Karlsruhe Institute of Technology

Advanced Data Structures

Lecture 12: Dynamic Bit Vectors and Succinct Trees

Florian Kurpicz

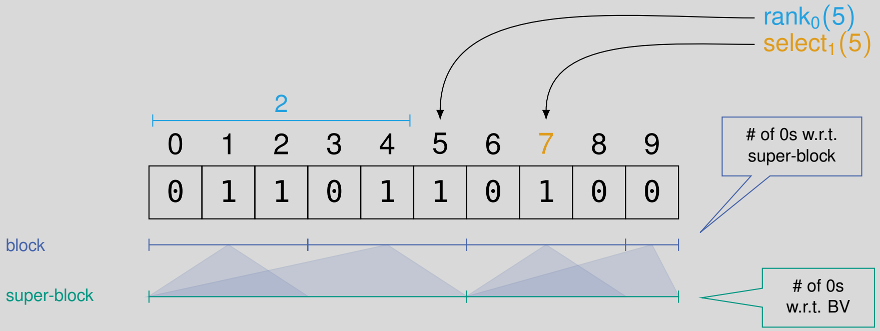
PINGO

https://pingo.scc.kit.edu/737426

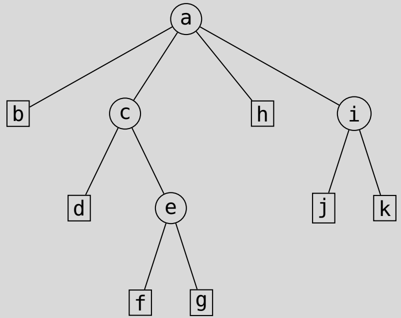
Recap: Rank Queries on Bit Vectors

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

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



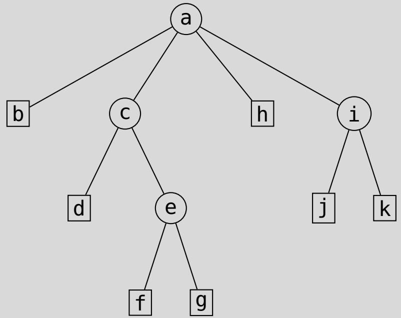
Recap: Succinct Trees



LOUDS

```
     ab  ch  id  ejkfg
1011110011001100110000
```

Recap: Succinct Trees



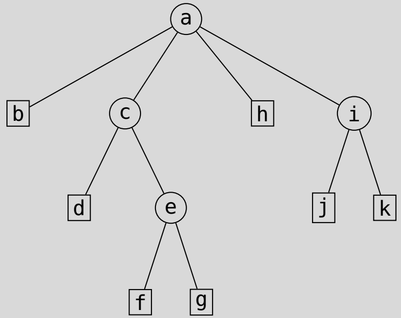
LOUDS

```
      ab  ch  id  ejkfg
1011110011001100110000
```

