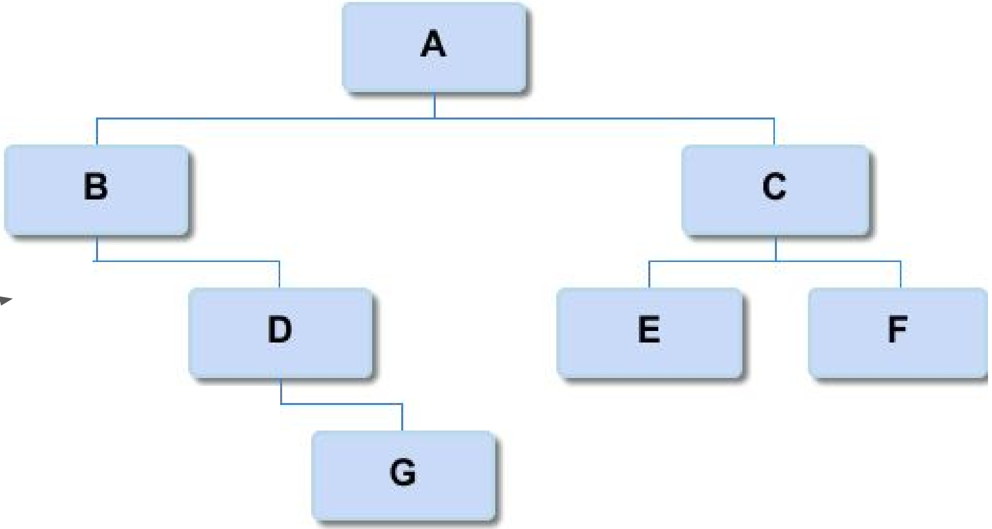
Succinct Data Structures

Calculating Rank and Select

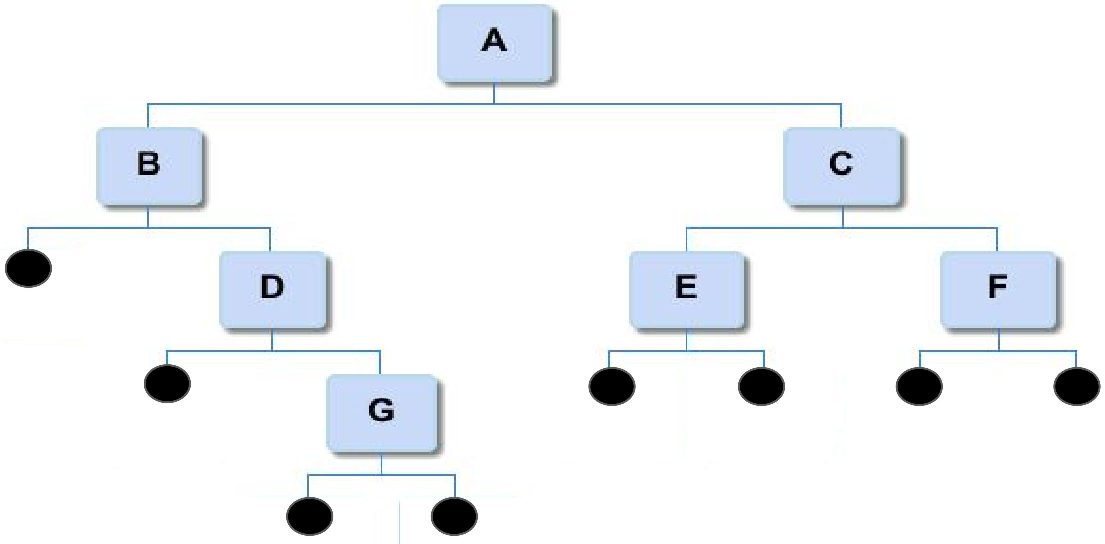
Sam Heilbron



Let the tree be represented by the bit string below

(1)	1	1	0	1	1	1	0	1	0	0	0	0	0	0
A	B	C		D	E	F		G						

2n + 1 bits



(1) 1 1 0 1 1 1 0 1 0 0 0 0 0 0

A B C D E F G

1: Internal Node
0: External Node

Unifies data: Ensures that each node has either 2 children or 0 children

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

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

Navigation

At ith node:

