Recap: Rank Queries on Bit Vectors

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

### Example

- $\text{rank}_0(5) = 2$:
  - # of 0s w.r.t. BV
- $\text{select}_1(5)$:
  - Position of 5-th 1

### Example Bit Vector

- 0 1 1 0 1 1 0 1 0 0

### Block and Super-block

- Block
- Super-block

### Additional Notes

- # of 0s w.r.t. super-block
- # of 0s w.r.t. BV
Recap: Succinct Trees

![Succinct Tree Diagram]

LOUDS

```
ab ch id ejkfg
1011100110011001100000
```
Recap: Succinct Trees

![Succinct Tree Diagram]

**LOUDS**

\[
\text{ab ch id ejkfg} \\
1011100110011001100000
\]

**BP**

\[
\text{ab cd ef g h ij k} \\
(()(()(()()))()(()()))
\]
Recap: Succinct Trees

**LOUDS**

ab ch id ejkfg
10111100110011001100000

**BP**

ab cd ef g h ij k
((((((((()))))))))))))))

**DFUDS**

a bc de fghi jk
(((((((())))))))))))))))})
What is a Dynamic Bit Vector?

Dynamic Bit Vector Operations

- *insert(BV, i, b)* inserts *b* between *BV[i−1]* and *BV[i]*
- *delete(BV, i)* deletes *BV[i]*
- *bitset(BV, i)* sets *B[i] = 1*
- *bitclear(BV, i)* sets *B[i] = 0*
What is a Dynamic Bit Vector?

### Dynamic Bit Vector Operations

- **insert**(BV, i, b) inserts b between BV[i − 1] and BV[i]
- **delete**(BV, i) deletes BV[i]
- **bitset**(BV, i) sets B[i] = 1
- **bitclear**(BV, i) sets B[i] = 0

- **bitset** and **bitclear** easy without rank and select
- **insert** and **delete** require more work
What is a Dynamic Bit Vector?

**Dynamic Bit Vector Operations**

- $insert(BV, i, b)$ inserts $b$ between $BV[i - 1]$ and $BV[i]$
- $delete(BV, i)$ deletes $BV[i]$
- $bitset(BV, i)$ sets $B[i] = 1$
- $bitclear(BV, i)$ sets $B[i] = 0$

- $bitset$ and $bitclear$ easy without rank and select
- $insert$ and $delete$ require more work

```
10011010001111
01001101001111
```
What is a Dynamic Bit Vector?

### Dynamic Bit Vector Operations

- **insert** \((BV, i, b)\) inserts \(b\) between \(BV[i - 1]\) and \(BV[i]\)
- **delete** \((BV, i)\) deletes \(BV[i]\)
- **bitset** \((BV, i)\) sets \(B[i] = 1\)
- **bitclear** \((BV, i)\) sets \(B[i] = 0\)

- **bitset** and **bitclear** easy without rank and select
- **insert** and **delete** require more work

---

**what update time do we want to have?**

- \(O(n)\)
- \(O(\log n)\)
- \(O(1)\)
What is a Dynamic Bit Vector?

Dynamic Bit Vector Operations

- `insert(BV, i, b)` inserts `b` between `BV[i − 1]` and `BV[i]`
- `delete(BV, i)` deletes `BV[i]`
- `bitset(BV, i)` sets `B[i] = 1`
- `bitclear(BV, i)` sets `B[i] = 0`

- `bitset` and `bitclear` easy without rank and select
- `insert` and `delete` require more work

- what update time do we want to have?
  - `O(n)`
  - `O(log n)`
  - `O(1)`

- is doubling the length sufficient ⌊ amortized analysis ⌋ PINGO

- 10011010001111
- 01001101001111

PINGO
What is a Dynamic Bit Vector?