BP

```
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?

Karlsruhe Institute of Technology

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

- `100110100001111`
- `010011010011111`

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

What is a Dynamic Bit Vector?

Karlsruhe Institute of Technology

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

- `100110100001111`
- `01001101001111`

- what update time do we want to have?
  - $O(n)$
  - $O(\log n)$
  - $O(1)$
- is doubling the length sufficient ℹ amortized analysis **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

- 10011010001111
- 01001101001111

- what update time do we want to have?
  - $O(n)$
  - $O(\log n)$
  - $O(1)$
- is doubling the length sufficient amortized analysis **PINGO**
- why not using a linked list? **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

- `100110100001111`
- `01001101001111`

- what update time do we want to have?
  - $O(n)$
  - $O(\log n)$
  - $O(1)$
- is doubling the length sufficient ℹ amortized analysis **PINGO**
- why not using a linked list? **PINGO**

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 $O(\log n)$

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 $O(\log n)$

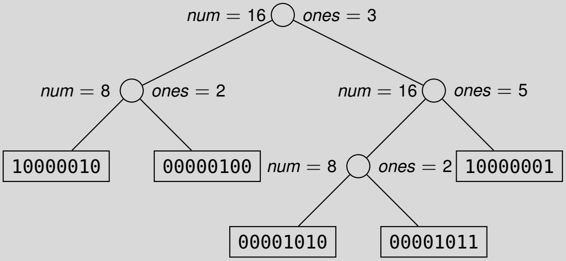
- query time $O(w)$
- $n + O(n/w)$ bits of space
- trade off between query time and space

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



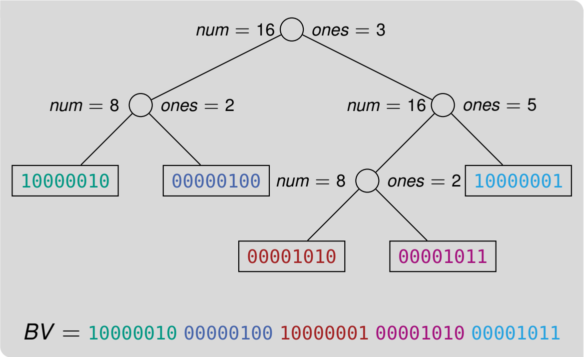
$BV =$ 10000010 00000100 10000001 00001010 00001011

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

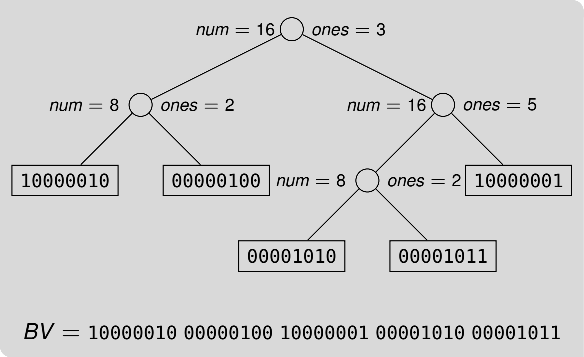
- use pointer-based balanced search tree
- leaves store pointer to $\Theta(w^2)$ bits
- inner nodes store total number of bits (*num*) and number of ones (*ones*) in left subtree



Practical Dynamic Bit Vectors (2/2)

Lemma: Practical Dynamic Bit Vectors Space

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



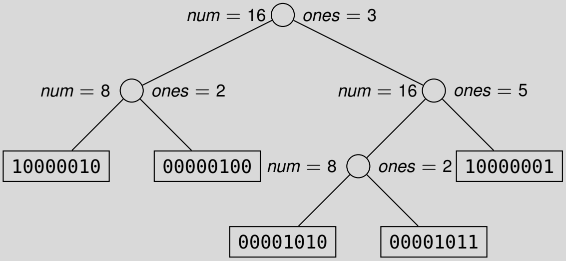
Practical Dynamic Bit Vectors (2/2)

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

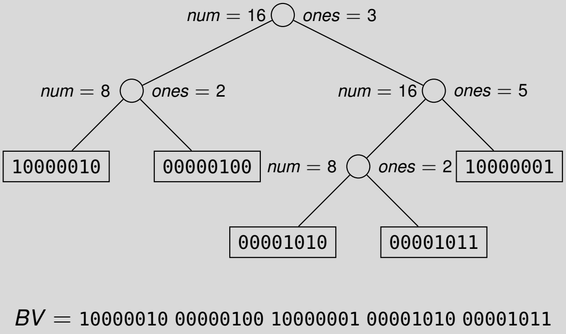


$BV =$ 10000010 00000100 10000001 00001010 00001011

Practical Dynamic Bit Vectors: Access

Access

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

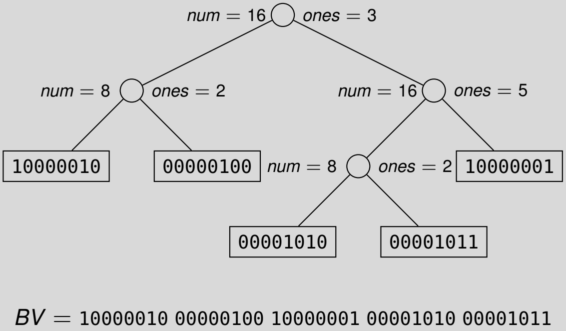


Practical Dynamic Bit Vectors: Access

Access

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

