Advanced Data Structures

Lecture 02: Succinct Trees

Florian Kurpicz
Recap: Rank Queries on Bit Vectors (1/2)

<table>
<thead>
<tr>
<th>α</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank_{\alpha}(i)</td>
<td># of \alpha s before i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>select_{\alpha}(j)</td>
<td>position of j-th \alpha</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
0 1 1 0 1 1 0 1 0 0
```
Recap: Rank Queries on Bit Vectors (1/2)

**rank**$_{\alpha}$(i) # of $\alpha$s before i
select$_{\alpha}$(j) position of j-th $\alpha$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

rank$_{0}$(5)
Recap: Rank Queries on Bit Vectors (1/2)

\[ \text{rank}_\alpha(i) \] # of \( \alpha \)s before \( i \)
\[ \text{select}_\alpha(j) \] position of \( j \)-th \( \alpha \)

---

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Recap: Rank Queries on Bit Vectors (1/2)

- \( \text{rank}_\alpha(i) \) # of \( \alpha \)s before \( i \)
- \( \text{select}_\alpha(j) \) position of \( j \)-th \( \alpha \)

\[
\begin{array}{cccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Recap: Rank Queries on Bit Vectors (1/2)

\( \text{rank}_\alpha(i) \) \# of \( \alpha \)s before \( i \)

\( \text{select}_\alpha(j) \) position of \( j \)-th \( \alpha \)

\( \text{rank}_0(5) \)

\( \text{select}_1(5) \)

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Recap: Rank Queries on Bit Vectors (1/2)

\[ \text{rank}_\alpha(i) \] \# of \( \alpha \)s before \( i \)
\[ \text{select}_\alpha(j) \] position of \( j \)-th \( \alpha \)

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Recap: Rank Queries on Bit Vectors (1/2)

\( \text{rank}_\alpha(i) \) # of \( \alpha \)s before \( i \)

\( \text{select}_\alpha(j) \) position of \( j \)-th \( \alpha \)

\( \text{rank}_0(5) \)

2

0 1 1 0 1 1 0 1 0 0
Recap: Rank Queries on Bit Vectors (1/2)

\( \text{rank}_\alpha(i) \) # of \( \alpha \)s before \( i \)

\( \text{select}_\alpha(j) \) position of \( j\)-th \( \alpha \)

\( \text{rank}_0(5) \)

\begin{array}{cccccccccc}
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}

# of 0s w.r.t. BV
Recap: Rank Queries on Bit Vectors (1/2)

\[
\text{rank}_\alpha(i) \quad \# \text{ of } \alpha \text{s before } i \\
\text{select}_\alpha(j) \quad \text{position of } j\text{-th } \alpha
\]

```plaintext
0 1 2 3 4 5 6 7 8 9
0 1 1 0 1 1 0 1 0 0
```

- \(\text{rank}_0(5)\)
- \(\# \text{ of 0s w.r.t. super-block}\)
- \(\# \text{ of 0s w.r.t. BV}\)
Recap: Rank Queries on Bit Vectors (1/2)

- $\text{rank}_\alpha(i)$: Number of $\alpha$s before $i$
- $\text{select}_\alpha(j)$: Position of $j$-th $\alpha$

\[
\begin{array}{cccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]

2

$\text{rank}_0(5)$

# of 0s w.r.t. BV

# of 0s w.r.t. super-block

block

super-block
Recap: Rank Queries on Bit Vectors (2/2)

Lemma: Binary Rank- and Select-Queries

Given a bit vector of size $n$, there exist data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time.
Lemma: Binary Rank- and Select-Queries

Given a bit vector of size $n$, there exist data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time.

Word RAM

- unlimited memory
- words of size $w = \Theta \log n$
- constant time load and store
- constant time bit operations on words
Plan for Today

- represent tree with $n$ nodes using $2n$ bits
- make succinct tree fully-functional using additional $o(n)$ bits
Plan for Today

- represent tree with $n$ nodes using $2n$ bits
- make succinct tree fully-functional using additional $o(n)$ bits

- trees are important
  - searching for keys
  - maintaining directories
  - representations of parsings
  - ...

![Tree Diagram]
Plan for Today

- represent tree with $n$ nodes using $2n$ bits
- make succinct tree fully-functional using additional $o(n)$ bits

- trees are important
  - searching for keys
  - maintaining directories
  - representations of parsings
  - . . .

- different representations
- supporting different operations
Plan for Today

- represent tree with $n$ nodes using $2n$ bits
- make succinct tree fully-functional using additional $o(n)$ bits

- trees are important
  - searching for keys
  - maintaining directories
  - representations of parsings
  - . . .

- different representations
- supporting different operations

Handout
Preliminaries

- a tree is an acyclic connected graph \( G = (V, E) \) with a root \( r \in V \)
- degree \( \delta \) is the number of children
- leaves have degree 0
- depth of a node is the length of the path from the root to that node
Level Ordered Unary Degree Sequence (1/2) [Jac88]

- represent tree level-wise
- use ≤ 2 bits per node

Definition: LOUDS

Starting at the root, all nodes on the same depth are visited from left to right and for node \(v\), \(\delta(v)\) 1's followed by a 0 are appended to the bit vector that contains an initial 10.

Lemma: Space Usage of LOUDS

Representing a tree with \(n\) nodes requires \(2^n + 1\) bits using LOUDS.

