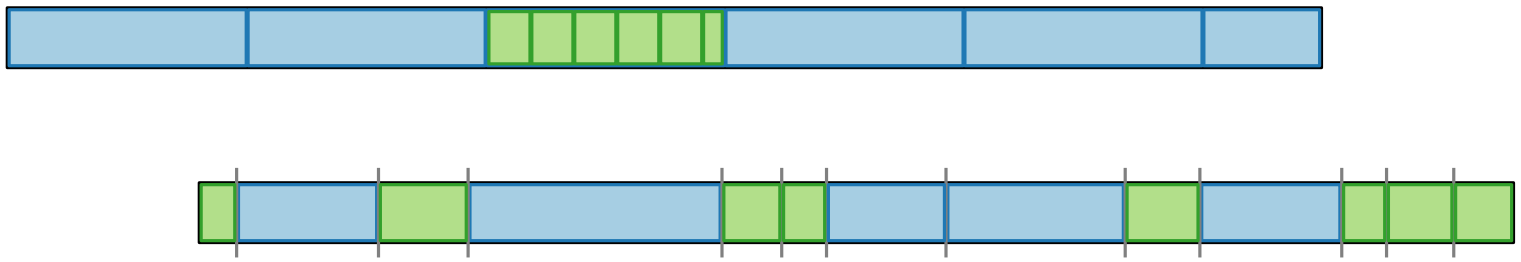
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

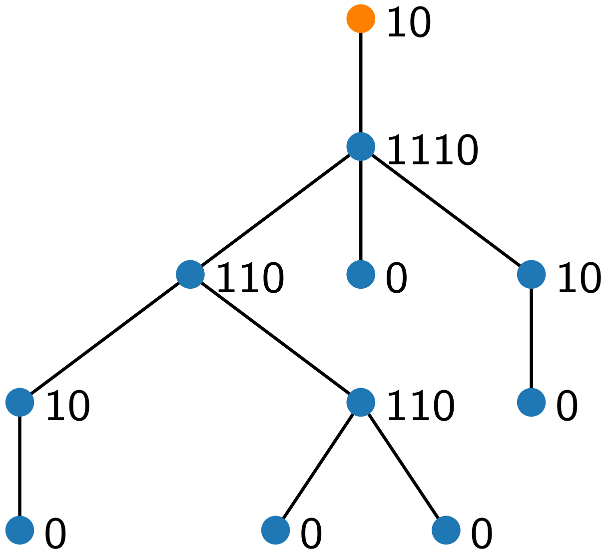
Advanced Algorithms

Succinct Data Structures

Indexable Dictionaries and Trees

Johannes Zink · WS23/24





Data Structures – Informal Definition

A **data structure** is a concept to

- **store**,
- **organize**, and
- **manage** data.

As such, it is a collection of

- **data values**,
- their **relations**, and
- the **operations** that can be applied to the data.

$\Rightarrow$

- What do we represent?
- How much space is required?
- Dynamic or static?
- Which operations are defined?
- How fast are they?

Remarks.

- We look at data structures as a designer/implementer (and not necessarily as a user).
- To define a data structure and to implement it are two different tasks.

Succinct Data Structures

Goal.

- Use space "close" to information-theoretical minimum,
- but still support time-efficient operations.

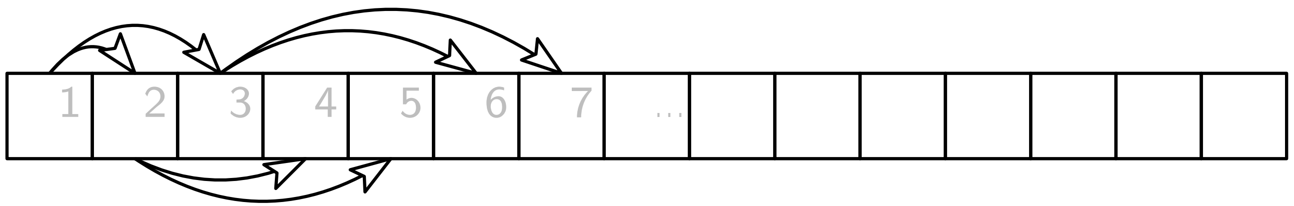
Let L be the information-theoretical lower bound to represent a class of objects.
Then a data structure, which still supports *time-efficient* operations, is called

- **implicit**, if it takes $L + O(1)$ bits of space;
- **succinct**, if it takes $L + o(L)$ bits of space;
- **compact**, if it takes $O(L)$ bits of space.

Examples?