Left child = 2i
Right child = 2i + 1
Parent = ⌊i/2⌋

(Proof by induction)

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

Navigation

At ith node:

Left child = 2i
Right child = 2i + 1
Parent = ⌊i/2⌋

(Proof by induction)

Issue

Working in different namespaces

On one hand, we're <u>counting by internal nodes (number of 1's)</u>

On the other hand, we're <u>counting by position in the array</u> (internal and external nodes)

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

We need some mechanism for counting 1’s, and 0’s and 1’s

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

We need some mechanism for counting 1's, and 0's and 1's

$\text{Rank}_1(i)$ = # 1's at or before position i *(Internal)*

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

We need some mechanism for counting 1's, and 0's and 1's

$Rank_1(i)$ = # 1's at or before position i *(Internal)*

$Select_1(j)$ = Position of the jth 1 bit *(Internal +External)*

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

We need some mechanism for counting 1's, and 0's and 1's

$Rank_1(i)$ = # 1's at or before position i *(Internal)*

$Select_1(j)$ = Position of the jth 1 bit *(Internal +External)*

left-child(i) = $2 * Rank_1(i)$
right-child(i) = $2 * Rank_1(i) + 1$
parent(i) = $Select_1(\lfloor i/2 \rfloor)$

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

We need some mechanism for counting 1's, and 0's and 1's

$\text{Rank}_1(i)$ = # 1's at or before position i *(Internal)*

$\text{Select}_1(j)$ = Position of the jth 1 bit *(Internal +External)*

left-child(i) = $2 * \text{Rank}_1(i)$
right-child(i) = $2 * \text{Rank}_1(i) + 1$
parent(i) = $\text{Select}_1(\lfloor i/2 \rfloor)$

If rank, select can be done in constant time, so too can left-child, right-child, and parent

Rank

Goal: o(n) space

Use a lookup table for bit strings of length x:

O(Bit strings of length x * possible answers to query (can query each i of n bits) * Bits for each answer (between 0 and x-1))

Goal: o(n) space

Use a lookup table for bit strings of length x:

O(	2^X	*	possible answers to query (can query each i of n bits)	*	Bits for each answer (between 0 and x-1)	)

Goal: o(n) space

Use a lookup table for bit strings of length x:

$$O\left(2^{x} \quad * \quad x \quad * \quad \text{Bits for each answer (between 0 and x-1)} \right)$$

Goal: o(n) space

Use a lookup table for bit strings of length x:

$$O(\quad 2^x \quad * \quad x \quad * \quad \log(x) \quad)$$

Goal: o(n) space

Use a lookup table for bit strings of length x:

$$O(2^{x} * x * \log x)$$

Goal: o(n) space

Use a lookup table for bit strings of length x:

$$O(2^X \ * \ x \ * \log x)$$

In order to achieve o(n) space, what is x?

Goal: o(n) space

Use a lookup table for bit strings of length x:

$$O(2^{X} \ * \ x \ * \log x)$$

In order to achieve o(n) space, what is x?

Let $x = \frac{1}{2} \lg n$ $\rightarrow$ $O(\sqrt{n} * \lg(n) * \lg(\lg(n)))$

Goal: o(n) space

Use a lookup table for bit strings of length x:

$$O(2^X * x * \log x)$$

In order to achieve o(n) space, what is x?

Let $x = \frac{1}{2} \lg n$ → $O(\sqrt{n} * \lg(n) * \lg(\lg(n)))$

If we can have bit strings of logarithmic size, we're all set!