Dynamic Bit Vector Operations

- `insert(BV, i, b)` inserts `b` between `BV[i - 1]` and `BV[i]`
- `delete(BV, i)` deletes `BV[i]`
- `bitset(BV, i)` sets `B[i] = 1`
- `bitclear(BV, i)` sets `B[i] = 0`

- `bitset` and `bitclear` easy without rank and select
- `insert` and `delete` require more work

- what update time do we want to have?
  - `O(n)`
  - `O(log n)`
  - `O(1)`

- is doubling the length sufficient? [amortized analysis](https://example.com)

- why not using a linked list? [PINGO](https://example.com)

```
10011010001111
01001101001111
```
What is a Dynamic Bit Vector?

**Dynamic Bit Vector Operations**

- `insert(BV, i, b)` inserts `b` between `BV[i − 1]` and `BV[i]`
- `delete(BV, i)` deletes `BV[i]`
- `bitset(BV, i)` sets `B[i] = 1`
- `bitclear(BV, i)` sets `B[i] = 0`

- `bitset` and `bitclear` easy without rank and select
- `insert` and `delete` require more work

- what update time do we want to have?
  - `O(n)`
  - `O(log n)`
  - `O(1)`

- is doubling the length sufficient ⌊ amortized analysis ⌋

- why not using a linked list?

Next

- dynamic bit vector including rank and select
Practical Dynamic Bit Vectors (1/2) [Nav16]

- for dynamic bit vector of size $n$
- use slowdown factor $O(w)$
- if $n$ is large, $O(w)$ becomes similar to $O(\log n)$
for dynamic bit vector of size $n$

- use slowdown factor $O(w)$
- if $n$ is large, $O(w)$ becomes similar to $O(\log n)$

- query time $O(w)$
- $n + O(n/w)$ bits of space
- trade off between query time and space
Practical Dynamic Bit Vectors (1/2) [Nav16]

- for dynamic bit vector of size $n$
- use slowdown factor $O(w)$
- if $n$ is large, $O(w)$ becomes similar to $O(\log n)$

- query time $O(w)$
- $n + O(n/w)$ bits of space
- trade off between query time and space

- use pointer-based balanced search tree
- leaves store pointer to $\Theta(w^2)$ bits
- inner nodes store total number of bits ($num$) and number of ones ($ones$) in left subtree

$BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011$
for dynamic bit vector of size $n$

- use slowdown factor $O(w)$
- if $n$ is large, $O(w)$ becomes similar to $O(\log n)$

query time $O(w)$

- $n + O(n/w)$ bits of space
- trade off between query time and space

use pointer-based balanced search tree

- leaves store pointer to $\Theta(w^2)$ bits
- inner nodes store total number of bits ($num$) and number of ones ($ones$) in left subtree

Practical Dynamic Bit Vectors (1/2) [Nav16]
Lemma: Practical Dynamic Bit Vectors

Space

The dynamic bit vector requires $n + O(n/w)$ bits of space.

Proof

$\Theta(w^2)$ bits per leaf

$O(n/w^2)$ nodes

Each (inner) node stores 2 pointers and 2 integers

$O(n/w)$ bits of space in addition to $n$ bits.

$BV = 10000010 00000100 10000001 00001010 00001011$

num = 16 ones = 3

num = 8 ones = 2

num = 8 ones = 2

num = 16 ones = 5

10000010

00000100

0001010

0001011

0000001

0001010

0001011

num = 16 ones = 5

10000010 00000100 10000001 00001010 00001011
Lemma: Practical Dynamic Bit Vectors

Space

The dynamic bit vector requires $n + O(n/w)$ bits of space.

Proof

- $\Theta(w^2)$ bits per leaf
- $O(n/w^2)$ nodes
- each (inner) node stores 2 pointers (and 2 integers)
- $O(n/w)$ bits of space in addition to $n$ bits
Practical Dynamic Bit Vectors: Access

**Access**
- follow path based on $num$
- requires $O(\log n)$ time \(\blacklozenge\) tree is balanced
- return bit
- example on the board \(\blacklozenge\)

![Diagram showing access in a balanced tree](image)

$BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011$

Example on the board:

- $num = 16$, $ones = 3$
- $num = 8$, $ones = 2$
- $num = 16$, $ones = 5$

$BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011$
Practical Dynamic Bit Vectors: Access