Examples for Implicit Data Structures

- **arrays** to represent lists
  - but why not linked lists?
- **1-dim arrays** to represent multi-dimensional arrays
- **sorted arrays** to represent sorted lists
  - but why not binary search trees?
- **arrays** to represent complete binary trees and heaps



$\texttt{leftChild}(i) = 2i$

$\texttt{rightChild}(i) = 2i + 1$

$\texttt{parent}(i) = \lfloor \frac{i}{2} \rfloor$

And unbalanced trees?

Succinct Indexable Dictionary

Represent a subset $S \subseteq \{1, 2, \ldots, n\}$ and support the following operations in $O(1)$ time:

- `member`(i) returns if $i \in S$
- `rank`(i) = number of elements in S that are less or equal to i
- `select`(j) = j-th element in S
- `predecessor`(i)
- `successor`(i)

How many different subsets of $\{1, 2, \ldots, n\}$ are there? 2^n

How many bits of space do we need to distinguish them?

$\log 2^n = n$ **bits**

our logarithms are all to basis 2, i.e., $\log_2$

Succinct Indexable Dictionary

Represent S with a bit vector b of length n where

$$b[i] = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases}$$

plus $o(n)$-space data structures to answer in $O(1)$ time

- $\texttt{rank}(i) = \#$ 1s at or before position i number of
- $\texttt{select}(j) =$ position of j-th 1 bit

$\Rightarrow$ Exercise: Use these methods to answer $\texttt{predecessor}(i)$ and $\texttt{successor}(i)$ in $O(1)$ time.

$S = \{3, 4, 6, 8, 9, 14\}$ where $n = 15$

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

$\texttt{select}(5) = 9$

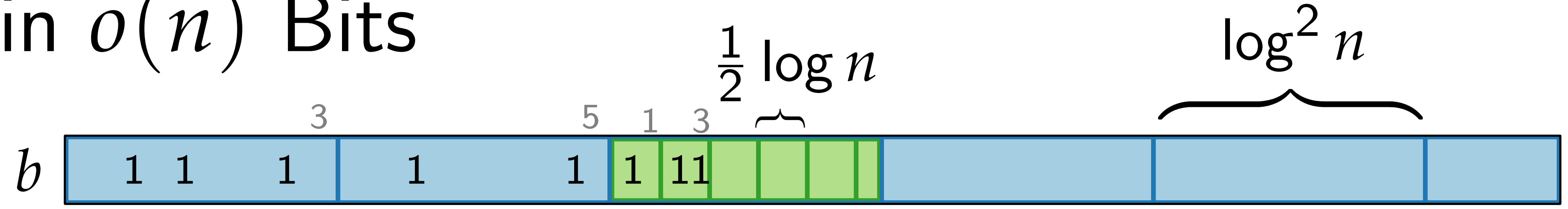
$\texttt{rank}(9) = 5 = \texttt{rank}(12)$

$\texttt{rank}(15) = 6$

$\texttt{member}(i)$ can trivially be answered in $O(1)$ time
(assuming that we can access any entry in constant time)

Rank in $o(n)$ Bits

$\log^2 n = (\log n)^2$



1. Split into $(\log^2 n)$-bit **chunks**

   and store cumulative rank: each needs $\leq \log n$ bits

   $$\Rightarrow O\left(\frac{n}{\log^2 n}\log n\right) = O\left(\frac{n}{\log n}\right) \subseteq o(n) \text{ bits}$$

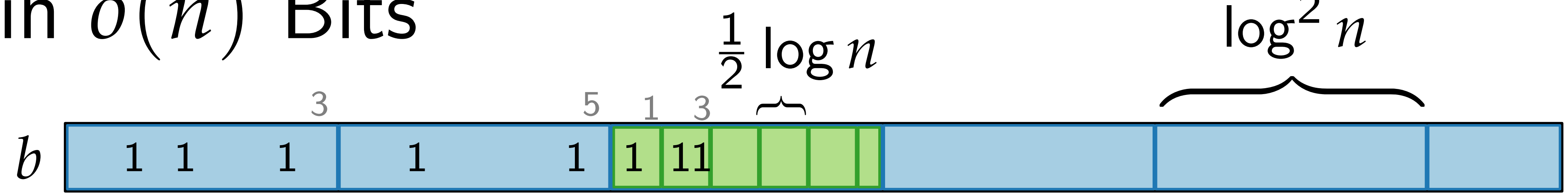
2. Split **chunks** into $(\frac{1}{2}\log n)$-bit **subchunks**

   and store cumulative rank within **chunk**: each needs $\leq \log\log^2 n = 2\log\log n$ bits

   $$\Rightarrow O\Big(\underbrace{\frac{n}{\log n}}_{\#\text{ subchunks}}\ \underbrace{\log\log n}_{\text{rel. rank}}\Big) \subseteq o(n) \text{ bits}$$