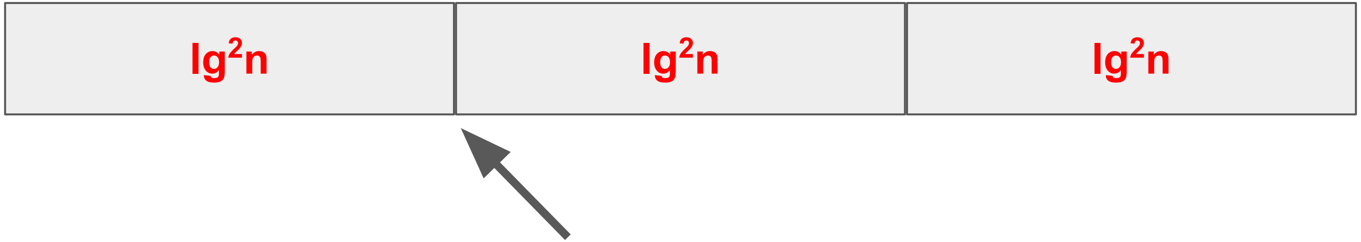
How do we reduce linearly sized bit strings

Step 1: Reduce to size $\lg^2 n$

$\lg^2 n$	$\lg^2 n$	$\lg^2 n$

How do we reduce linearly sized bit strings

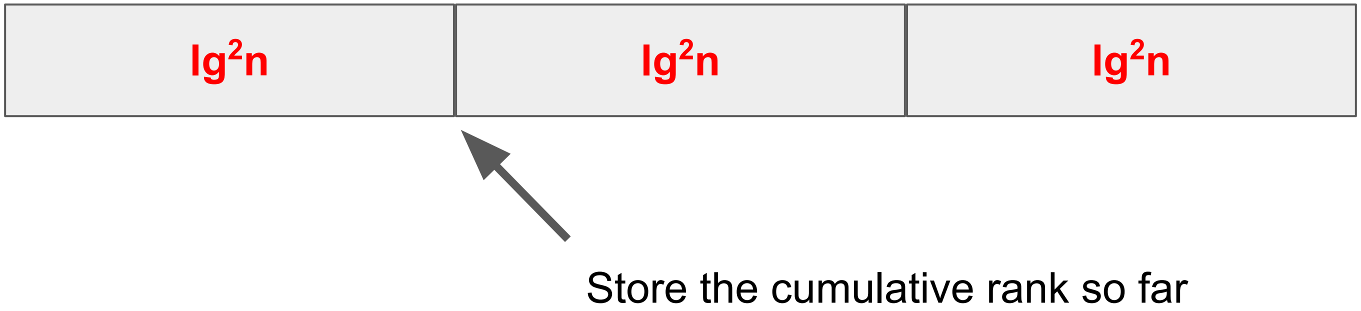
Step 1: Reduce to size $\lg^2 n$



Store the cumulative rank so far

How do we reduce linearly sized bit strings

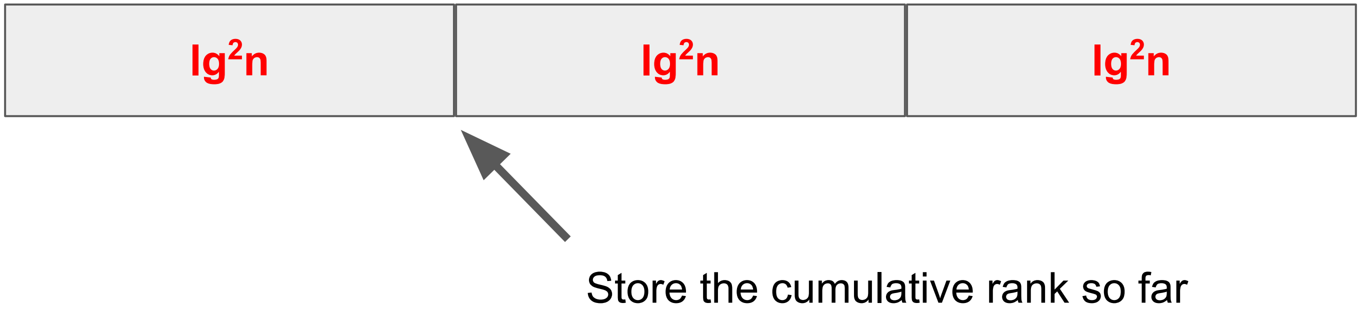
Step 1: Reduce to size $\lg^2 n$



$$O(\ \#\ \text{sections} \ * \ \#\ \text{bits / section}\ )$$

How do we reduce linearly sized bit strings

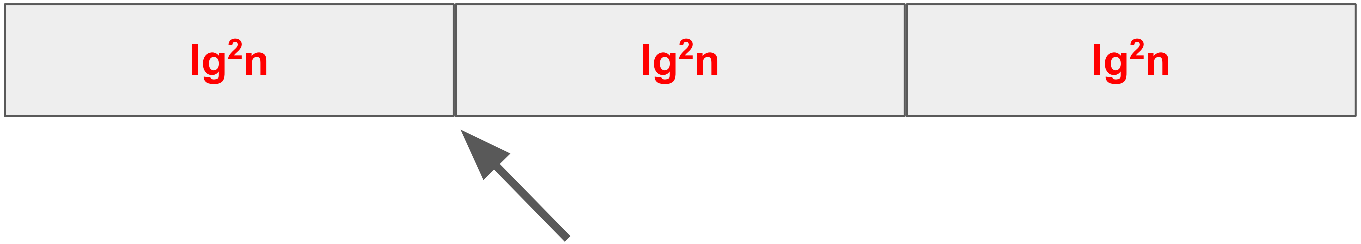
Step 1: Reduce to size $\lg^2 n$



$$O\left(n/\lg^2 n \quad * \quad \lg(n) \right)$$

How do we reduce linearly sized bit strings

Step 1: Reduce to size $\lg^2 n$