```
write down the LOUDS representation of this example tree
```
Level Ordered Unary Degree Sequence (1/2) [Jac88]

- represent tree level-wise
- use $\leq 2$ bits per node

**Definition: LOUDS**

Starting at the root, all nodes on the same depth
- are visited from left to right and
- for node $v$, $\delta(v)$ 1’s followed by a 0 are appended to the bit vector that contains an initial 10
**Level Ordered Unary Degree Sequence (1/2) [Jac88]**

- represent tree level-wise
- use \( \leq 2 \) bits per node

**Definition: LOUDS**

Starting at the root, all nodes on the same depth
- are visited from left to right and
- for node \( v \), \( \delta(v) \) 1’s followed by a 0 are appended to the bit vector that contains an initial 10

**Lemma: Space Usage of LOUDS**

Representing a tree with \( n \) nodes requires \( 2n + 1 \) bits using LOUDS
Level Ordered Unary Degree Sequence (1/2) [Jac88]

- represent tree level-wise
- use $\leq 2$ bits per node

**Definition: LOUDS**

Starting at the root, all nodes on the same depth
- are visited from left to right and
- for node $v$, $\delta(v)$ 1’s followed by a 0 are appended to the bit vector that contains an initial 10

**Lemma: Space Usage of LOUDS**

Representing a tree with $n$ nodes requires $2n + 1$ bits using LOUDS

- write down the LOUDS representation of this example tree
Level Ordered Unary Degree Sequence (2/2)

1011110
Level Ordered Unary Degree Sequence (2/2)

10111001100110

```
    a
   / \  /  /
  b   c d  e
   \  /\  /
    h i f g
    j  k
```
Level Ordered Unary Degree Sequence (2/2)

```
10111001100110011000
```

![Diagram of a level ordered unary degree sequence](image)
Level Ordered Unary Degree Sequence (2/2)

```
10111100110011001100000
```

![Graph of Level Ordered Unary Degree Sequence]
Level Ordered Unary Degree Sequence (2/2)

- node start at pertinent 0

```
ab ch id ejkfg
1011100110011001100000
```
What is Fully-Functional?

Operations
- degree \( i \) is leaf
- \( i \)-th child
- parent
- subtree size
What is Fully-Functional?

Operations

- degree 1 is leaf
- i-th child
- parent
- subtree size

- depth
- lowest common ancestor
- rank (pre- or post-order)
- ...