Rank in $o(n)$ Bits

$\log^2 n = (\log n)^2$



Example: $n = 64 \Rightarrow \frac{1}{2}\log n = 3$

position →	1	2	3
bitstring 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

1. Split

   and … s $\leq \log n$ bits

   $O(\frac{n}{\log^2 n}\log n) = O(\frac{n}{\log n}) \subseteq o(n)$ bits

2. Split … ks

   and … each needs $\leq \log\log^2 n = 2\log\log n$ bits

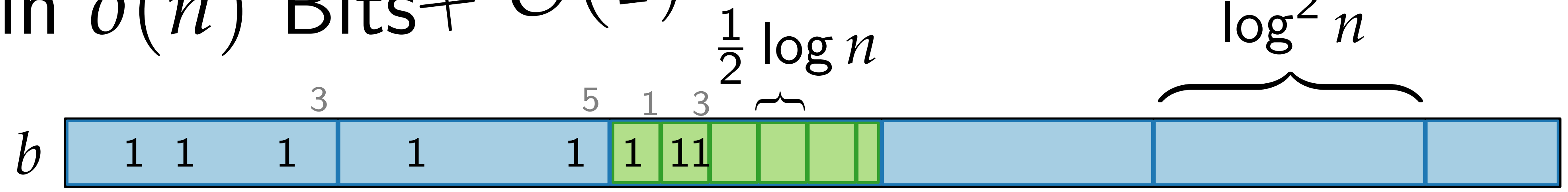
   $\Rightarrow O(\frac{n}{\log n}\log\log n) \subseteq o(n)$ bits

3. Use **lookup table** for bitstrings of length $(\frac{1}{2}\log n)$: $2^{\frac{1}{2}\log n} = \sqrt{n}$ distinct bitstrings

   $\Rightarrow O(\underbrace{\sqrt{n}}_{\#\text{ rows}}\underbrace{\log n}_{\#\text{ columns}}\underbrace{\log\log n}_{\text{rel. rank}}) \subseteq o(n)$ bits

Rank in $o(n)$ Bits+ $O(1)$ Time

$\log^2 n = (\log n)^2$

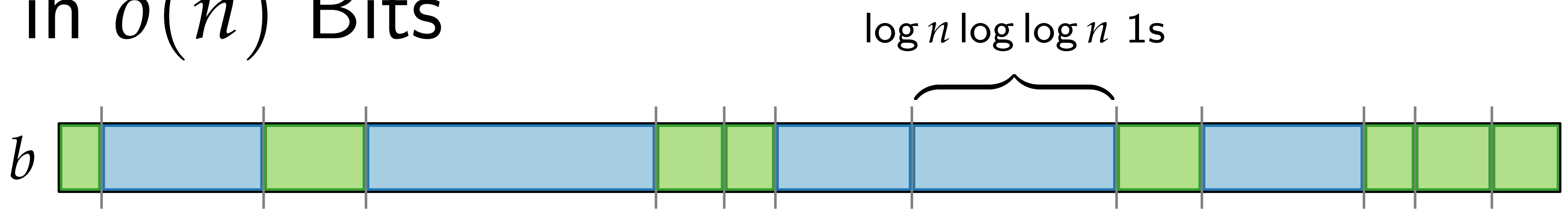


1. Split into $(\log^2 n)$-bit **chunks**
   and store cumulative rank: each needs $\leq \log n$ bits
   $\Rightarrow O(\frac{n}{\log^2 n}\log n) = O(\frac{n}{\log n}) \subseteq o(n)$ bits
2. Split **chunks** into $(\frac{1}{2}\log n)$-bit **subchunks**
   and store cumulative rank within **chunk**: each needs $\leq \log\log^2 n = 2\log\log n$ bits
   $\Rightarrow O(\frac{n}{\log n}\log\log n) \subseteq o(n)$ bits
3. Use **lookup table** for bitstrings of length $(\frac{1}{2}\log n)$: $2^{\frac{1}{2}\log n} = \sqrt{n}$ distinct bitstrings
   $\Rightarrow O(\sqrt{n}\log n\log\log n) \subseteq o(n)$ bits
4. $\texttt{rank}(i) = \texttt{rank}$ of **chunk**
   $+$ relative $\texttt{rank}$ of **subchunk** within **chunk**
   $+$ relative $\texttt{rank}$ of element i within **subchunk**

$\Rightarrow O(1)$ time

(assume read/write numbers in $O(1)$ time)