Store the cumulative rank so far

$$O(n / \lg n) \quad \approx o(n)$$

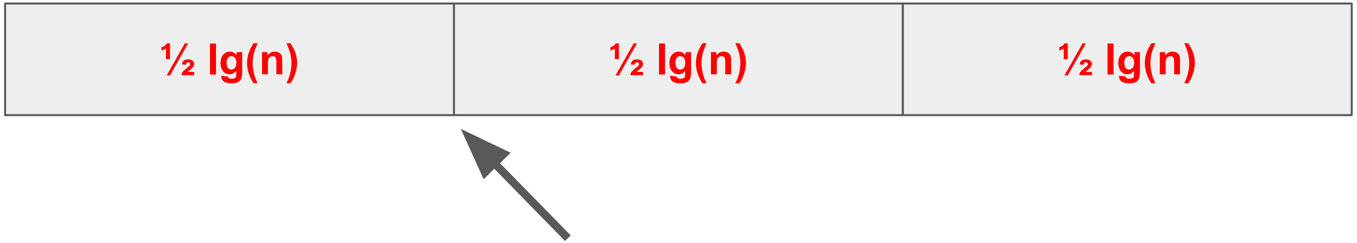
How do we reduce linearly sized bit strings?

Step 2: Reduce each chunk to size ½ lg(n)

½ lg(n)	½ lg(n)	½ lg(n)

How do we reduce linearly sized bit strings?

Step 2: Reduce each chunk to size ½ lg(n)



Store the cumulative rank **within the chunk**

How do we reduce linearly sized bit strings?

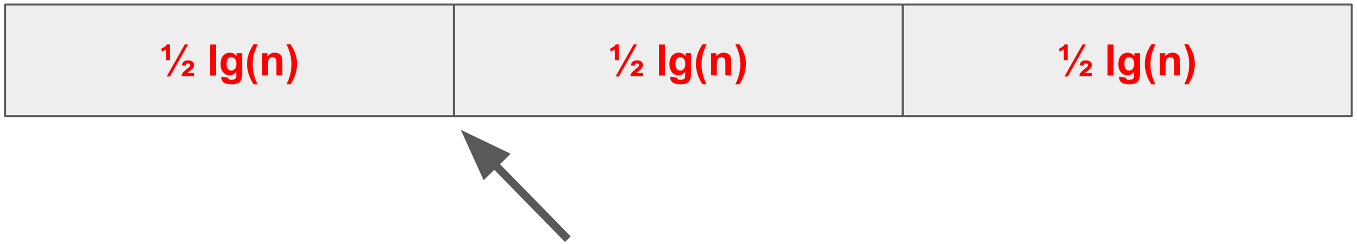
Step 2: Reduce each chunk to size ½ lg(n)

Store the cumulative rank within the chunk

Why not just subdivide before? Why does this help?

How do we reduce linearly sized bit strings?

Step 2: Reduce each chunk to size ½ lg(n)



Store the cumulative rank within the chunk

Why not just subdivide before? Why does this help?