Access
- follow path based on $num$
- requires $O(\log n)$ time \(\circ\) tree is balanced
- return bit
- example on the board 📚

- can return $O(w^2)$ bits at the same cost
- unlike std::vector<bool> 😋

$BV = 10000010 00000100 10000001 00001010 00001011$
Practical Dynamic Bit Vectors: Rank

- keep track of ones to the left
- update based on \textit{ones} stored in node
- traverse tree accordingly in $O(\log n)$ time
- popcount on the leaf in $O(w)$ time
- example on the board

\[ BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011 \]

\[ num = 16 \quad ones = 3 \]
\[ num = 8 \quad ones = 2 \]
\[ num = 16 \quad ones = 5 \]
\[ num = 8 \quad ones = 2 \]
\[ BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011 \]
Practical Dynamic Bit Vectors: Select

- Select is similar to rank
- Keep track of ones
- Or number of bits minus ones for select_0
- Traverse tree accordingly in $O(\log n)$ time
- Popcount and scan on the leaf in $O(w)$ time
- Example on the board

---

$BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011$

Example:

- $num = 16$, $ones = 3$
- $num = 8$, $ones = 2$
- $num = 16$, $ones = 5$
- $num = 8$, $ones = 2$
- $num = 16$, $ones = 5$
- $num = 8$, $ones = 2$
- $num = 16$, $ones = 5$
- $num = 8$, $ones = 2$
- $num = 16$, $ones = 5$
Practical Dynamic Bit Vectors: Insert

- inserting bit traverses down to leaf
- update \( num \) and \( ones \) on the path
- insert in bit vector at leaf
- allocate additional \( w \) bits if necessary
- tracking used space requires \( O(n/w) \) bits

Example:

\[ BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011 \]

\[ num = 16, \ ones = 3 \]
\[ num = 8, \ ones = 2 \]
\[ num = 16, \ ones = 5 \]

\[ num = 8, \ ones = 2 \]
\[ num = 16, \ ones = 5 \]

11/24 2023-07-17 Florian Kurpicz | Advanced Data Structures | 12 Dynamic Bit Vectors & Succinct Trees

Institute of Theoretical Informatics, Algorithm Engineering
Practical Dynamic Bit Vectors: Insert

- inserting bit traverses down to leaf
- update \( \text{num} \) and \( \text{ones} \) on the path
- insert in bit vector at leaf
- allocate additional \( w \) bits if necessary
- tracking used space requires \( O(n/w) \) bits space

- at most every \( w \) inserts a new allocation
- constant time copy of computer word
- are we done? PINGO
Maintaining Leaf Sizes (Insert)

- ensure leaves contain $\Theta(w^2)$ bits
- here $< 2w^2$ bits
Maintaining Leaf Sizes (Insert)

- ensure leaves contain $\Theta(w^2)$ bits
- here $< 2w^2$ bits

- if leaf contains too many bits split leaf
- splitting can require rebalancing of tree
- (left/right) rotation is sufficient
- example on the board
Maintaining Leaf Sizes (Insert)

- Ensure leaves contain $\Theta(w^2)$ bits
- Here $< 2w^2$ bits

- If leaf contains too many bits split leaf
- Splitting can require rebalancing of tree
- (Left/right) rotation is sufficient
- Example on the board

**Lemma: Practical Dynamic Bit Vector Insert Time**

Inserting a bit in the bit vector requires $O(w + \log n)$ time
Determining Leaf Sizes (Insert)

- Ensure leaves contain $\Theta(w^2)$ bits
- Here $< 2w^2$ bits

If leaf contains too many bits, split leaf
- Splitting can require rebalancing of tree
- (left/right) rotation is sufficient
- Example on the board

**Proof**
- Finding leaf takes $O(w)$ time
- Splitting leaf takes $O(w)$ time
- Balancing tree takes $O(\log n)$ time

**Lemma: Practical Dynamic Bit Vector Insert Time**

Inserting a bit in the bit vector requires $O(w + \log n)$ time
Practical Dynamic Rank Data Structure: Delete