Select in $o(n)$ Bits



1. Store indices of every $(\log n \log\log n)$-th 1 bit in array

$$\Rightarrow O\left(\frac{n}{\log n \log\log n} \log n\right) = O\left(\frac{n}{\log\log n}\right) \subseteq o(n) \text{ bits}$$

2. Within group of $(\log n \log\log n)$ 1 bits of length r bits:

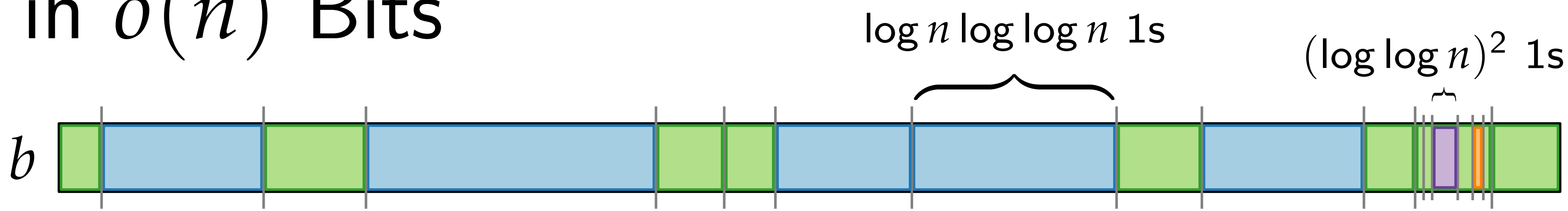
   if $r \geq (\log n \log\log n)^2$
   then store indices of 1 bits in group in array

$$\Rightarrow O\left(\frac{n}{(\log n \log\log n)^2} (\log n \log\log n) \log n\right) \subseteq O\left(\frac{n}{\log\log n}\right) \text{ bits}$$

   else problem is reduced to bitstrings of length $r < (\log n \log\log n)^2$

3. Repeat 1. and 2. on reduced bitstrings

Select in $o(n)$ Bits



3. Repeat 1. and 2. on reduced bitstrings ($r < (\log n \log\log n)^2$):

1' Store relative indices of every $(\log\log n)^2$-th 1 bit in array

$$\Rightarrow O\left(\frac{n}{(\log\log n)^2}\log\log n\right) = O\left(\frac{n}{\log\log n}\right) \text{ bits}$$

2' Within group of $(\log\log n)^2$ 1 bits of length r' bits:

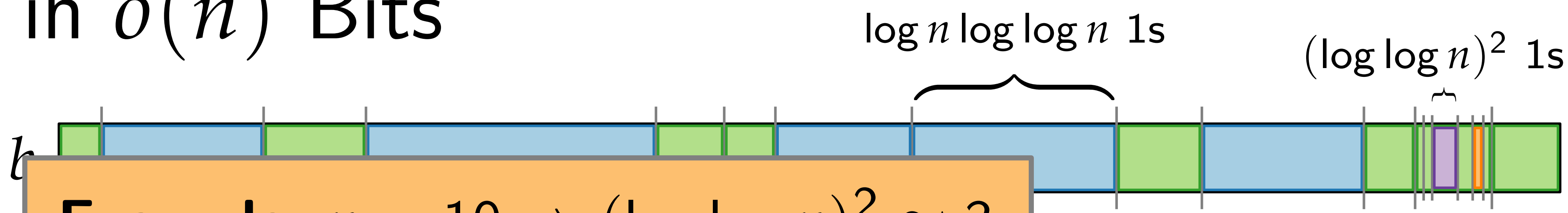
if $r' \geq (\log\log n)^4$

then store relative indices of 1 bits in subgroup in array

$$\Rightarrow O\left(\frac{n}{(\log\log n)^4}(\log\log n)^2\log\log n\right) = O\left(\frac{n}{\log\log n}\right) \text{ bits}$$

else problem is reduced to bitstrings of length $r' < (\log\log n)^4$

Select in $o(n)$ Bits



Example: $n = 10 \Rightarrow (\log\log n)^2 \approx 3$

$\Rightarrow r' < (\log\log n)^4 \approx 9$

bitstring / select →	1	2	3
00000111	6	7	8
00001011	5	7	8
00001101	5	6	8
⋮			
11001000	1	2	5
11010000	1	2	4
11100000	1	2	3