The size of a chunk is lg^2n, so you only need lg(lg(n)) bits to write down the cumulative rank

How do we reduce linearly sized bit strings?

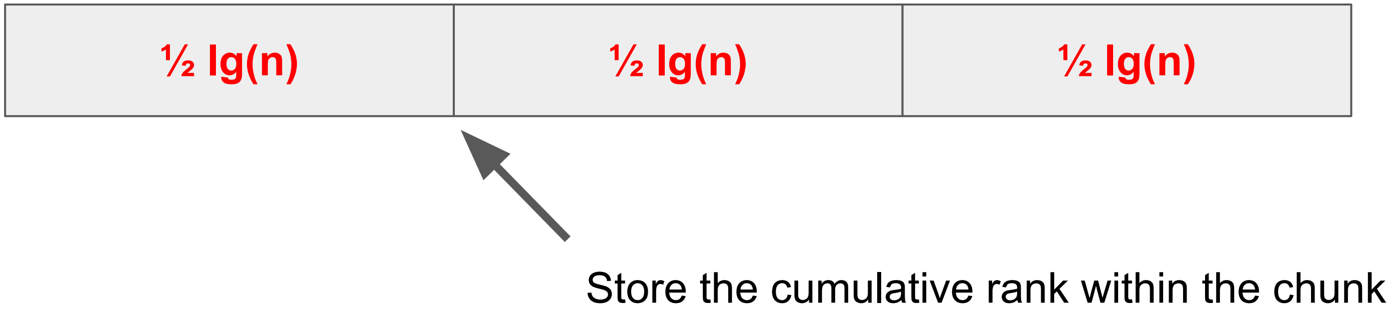
Step 2: Reduce each chunk to size ½ lg(n)

Store the cumulative rank within the chunk

$$O(\ \#\ \text{sections} \quad * \quad \#\ \text{bits / section}\ )$$

How do we reduce linearly sized bit strings?

Step 2: Reduce each chunk to size ½ lg(n)



$$O\left(n/\lg(n) \quad * \quad \lg(\lg(n)) \right)$$

How do we reduce linearly sized bit strings?

Step 2: Reduce each chunk to size ½ lg(n)

Store the cumulative rank within the chunk

$$O\left(n/\lg(n) * \lg(\lg(n)) \right) \approx o(n)$$

How do we reduce linearly sized bit strings?

Step 2: Reduce each chunk to size ½ lg(n)

½ lg(n)	½ lg(n)	½ lg(n)

Store the cumulative rank within the chunk

$$O(\ n/\lg(n) * \lg(\lg(n))\ )$$ $\approx o(n)$

We're all set!

How do we reduce linearly sized bit strings?

Step 3: Calculate Rank

1. Find which chunk you're in (integer division, since each chunk can be stored in an array from Step 1)
2. Find which subchunk you're in (each subchunk stored in an array from Step 2)
3. Find rank of element in subchunk (use lookup table)

Rank = rank of chunk +
rank of subchunk +
rank of element in subchunk

O(1) time, O(n/lg(n) * lg(lg(n))) space

How do we reduce linearly sized bit strings?

Step 4: Calculate Select

Select is handled in a very similar way to rank

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

We need some mechanism for counting 1's, and 0's and 1's

$Rank_1(i)$ = # 1's at or before position i *(Internal)*

$Select_1(j)$ = Position of the jth 1 bit *(Internal +External)*

left-child(i) = $2 * Rank_1(i)$
right-child(i) = $2 * Rank_1(i) + 1$
parent(i) = $Select_1(\lfloor i/2 \rfloor)$

If rank, select can be done in constant time, so too can left-child, right-child, and parent

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

We need some mechanism for counting 1's, and 0's and 1's

$Rank_1(i)$ = # 1's at or before position i *(Internal)*

$Select_1(j)$ = Position of the jth 1 bit *(Internal +External)*

left-child(i) = $2 * Rank_1(i)$
right-child(i) = $2 * Rank_1(i) + 1$
parent(i) = $Select_1(\lfloor i/2 \rfloor)$

Since *rank, select can be done in constant time, so too can left-child, right-child, and parent*