- deleting bit traverses down to leaf
- update \( num \) and \( ones \) on the path
- delete in bit vector at leaf
- free \( w \) bits if possible
- tracking used space requires \( O(m/w) \) bits space

![Diagram](image-url)

\[ BV = 1000 0000100 10000001 00001010 00001011 \]
Practical Dynamic Rank Data Structure: Delete

- deleting bit traverses down to leaf
- update \( \text{num} \) and \( \text{ones} \) on the path
- delete in bit vector at leaf
- free \( w \) bits if possible
- tracking used space requires \( O(m/w) \) bits space

- at most every \( w \) deletes a free
- are we done?

\[
\begin{align*}
\text{BV} &= 1000 \ 0000100 \ 1000001 \ 0001010 \ 0001011 \\
\text{num} &= 16 \quad \text{ones} = 3 \\
\text{num} &= 8 \quad \text{ones} = 2 \\
\text{num} &= 16 \quad \text{ones} = 5 \\
\text{num} &= 8 \\
\end{align*}
\]
Maintaining Leaf Sizes (Delete)

- Ensure leaves contain $\Theta(w^2)$ bits
- Here $> w^2/2$ bits
Maintaining Leaf Sizes (Delete)

- ensure leaves contain $\Theta(w^2)$ bits
- here $> w^2/2$ bits

- if leaf contains not enough bits steal bits from preceding or following leaf or
- merge leaves merging does not result in overflow
- merging can require rebalancing of tree
- (left/right) rotation is sufficient
- example on the board
Maintaining Leaf Sizes (Delete)

- Ensure leaves contain $\Theta(w^2)$ bits
- Here > $w^2/2$ bits

- If leaf contains not enough bits steal bits from preceding or following leaf or
- Merge leaves merging does not result in overflow
- Merging can require rebalancing of tree
- (Left/right) rotation is sufficient
- Example on the board

Lemma: Practical Dynamic Bit Vector Insert Time

Deleting a bit in the bit vector requires $O(w + \log n)$ time
Maintaining Leaf Sizes (Delete)

- Ensure leaves contain $\Theta(w^2)$ bits.
- Here > $w^2/2$ bits.

- If leaf contains not enough bits steal bits from preceding or following leaf or merge leaves. Merging does not result in overflow.
- Merging can require rebalancing of tree.
- (left/right) rotation is sufficient.
- Example on the board.

Lemma: Practical Dynamic Bit Vector Insert

Deleting a bit in the bit vector requires $O(w + \log n)$ time.

Proof:
- Finding leaf takes $O(w)$ time.
- Stealing bit requires $O(1)$ time.
- Merging leaves takes $O(1)$ time.
- Balancing tree takes $O(\log n)$ time.
if bit toggles, traverse and update \textit{ones} 
- toggle bit in leaf 
- otherwise (unsure if bit toggles) find bit and 
- if necessary backtrack path and update \textit{ones}
Partial Sums

**Definition: Partial Sum**

Given an array $A$ containing $n$ non-negative numbers all $\leq \ell$

- $\text{sum}(A, i)$ returns $\sum_{j=0}^{i-1} A[j]$ \(\odot\) $\text{sum}(A, 0) = 0$
- $\text{search}(A, j)$ returns $\min\{i \geq 0, \text{sum}(A, i) \geq j\}$
Partial Sums

Definition: Partial Sum

Given an array \( A \) containing \( n \) non-negative numbers all \( \leq \ell \)

- \( \text{sum}(A, i) \) returns \( \sum_{j=0}^{i-1} A[j] \) \( \oplus \) \( \text{sum}(A,0)=0 \)
- \( \text{search}(A, j) \) returns \( \min\{i \geq 0, \text{sum}(A, i) \geq j\} \)

- what has this to do with \( \text{rank} \) and \( \text{select} \)
Partial Sums

**Definition: Partial Sum**

Given an array $A$ containing $n$ non-negative numbers all $\leq \ell$

- $\text{sum}(A, i)$ returns $\sum_{j=0}^{i-1} A[j] \oplus \text{sum}(A, 0)=0$
- $\text{search}(A, j)$ returns $\min\{i \geq 0, \text{sum}(A, i) \geq j\}$

- what has this to do with *rank* and *select*?