3. Repea… $n \log\log n)^2$):

1' Store… 1 bit in array

… $n)^2$ $\log\log n) = O(\frac{n}{\log\log n})$ bits

2' With… r' bits:

if r'

then … up in array

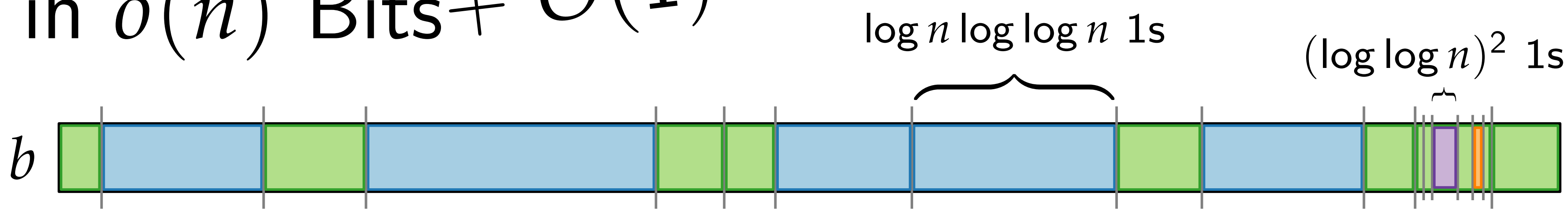
… $n)^2 \log\log n) = O(\frac{n}{\log\log n})$ bits

else problem is reduced to bitstrings of length $r' < (\log\log n)^4$

4. Use **lookup table** for bitstrings of length $r' \leq (\log\log n)^4$:

$\underbrace{2^{(\log\log n)^4}}_{\#\text{ rows}} \in O(2^{\frac{1}{2}\log n}) = O(\sqrt{n})$; $\underbrace{(\log\log n)^2}_{\#\text{ columns}} \in O(\log n) \Rightarrow O(\underbrace{\sqrt{n}}_{\#\text{ rows}}\,\underbrace{\log n}_{\#\text{ columns}}\,\underbrace{\log\log n}_{\text{rel. index}}) = o(n)$ bits

Select in $o(n)$ Bits+ $O(1)$ Time

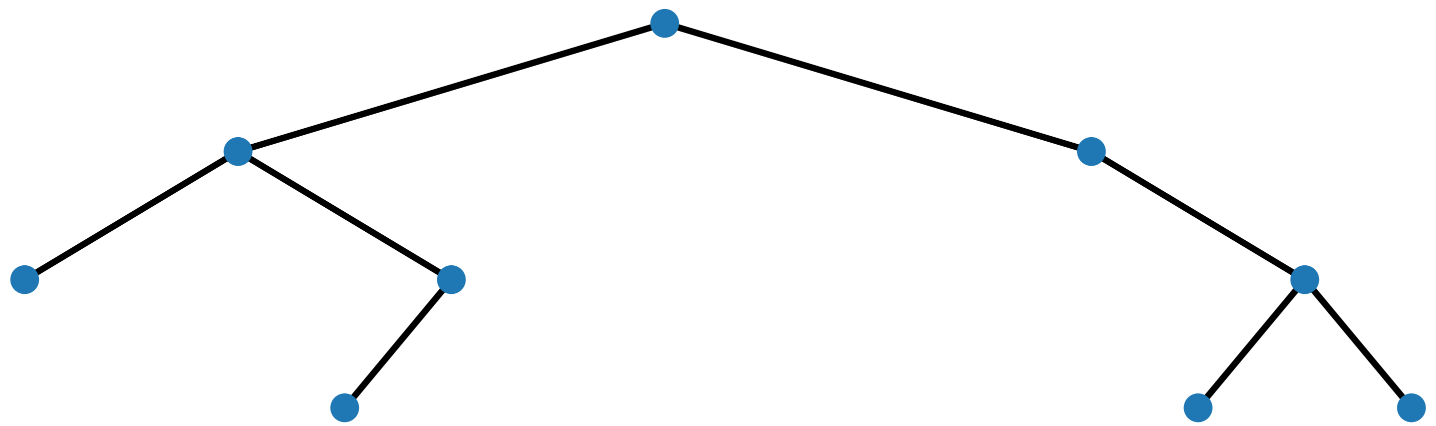


4. $\texttt{select}(j) = \texttt{select}$ J-th group where $J = \lfloor j/(\log n \log\log n) \rfloor$

   + directly $\texttt{select}$ $(j - J)$-th 1 bit
   or $\texttt{select}$ J'-th subgroup where $J' = \lfloor (j - J)/(\log\log n)^2 \rfloor$

   + directly $\texttt{select}$ $(j - J - J')$-th 1 bit
   or $\texttt{select}$ it in the lookup table

