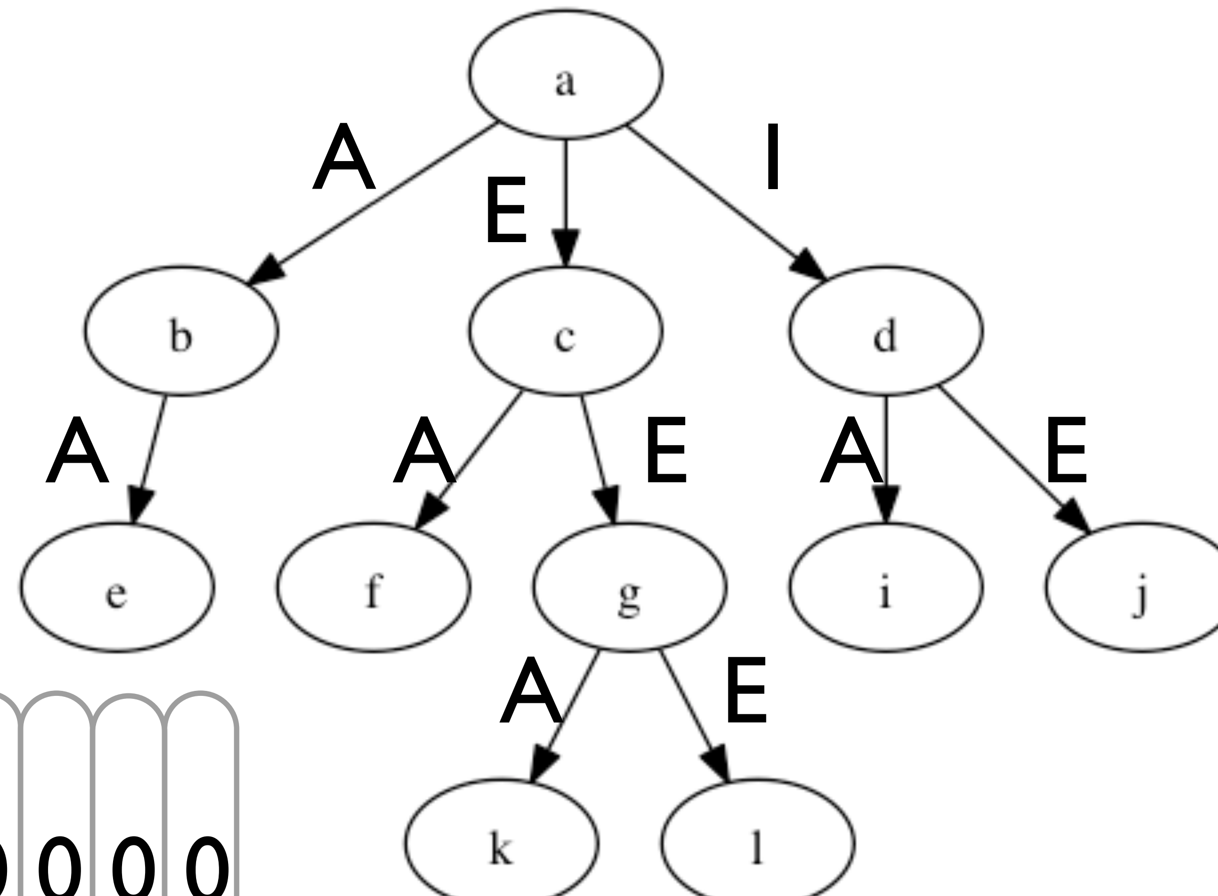
Finding an element: $O(m)$

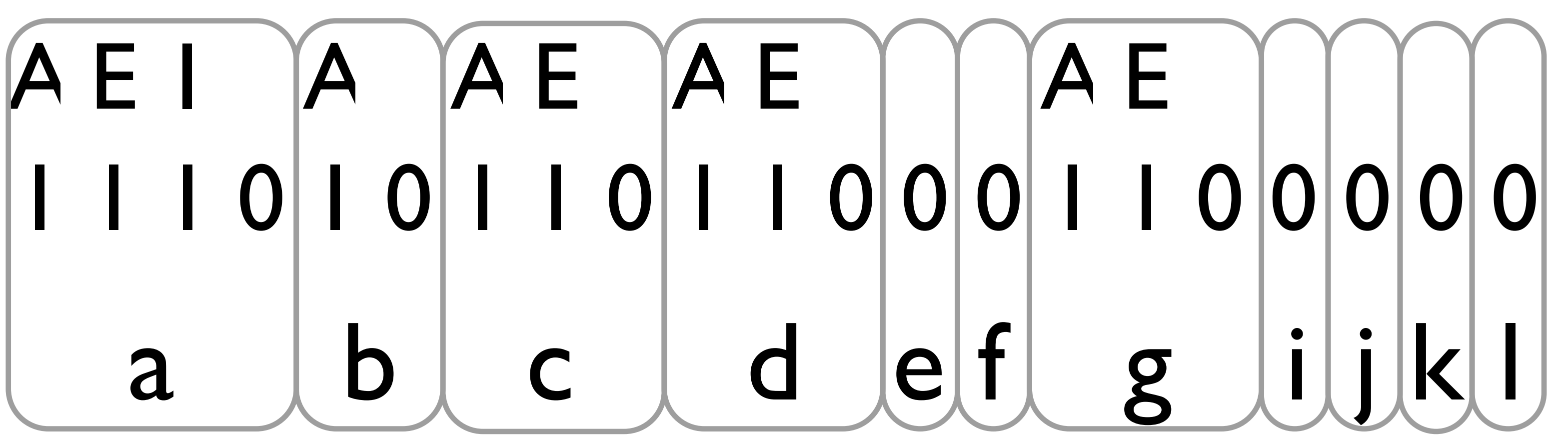
A simple trie

- We will represent the tree using LOUDS
- Each 1 in the LOUDS representation will have a label associated with it
- We want to support the following queries:
  - child(i, a): child of node i through label a, or -1 if it doesn't exist
  - parent-label(i): which label brought me from my parent?

A simple trie

- A Trie is simply a tree symbols in the edges





A simple trie

- Space required by the trie (in bits):

$$n \log \sigma + o(n \log \sigma)$$

- Time for label-related queries:

$$O(\log \sigma) \quad \text{or} \quad O(\log \log \sigma)$$

A simple trie

- Space required by the trie (in bits):

$$n \log \sigma + o(n \log \sigma)$$

- Time for searching a string of length m

$$O(m \log \sigma)$$

- ... or

$$O(m \log \log \sigma)$$

Wrapping up

LIBCDS2

- And we are getting more help
  - Alex Bowe
  - Rodrigo Cánovas
  - Roberto Konow

LIBCDS2

- It's based on libcds, but for big datasets
- We are making some time tradeoffs, but it's easier to use
- It's almost ready to use in multi-threading settings
- Basic support for other languages (at the time Go, Python in process)

LIBCDS

- Version 1 -- “stable”
  - http://libcds.recoded.cl
- Version 2 -- ready to start trying it out
  - http://libcds2.recoded.cl
  - http://github.com/fclaude/libcds2

Conclusions

- We can save considerable space for static data structures using these techniques
- The same principles work in practice for dynamic data structures (but there is still a lot to be done here)
- Try it out and see for yourself how these structures run

References

- Bitmaps: Jacobson ’89; Clark & Munro ’96
- Huffman: Huffman ’52
- LOUDS: Jacobson ’89
- Wavelet Trees: Grossi, Gupta & Vitter ’03
- Permutations: Munro, Raman, Raman & Rao ’03
- GMR: Golynski, Munro & Rao ’06
- K2Tree: Brisaboa, Ladra & Navarro ’09
- DACs: Brisaboa, Ladra & Navarro ’09

Thanks!

Now...