- $\text{sum}$ can be answered in $O(1)$ time using $O(wn)$ bits of space
- using $S[i] = \text{sum}(A, i)$
- $\text{search}$ can be answered in $O(\log n)$ time on $S$
Partial Sums

**Definition: Partial Sum**

Given an array $A$ containing $n$ non-negative numbers all $\leq \ell$

- $\text{sum}(A, i)$ returns $\sum_{j=0}^{i-1} A[j] \oplus \text{sum}(A, 0) = 0$
- $\text{search}(A, j)$ returns $\min\{i \geq 0, \text{sum}(A, i) \geq j\}$

- what has this to do with rank and select
- PINGO

**Sampling**

- sample every $k$-th sum in $S$ of length $\lceil n/k \rceil$
- $S[i] = \text{sum}(A, ik)$
- $\text{sum}(A, i) = S[\lfloor i/k \rfloor] + \sum_{j=\lceil i/k \rceil+1}^{i-1} A[j]$  

- sum requires $O(k)$ time
- search requires $O(\log n + k)$
- requiring $O(w \lceil n/k \rceil)$ bits of space

- sum can be answered in $O(1)$ time using $O(wn)$ bits of space
- using $S[i] = \text{sum}(A, i)$
- search can be answered in $O(\log n)$ time on $S$
Theoretical Dynamic Rank and Select Data Structure

- For $\ell = 1$ partial sums is rank and select on bit vectors
- $O(\log n / \log \log n)$ query time [RRR01]
- $n + o(n)$ bits of space
- Amortized update times

- $nH_0(BV) + o(n)$ bits of space with optimal query [HM14; NS14]
- $H_0$ means 0-th order empirical entropy [KM99]
- More on measurements for compressibility in lecture Text-Indexierung
What is a Dynamic Succinct Tree

\[ \text{deletenode}(T, v) \]
- deletes node \( v \) such that
- \( v \)'s children are now children of \( v \)'s parent
- cannot delete the root
**What is a Dynamic Succinct Tree**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>deletenode</strong>((T, v))**</td>
<td>- deletes node (v) such that (v)'s children are now children of (v)'s parent&lt;br&gt;- cannot delete the root</td>
</tr>
<tr>
<td><strong>insertchild</strong>((T, v, i, k))**</td>
<td>- insert new (i)-th child of node (v) such that&lt;br&gt;- the new node becomes parent of&lt;br&gt;- the previously (i)-th to ((i + k - 1))-th child of (v)</td>
</tr>
</tbody>
</table>
## What is a Dynamic Succinct Tree

### `deletenode(T, v)`
- deletes node $v$ such that
- $v$’s children are now children of $v$’s parent
- cannot delete the root

### `insertchild(T, v, i, k)`
- insert new $i$-th child of node $v$ such that
- the new node becomes parent of
- the previously $i$-th to $(i + k - 1)$-th child of $v$

### `insertchild(T, v, i, 0)`
- inserts new leaf

### `insertchild(T, v, i, 1)`
- inserts new parent of only the previously $i$-th child

### `insertchild(T, v, 1, \delta(v))`
- inserts new parent of all $v$’s children
Example of *insertchild*
Example of \textit{insertchild}

\texttt{insertchild}\((T, r, 2, 1)\)

\begin{itemize}
  \item Which one is the hardest representation to insert and delete?
\end{itemize}
Example of \textit{insertchild}

\begin{itemize}
\item $insertchild(T, r, 2, 1)$
\item $insertchild(T, r, 3, 0)$
\end{itemize}
Example of \textit{insertchild}

\begin{align*}
\text{insertchild}(T, r, 2, 1) & \quad \text{insertchild}(T, r, 3, 0) \\
\end{align*}
Example of *insertchild*

```
insertchild(T, r, 2, 1)
insertchild(T, r, 3, 0)
insertchild(T, r, 2, 3)
```
Example of $insertchild$