Succinct Representation of Binary Trees



C_n is the n-th *Catalan number* and $C_0 = 1$

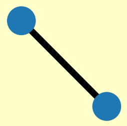
Number of binary trees on n vertices: $C_n = \sum_{i=0}^{n-1} C_i \cdot C_{n-1-i} = \frac{(2n)!}{(n+1)!\,n!}$

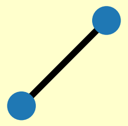
$n = 0$: "empty tree"

1 possibility

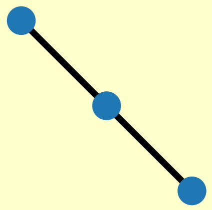
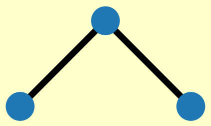
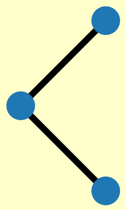
$n = 1$: 

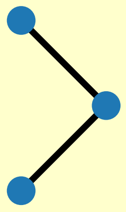
1 possibility

$n = 2$: 

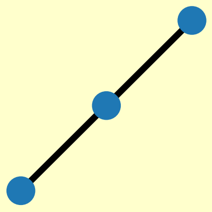


2 possibilities

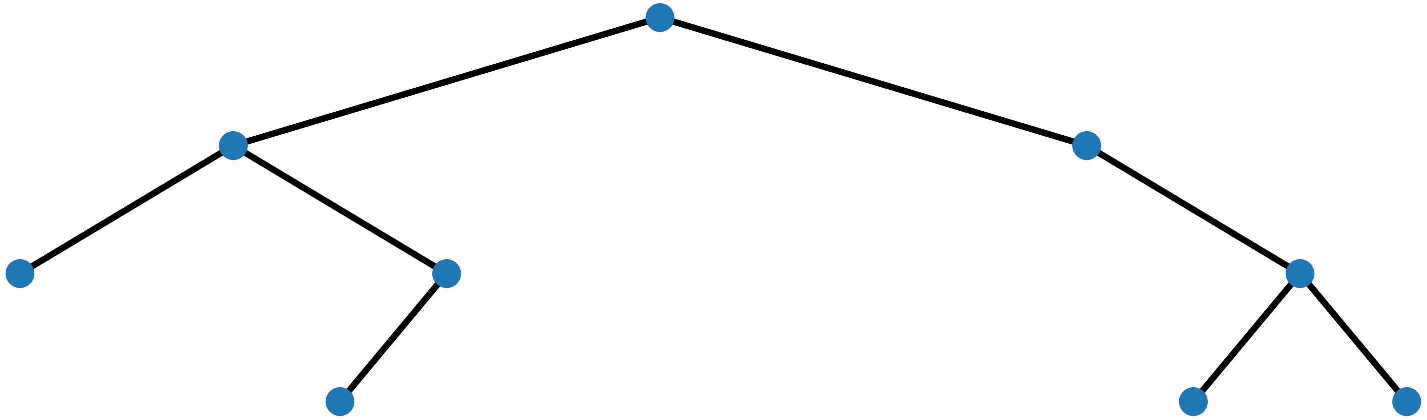
$n = 3$:   