```
  a
  /   \
 b     c
  \\   /   \       \
 d   e   h     i
     \\    \  \    \  \
      f g   j k
```
Making LOUDS Fully-Functional

- degree of $p$: $p - \text{select}_0(\text{rank}_0(p)) - 1$

- explanation on the board 📚
Making LOUDS Fully-Functional

- degree of \( p \): \( p - \text{select}_0(\text{rank}_0(p)) - 1 \)
- \( i \)-th child of \( p \):
  \[\text{select}_0(\text{rank}_1(\text{select}_0(\text{rank}_0(p)))) + i + 1\]

- explanation on the board 📚
Making LOUDS Fully-Functional

- degree of $p$: $p - select_0(rank_0(p)) - 1$
- $i$-th child of $p$: $select_0(rank_1(select_0(rank_0(p)))) + i + 1$
- parent of $p$: $select_0(rank_0(select_1(rank_0(p)))) + 1$

- explanation on the board 🎨
Making LOUDS Fully-Functional

- degree of $p$: $p - \text{select}_0(\text{rank}_0(p)) - 1$
- $i$-th child of $p$:
  \[ \text{select}_0(\text{rank}_1(\text{select}_0(\text{rank}_0(p)))) + i + 1 \]
- parent of $p$:
  \[ \text{select}_0(\text{rank}_0(\text{select}_1(\text{rank}_0(p)))) + 1 \]

- explanation on the board

- subtree size

---

ab ch id ejkfg
1011100110011001100000
From Bit Vectors to Parentheses

- instead of 0 and 1
- use ( and )

- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right.
Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right.
From Bit Vectors to Parentheses

- instead of 0 and 1
- use ( and )

- requires the same space
- can add relation between parentheses

**Definition: Balanced String of Parentheses**

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right.

- `findclose(i)`: find the right parenthesis matching the left parenthesis at position `i`
### From Bit Vectors to Parentheses

- instead of 0 and 1
- use ( and )
- requires the same space
- can add relation between parentheses

---

**Definition: Balanced String of Parentheses**

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right.

- $\text{findclose}(i)$: find the right parenthesis matching the left parenthesis at position $i$
- $\text{findopen}(i)$: find the left parenthesis matching the right parenthesis at position $i$
From Bit Vectors to Parentheses

- instead of 0 and 1
- use ( and )
- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right.

- findclose\(i\): find the right parenthesis matching the left parenthesis at position \(i\)
- findopen\(i\): find the left parenthesis matching the right parenthesis at position \(i\)
- excess\(i\): find the difference between the number of left and right parentheses before position \(i\)
instead of 0 and 1
use ( and )

requires the same space
can add relation between parentheses

Definition: Balanced String of Parentheses
A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right

findclose(i): find the right parenthesis matching the left parenthesis at position i
findopen(i): find the left parenthesis matching the right parenthesis at position i
excess(i): find the difference between the number of left and right parentheses before position i
enclose(i): given a parentheses pair with the left parenthesis at position i, return the position of the closest left parenthesis belonging to the parentheses pair enclosing it
Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right.

- \( \text{findclose}(i) \): find the right parenthesis matching the left parenthesis at position \( i \)
- \( \text{findopen}(i) \): find the left parenthesis matching the right parenthesis at position \( i \)
- \( \text{excess}(i) \): find the difference between the number of left and right parentheses before position \( i \)
- \( \text{enclose}(i) \): given a parentheses pair with the left parenthesis at position \( i \), return the position of the closest left parenthesis belonging to the parentheses pair enclosing it

- instead of 0 and 1
- use ( and )
- requires the same space
- can add relation between parentheses

- how can we answer excess queries
all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space

here, a little bit simpler
all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space

due to a little bit simpler

- $\text{excess}(i) = \text{rank}_{\cdot}(i) - \text{rank}_{\cdot}(i)$
- $\text{fwd}_\text{search}(i, d) = \min\{j > i: \text{excess}(j) - \text{excess}(i - 1) = d\}$
- $\text{bwd}_\text{search}(i, d) = \max\{j < i: \text{excess}(i) - \text{excess}(j - 1) = d\}$
From Bit Vectors to Parentheses

- all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space
- here, a little bit simpler

- $\text{excess}(i) = \text{rank}^{-1}(i) - \text{rank}^{-1}(i-1)$
- $\text{fwd\_search}(i, d) = \min\{j > i : \text{excess}(j) - \text{excess}(i-1) = d\}$
- $\text{bwd\_search}(i, d) = \max\{j < i : \text{excess}(i) - \text{excess}(j-1) = d\}$

- $\text{findclose}(i) = \text{fwd\_search}(i, 0)$
- $\text{findopen}(i) = \text{bwd\_search}(i, 0)$
- $\text{enclose}(i) = \text{bwd\_search}(i, 2)$
all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space

here, a little bit simpler

excess($i$) = rank($i$) − rank($i−1$)

fwd_search($i, d$) = min{$j > i$: excess($j$) − excess($i−1$) = $d$}

bwd_search($i, d$) = max{$j < i$: excess($i$) − excess($j−1$) = $d$}

findclose($i$) = fwd_search($i, 0$)

findopen($i$) = bwd_search($i, 0$)

enclose($i$) = bwd_search($i, 2$)

can be answered with a min-max-tree later in this course
Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses

Definition: BP

Starting at the root, traverse the tree in depth-first order and append a left parenthesis if a node is visited the first time, right parenthesis if a node is visited the last time to the bit vector.

Lemma: Space Usage of BP

Representing a tree with \( n \) nodes requires \( 2^n \) bits using BP.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
</table>
Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses

Definition: BP

Starting at the root, traverse the tree in depth-first order and append a

- left parenthesis if a node is visited the first time
- right parenthesis if a node is visited the last time
to the bit vector
Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses

Definition: BP

Starting at the root, traverse the tree in depth-first order and append a
- left parenthesis if a node is visited the first time
- right parenthesis if a node is visited the last time
to the bit vector

Lemma: Space Usage of BP

Representing a tree with \( n \) nodes requires \( 2n \) bits using BP
Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses

**Definition: BP**

Starting at the root, traverse the tree in depth-first order and append a
- left parenthesis if a node is visited the first time
- right parenthesis if a node is visited the last time

to the bit vector

**Lemma: Space Usage of BP**

Representing a tree with $n$ nodes requires $2n$ bits using BP

- write down the BP representation of this example tree
Balanced Parentheses (2/2)