- $insertchild(T, r, 2, 1)$
- $insertchild(T, r, 3, 0)$
- $insertchild(T, r, 2, 3)$

Which one is the hardest representation to insert and delete?
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
Dynamic LOUDS

**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.

**$\text{insertchild}(T, v, i, k)$**

- add 1 to node
- add 0 at next level accordingly
- only works efficiently with leaves 🎧
Dynamic LOUDS

**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.

**insertchild** \( (T, v, i, k) \)
- add 1 to node
- add 0 at next level accordingly
- only works efficiently with leaves

**deletenode** \( (T, v) \)
- remove 0 representing leaf
- remove 1 representing edge/child
- only works efficiently with leaves
Dynamic BP

**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
Dynamic BP

**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

**insertchild**($T, v, i, k$)
- find parentheses representing subtree under new node
- can be empty if new leaf is inserted
- enclose these parentheses to add new node
**Dynamic BP**

**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

**insertchild(T, v, i, k)**

- find parentheses representing subtree under new node
- can be empty if new leaf is inserted
- enclose these parentheses to add new node

**deletenode(T, v)**

- remove both parentheses belonging to node
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.
Dynamic 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

\[ \text{insertchild}(T, v, i, k) \]

- find position where node is inserted
- if \(i = \delta(v) + 1\) insert at end of subtree
- insert \(^{(k)}\) \(O(w)\) time if \(k = O(w^2)\)
- if \(k > 1\) remove \(k - 1\) left parentheses from \(v\)
Dynamic 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 \( \circ \) to make them balanced

\[ \text{insertchild}(T, v, i, k) \]
- find position where node is inserted
- if \( i = \delta(v) + 1 \) insert at end of subtree
- insert \( \langle k \rangle \) \( O(w) \) time if \( k = O(w^2) \)
- if \( k > 1 \) remove \( k - 1 \) left parentheses from \( v \)

\[ \text{deletenode}(T, v) \]
- find node \( v \) to delete and remove it from bit vector
- update arity of parent by inserting \( \delta(v) - 1 \)
  before \( v \)’s parent
- if \( v \) is leaf remove one left parenthesis instead
LOUDS and BP can be updated in time $O(t_{update})$, where $t_{update}$ is the time to update the bit vector. LOUDS can be updated in the same time, if the dynamic bit vector supports updates of blocks of size $\delta(v)$ for any node $v$.

Dynamic Range Min-Max Tree

- range min-max trees needed for BP and DFUDS
- support operations in $O(\log n)$ time
- now range min-max trees must be dynamic
- we will see this later when introducing range min-max trees
This Lecture
- dynamic bit vectors with rank and select support
- dynamic succinct trees

Advanced Data Structures
- String B-tree
- SA & LCP
- PaChash
- Successor
- CSA
- RMQ
- Kd- & Range Tree
- static BV
- static succ. trees
- range min-max tree
- succ. graphs

Conclusion and Outlook
**Conclusion and Outlook**

### This Lecture
- Dynamic bit vectors with rank and select support
- Dynamic succinct trees
- Partial sum
- Theoretical results for dynamic bit vectors

### Advanced Data Structures

- **String B-tree**
- **SA & LCP**
- **PaCHash**
- **Successor**
- **CSA**
- **RMQ**
- **Kd- & Range Tree**
- **static BV**
- **static succ. trees**
- **range min-max tree**
- **succ. graphs**
Conclusion and Outlook

This Lecture
- dynamic bit vectors with rank and select support
- dynamic succinct trees
- partial sum
- theoretical results for dynamic bit vectors

Next Lecture
- recap
- Q&A
- discussion project

Advanced Data Structures
- String B-tree
- SA & LCP
- PaCHash
- Successor
- CSA
- RMQ
- Kd- & Range Tree
- static BV
- static succ. trees
- range min-max tree
- succ. graphs
Bibliography I