5 possibilities



Succinct Representation of Binary Trees



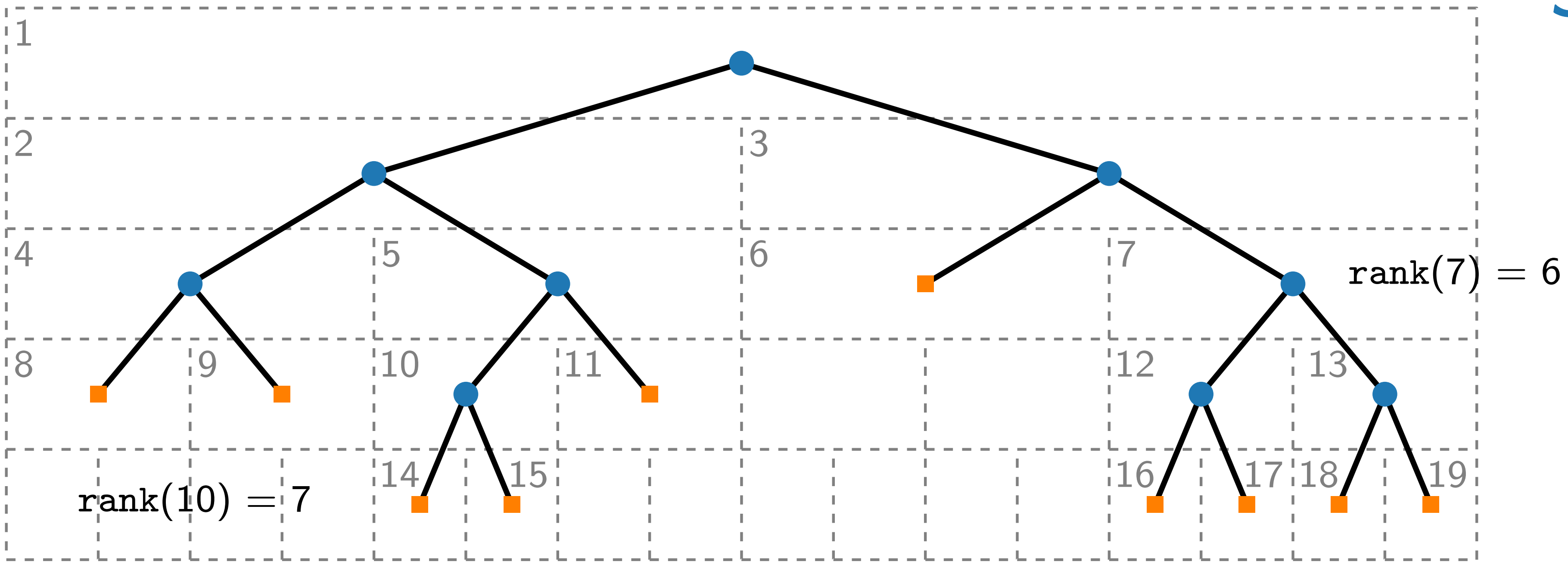
Number of binary trees on n vertices: $C_n = \sum_{i=0}^{n-1} C_i \cdot C_{n-1-i} = \frac{(2n)!}{(n+1)!\,n!}$

$$\log C_n = 2n + o(n) \text{ (by Stirling's approximation)}$$

$\Rightarrow$ We can use $2n + o(n)$ bits to represent binary trees.

Difficulty is when a binary tree is not full.

Succinct Representation of Binary Trees



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	1	1	1	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0

Size.

- $2n+1$ bits for b
- $o(n)$ for `rank` and `select`

Proof is exercise.

Idea.

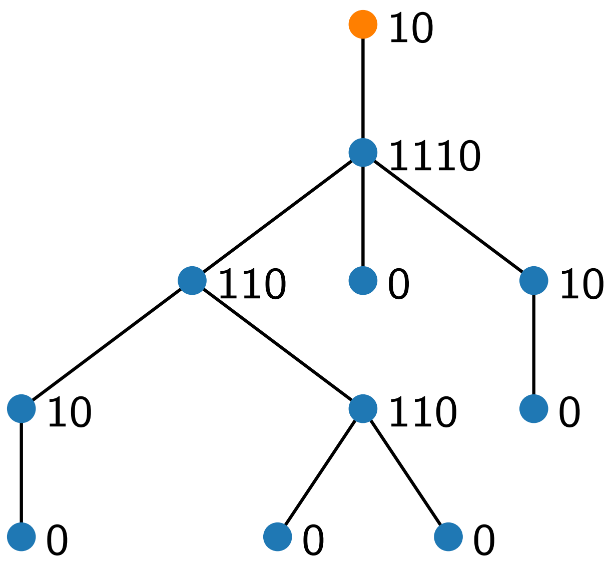
- Add external nodes to have out-degree 2 or 0 at every node
- Read internal nodes as 1
- Read external nodes as 0
- Use `rank` and `select`

Operations.

- $\texttt{parent}(i) = \texttt{select}(\lfloor \frac{i}{2} \rfloor)$
- $\texttt{leftChild}(i) = 2\,\texttt{rank}(i)$
- $\texttt{rightChild}(i) = 2\,\texttt{rank}(i) + 1$
- $\texttt{rank}(i)$ is index for array storing actual values

Succinct Representation of Trees - LOUDS

[Level Order Unary Degree Sequence]



- add extra root with out-degree 1
- unary encoding of out-degree terminated by a 0
- gives LOUDS sequence

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	1	1	1	0	1	1	0	0	1	0	1	0	1	1	0	0	0	0	0

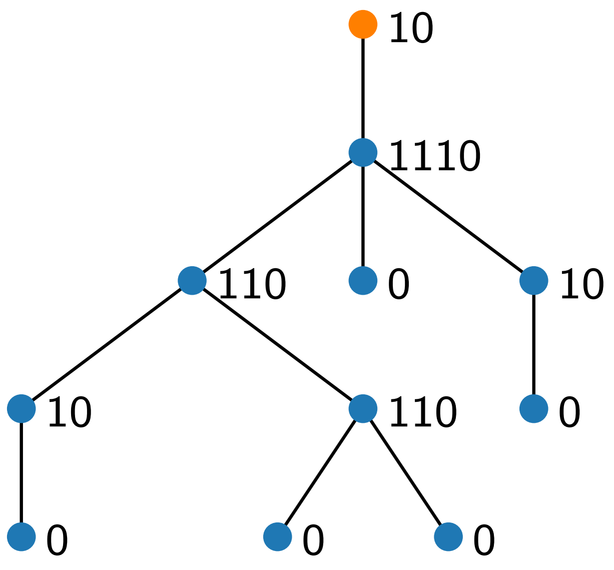
Size.