```
a
(  
```
Balanced Parentheses (2/2)

ab
(()

node starts at first parenthesis
subtree structure is encoded in parentheses
Balanced Parentheses (2/2)

ab cd
(())(())

Node starts at first parenthesis. Subtree structure is encoded in parentheses.
Balanced Parentheses (2/2)

ab cd ef g

(()(()(()())))

[Diagram of a balanced parentheses structure with nodes and edges labeled with letters A to K.]

node starts at first parenthesis
subtree structure is encoded in parentheses
Balanced Parentheses (2/2)

ab cd ef g h

(((())(((())()))))()
Balanced Parentheses (2/2)

ab cd ef g h i j k

(((())())())((()))(())

\[
\begin{array}{c}
\text{Diagram:} \\
\quad a \\
\quad \quad b \quad c \quad h \quad i \\
\quad \quad \quad d \quad e \quad j \quad k \\
\quad \quad \quad \quad f \quad g
\end{array}
\]
Balanced Parentheses (2/2)

- node starts at first parenthesis
- subtree structure is encoded in parentheses

\[
\text{ab cd ef g h ij k}
\]

\[
(()(()(()(()))))(()()))
\]
Making BP Fully-Functional

- subtree size of $p$: $\frac{\text{findclose}(p) - p + 1}{2}$

- explanation on the board

```
ab cd ef g h ij k
(()(()(()()))()(()()))
```
Making BP Fully-Functional

- Subtree size of $p$: $\frac{(\text{findclose}(p) - p + 1)}{2}$
- Parent of $p$: $\text{enclose}(p)$

Explanation on the board
Making BP Fully-Functional

- subtree size of $p$: $(\text{findclose}(p) - p + 1)/2$
- parent of $p$: $\text{enclose}(p)$

Complicated Constant Time [NS14]
- degree
- $i$-th child

Diagram:
```
         a
        /|
        / |
       b  c
       /|
       / |
      d  e
      /|
      / |
     f  g
     /|
     / |
    h  i
    /|
    / |
   j  k
```

- explanation on the board
Advantages and Disadvantages of Both Approaches

- LOUDS cannot answer subtree size
- BP cannot easily answer $i$-th child and degree
- all other operations can be done easily
Definition: DFUDS

Starting at the root, traverse tree in depth-first order and append

- for node $v$, $\delta(v)$ left parentheses and
- a right parenthesis if $v$ is visited the first time
to the bit vector that initially contains a left parenthesis $\delta$ to make them balanced.

Lemma: Space Usage of DFUDS

Representing a tree with $n$ nodes requires $2^n$ bits using DFUDS.
Definition: DFUDS
Starting at the root, traverse tree in depth-first order and append
- for node $v$, $\delta(v)$ left parentheses and
- a right parenthesis if $v$ is visited the first time
to the bit vector that initially contains a left parenthesis $\delta$ to make them balanced

Lemma: Space Usage of DFUDS
Representing a tree with $n$ nodes requires $2n$ bits using DFUDS
Definition: DFUDS

Starting at the root, traverse tree in depth-first order and append
- for node $v$, $\delta(v)$ left parentheses and
- a right parenthesis if $v$ is visited the first time
to the bit vector that initially contains a left parenthesis to make them balanced.

Lemma: Space Usage of DFUDS

Representing a tree with $n$ nodes requires $2n$ bits using DFUDS.

Write down the DFUDS representation of this example tree:

