https://pingo.scc.kit.edu/165540
Recap: Rank Queries on Bit Vectors

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

LOUDS

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

LOUDS

ab ch id e j k f g
1011100110011001100000

BP

ab cd e f g h i j k
(()(()(()()))()(()()))
Recap: Succinct Trees

**LOUDS**

```
ab ch id ejkfg
1011100110011001100000
```

**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

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

- \textit{bitset} and \textit{bitclear} easy without rank and select
- \textit{insert} and \textit{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?**
  - $O(n)$
  - $O(\log n)$
  - $O(1)$

- 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

10011010001111
01001101001111
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

### Questions

- 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?
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?

- why not using a linked list?

Next

- dynamic bit vector including rank and select

10011010001111
01001101001111

10011010001111
01001101001111
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
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

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

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

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

Proof

$Θ(w^2)$ bits per leaf node

$O\left(n/w^2\right)$ nodes each (inner) node stores 2 pointers (and 2 integers)

$O\left(n/w\right)$ bits of space in addition to $n$ bits
Lemma: Practical Dynamic Bit Vectors

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

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

- follow path based on $num$
- requires $O(\log n)$ time ③ tree is balanced
- return bit
- example on the board

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

\[
\begin{align*}
\text{num} &= 16 \quad \text{ones} = 3 \\
\text{num} &= 8 \quad \text{ones} = 2 \\
\text{num} &= 8 \\
\text{num} &= 16 \quad \text{ones} = 5
\end{align*}
\]
Practical Dynamic Bit Vectors: Access

- Access
  - follow path based on \( \text{num} \)
  - requires \( O(\log n) \) time \( \text{tree is balanced} \)
  - return bit
  - example on the board
  - can return \( O(w^2) \) bits at the same cost
  - unlike \texttt{std::vector<bool>}

\[
\text{BV} = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011
\]
Practical Dynamic Bit Vectors: Rank

### 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 

\(BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011\)

\(num = 16 \ \ ones = 3\)
\(num = 8 \ \ ones = 2\)
\(num = 16 \ \ ones = 5\)
\(num = 8 \ \ ones = 2\)
\(10000001 \ 00000100 \ 00001010 \ 00001011\)

\(BV = 10000010 \ 00000100 \ 10000001 \ 00001010 \ 00001011\)
Practical Dynamic Bit Vectors: Select

Select
- 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 🎨

$BV = 10000010 00000100 10000001 00001010 00001011$

$num = 16 \quad \text{ones} = 3$

$num = 8 \quad \text{ones} = 2$

$num = 16 \quad \text{ones} = 5$

$10000010$ $00000100$ $10000001$

$0001010$ $0001011$

$00001010$ $00001011$
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

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

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

\[
\text{BV} = 10000010 00000100 10000001 00001010 00001011
\]

\[
\begin{align*}
\text{num} &= 16 & \text{ones} &= 3 \\
\text{num} &= 8 & \text{ones} &= 2 \\
\text{num} &= 16 & \text{ones} &= 5 \\
\text{10000010} & & \text{00000100} & & \text{10000001} \\
\text{00001010} & & \text{00001011} & & \text{00001011}
\end{align*}
\]
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

Lemma: Practical Dynamic Bit Vector Insert

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

Proof
finding leaf takes $O(w)$ time
splitting leaf takes $O(w)$ time
balancing tree takes $O(\log n)$ time
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
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

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

```
BV = 1000 00000100 10000001 00001010 00001011
```

```
num = 16  ones = 3

num = 8   ones = 2
num = 16  ones = 5

1000
00000100
0001010
001011
```

`num = 16` and `ones = 3`

`num = 8` and `ones = 2`

`num = 16` and `ones = 5`
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

- at most every $w$ deletes a free
- are we done?

![Diagram](BV = 1000 0000100 10000001 00001010 00001011)

$BV = 1000 00000100 00001001 00001010 00001011$
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 Time

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
Practical Dynamic Rank Data Structure: Set/Unset

- if bit toggles, traverse and update *ones*
- toggle bit in leaf
- otherwise (unsure if bit toggles) find bit and
- if necessary backtrack path and update *ones*
Partial Sums

Definition: Partial Sum

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

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

Definition: Partial Sum

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

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

- what has this to do with rank and select

PINGO
Partial Sums

Definition: Partial Sum

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

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

- what has this to do with rank and select

PINGO

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

- $sum$ 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 $S[i] = sum(A, i)$
- $search$ 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

\textit{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><strong>deletenode</strong>((T, v))</th>
<th></th>
<th><strong>insertchild</strong>((T, v, i, k))</th>
</tr>
</thead>
<tbody>
<tr>
<td>deletes node (v) such that</td>
<td></td>
<td>insert new (i)-th child of node (v) such that</td>
</tr>
<tr>
<td>(v)'s children are now children of (v)'s parent</td>
<td></td>
<td>the new node becomes parent of</td>
</tr>
<tr>
<td>cannot delete the root</td>
<td></td>
<td>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 \((T, v, i, k)\)
- Inserts 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 \textit{insertchild}

\begin{itemize}
  \item \texttt{insertchild} (\(T, r, 2, 1\))
  \item \texttt{insertchild} (\(T, r, 3, 0\))
\end{itemize}

which one is the hardest representation to insert and delete
Example of *insertchild*

```
insertchild(T, r, 2, 1)
```

![Diagram of insertchild operation](image)
Example of \textit{insertchild}

\[
\text{insertchild}(T, r, 2, 1) \quad \text{insertchild}(T, r, 3, 0)
\]
Example of $\text{insertchild}$

$\text{insertchild}(T, r, 2, 1)$  $\text{insertchild}(T, r, 3, 0)$

PINGO

![Diagram showing the process of inserting and deleting nodes in a tree structure.](image-url)
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?
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
**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

$\text{insertchild}(T, v, i, k)$
$\text{add} 1$ to node
$\text{add} 0$ at next level accordingly

$\text{deletenode}(T, v)$
$\text{remove} 0$ representing leaf
$\text{remove} 1$ representing edge/child

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.

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

- 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
**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

\textit{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
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 \( 1 \) to make them balanced
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

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

- 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 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 $k$ $O(w)$ 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 $O(t_{\text{update}})$, where $t_{\text{update}}$ 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
Conclusion and Outlook

This Lecture
- dynamic bit vectors with rank and select support
- dynamic succinct trees

Advanced Data Structures
- static/dynamic BV
- static/dynamic succinct trees
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
- static/dynamic BV
- static/dynamic succ. trees
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
- succinct graphs
- range min-max trees
- concluding succinct data structures
- introducing the project tasks

Advanced Data Structures
- static/dynamic BV
- static/dynamic succinct trees
Bibliography I