- each vertex (except root) is represented twice, namely with a 1 and with a 0
- $o(n)$ bits for `rank` and `select`

$\Rightarrow 2n + o(n)$ bits

Succinct Representation of Trees - LOUDS

[Level Order Unary Degree Sequence]



- add extra root with out-degree 1
- unary encoding of out-degree terminated by a 0
- gives LOUDS sequence

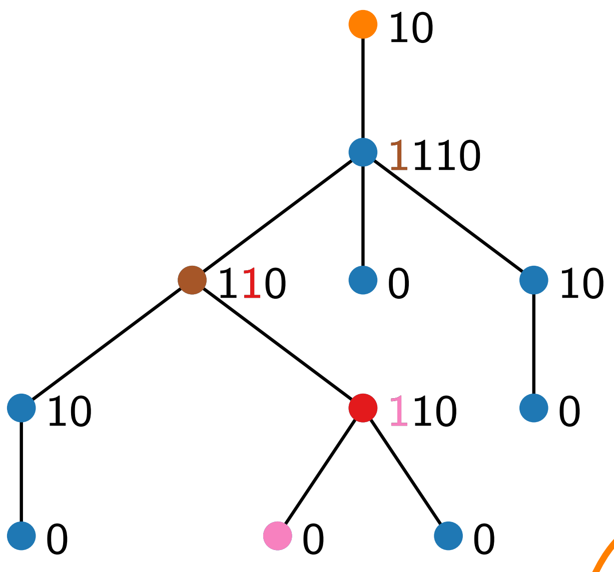
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	1	1	1	0	1	1	0	0	1	0	1	0	1	1	0	0	0	0	0

Operations.

- Let i be index of 1 in LOUDS sequence. This 1 represents a node (e.g. first 1 represents the root).
- $\texttt{rank}(i)$ is index for array storing actual values of the nodes.

Succinct Representation of Trees - LOUDS

[Level Order Unary Degree Sequence]



- add extra root with out-degree 1
- unary encoding of out-degree terminated by a 0
- gives LOUDS sequence

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	1	1	1	0	1	1	0	0	1	0	1	0	1	1	0	0	0	0	0
1	0	1	1	1	0	1	1	0	0	1	0	1	0	1	1	0	0	0	0	0

execute $\mathtt{select}(j)$ on the 0s instead of the 1s

execute $\mathtt{rank}(i)$ on the 1s (as before)

- $\mathtt{firstChild}(i) = \mathtt{select}_0(\mathtt{rank}_1(i)) + 1$

  $\mathtt{firstChild}(8) = \mathtt{select}_0(\mathtt{rank}_1(8)) + 1$
  $= \mathtt{select}_0(6) + 1 = 14 + 1 = 15$

- $\mathtt{parent}(i) = \mathtt{select}_1(\mathtt{rank}_0(i))$

  $\mathtt{parent}(8) = \mathtt{select}_1(\mathtt{rank}_0(8))$
  $= \mathtt{select}_1(2) = 3$

- $\mathtt{nextSibling}(i) = i + 1$

Exercise: $\mathtt{child}(i, j)$ with validity check

Discussion

- Succinct data structures are
  - space efficient
  - support fast operations

  but
  - are mostly static (dynamic at extra cost), that means insertions & deletions
  - number of operations is limited,
  - complex $\rightarrow$ harder to implement,
  - the $o(n)$ and $O(1)$ term hide constants that might dominate before any asymptotic advantage over the "best" compact data structures becomes apparent.
    $\rightarrow$ primarily a theoretical result (also does not consider hardware architecture)
- `rank` and `select` form the basis for many succinct representations (e.g., for specific types of trees or strings).
- There are implementations of succinct data structures being used in practice for large data sets in information retrieval, language model representation, bioinformatics, etc.

Literature

Main reference:

- Lecture 17 of Advanced Data Structures (MIT, Fall'17) by Erik Demaine
- [Jac '89] "Space efficient Static Trees and Graphs"

Recommendations:

- Lecture 18 of Demaine's course on compact & succinct arrays & trees