```
| a -- b
|    c -- d
|     e
| h -- f
i -- g
j
k
```
Depth First Unary Degree Sequence (2/2)

```
a  ((((()
```

Diagram:
```
node starts at first parenthesis
subtree structure is encoded
```
Depth First Unary Degree Sequence (2/2)

\[
\begin{array}{ll}
\text{a} & \text{b} \\
((()()())
\end{array}
\]
Depth First Unary Degree Sequence (2/2)

```
(a b c
  (((())()))
```

![Tree Diagram]
Depth First Unary Degree Sequence (2/2)

\[
\begin{array}{cccc}
  a & b & c & d \\
  (((())))(())
\end{array}
\]
Depth First Unary Degree Sequence (2/2)

a b c d e f g
(((())(())(()))())

```
node starts at first parenthesis
subtree structure is encoded
/chalkboard-◎eacher
```
Depth First Unary Degree Sequence (2/2)

\[
a \ bc \ de \ fgh \\
(((((()))(())(())))
\]
Depth First Unary Degree Sequence (2/2)

\[ (a \ b \ c \ d \ e \ f \ g \ h \ i \ j \ k) \]

```
node starts at first parenthesis
subtree structure is encoded
```

```
18/20
2022-05-02 Florian Kurpicz | Advanced Data Structures | 02 Succinct Trees
Institute of Theoretical Informatics, Algorithm Engineering
```

```
18/20
2022-05-02 Florian Kurpicz | Advanced Data Structures | 02 Succinct Trees
Institute of Theoretical Informatics, Algorithm Engineering
```
Depth First Unary Degree Sequence (2/2)

- node starts at first parenthesis
- subtree structure is encoded

```
(((((()))((()))))))
```

```
node starts at first parenthesis
subtree structure is encoded
```
Making DFUDS Fully-Functional

- degree of \( p \): \( \text{select}^{-1} (\text{rank}^{-1} (p) + 1) - p \)

- explanation on the board 📚
Making DFUDS Fully-Functional

- degree of $p$: $\text{select}^{-1}(\text{rank}^{-1}(p) + 1) - p$
- $i$-th child of $p$:
  $\text{findclose}(\text{select}^{-1}(\text{rank}^{-1}(p) + 1) - i) + 1$

- explanation on the board 📝
Making DFUDS Fully-Functional

- degree of $p$: $\text{select}^{-1}(\text{rank}^{-1}(p) + 1) - p$
- $i$-th child of $p$: $\text{findclose}(\text{select}^{-1}(\text{rank}^{-1}(p) + 1) - i) + 1$
- parent of $p$: $\text{select}^{-1}(\text{rank}^{-1}(\text{findopen}(p-1)))+1$

- explanation on the board 📚
Making DFUDS Fully-Functional

- degree of $p$: $\text{select}(\text{rank}(p) + 1) - p$
- $i$-th child of $p$: $\text{findclose}(\text{select}(\text{rank}(p) + 1) - i) + 1$
- parent of $p$: $\text{select}(\text{rank}(\text{findopen}(p - 1))) + 1$
- subtree size of $p$: $(\text{findclose}(\text{enclose}(p)) - p)/2 + 1$

Explanation on the board 📜
Conclusion and Outlook

This Lecture
- three succinct tree representations
- different advantages and disadvantages

Advanced Data Structures

BV  succ. trees
Conclusion and Outlook

This Lecture
- three succinct tree representations
- different advantages and disadvantages
- outlook to min-max-trees
Conclusion and Outlook

This Lecture
- three succinct tree representations
- different advantages and disadvantages
- outlook to min-max-trees

Next Lecture
- dynamic bit vectors and succinct trees
- maybe succinct graphs

Advanced Data Structures
- BV
- succ. trees
<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Location</th>
<th>Year</th>
</tr>
</thead>
</table>