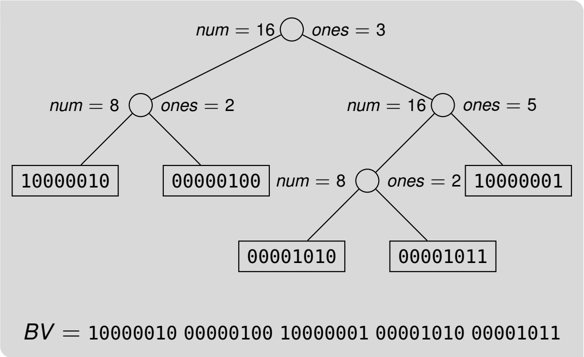
- can return $O(w^2)$ bits at the same cost
- unlike `std::vector<bool>`



Practical Dynamic Bit Vectors: Rank

Rank

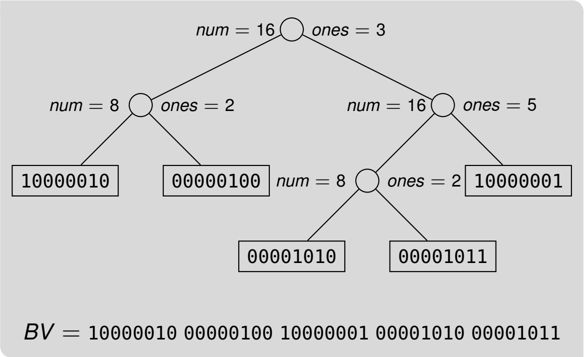
- keep track of ones to the left
- update based on *ones* stored in node
- traverse tree accordingly in $O(\log n)$ time
- popcount on the leaf in $O(w)$ time
- example on the board



Practical Dynamic Bit Vectors: Select

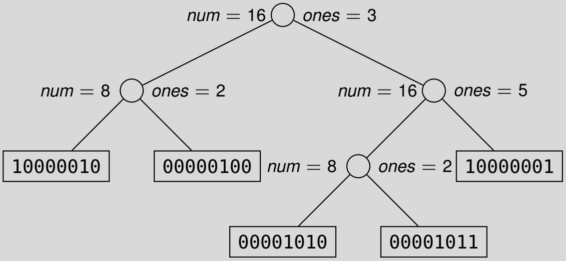
Select

- 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



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 space

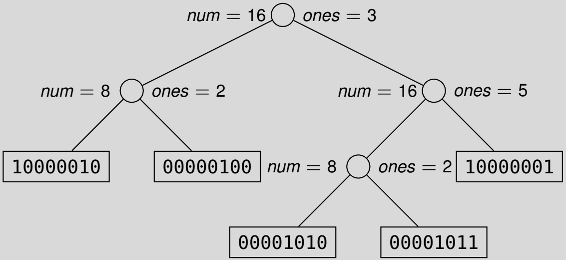


$BV =$ 10000010 00000100 10000001 00001010 00001011

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 space

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



$BV =$ 10000010 00000100 10000001 00001010 00001011

Maintaining Leaf Sizes (Insert)

Karlsruhe Institute of Technology

- 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

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

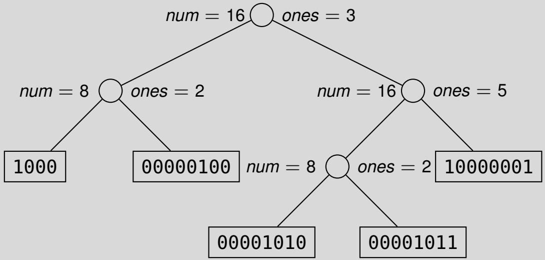
Inserting a bit in the bit vector requires $O(w + \log n)$ time

Proof

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

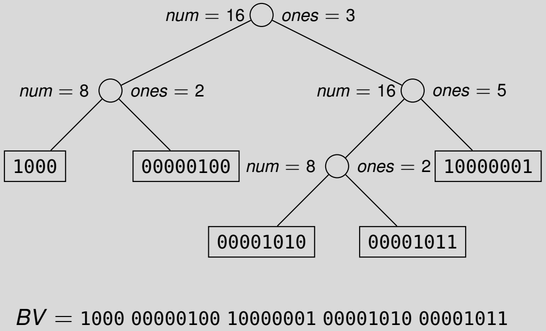


$BV = $ 1000 00000100 10000001 00001010 00001011

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?



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 $\sum_{j=0}^{i-1} A[j]$ ⓘ 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 $\sum_{j=0}^{i-1} A[j]$ 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

Karlsruhe Institute of Technology

Definition: Partial Sum

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

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

- *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

Partial Sums

Karlsruhe Institute of Technology

Definition: Partial Sum

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

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

- 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

Sampling

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

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

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

Karlsruhe Institute of Technology

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

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

What is a Dynamic Succinct Tree

deletenode(T, v)

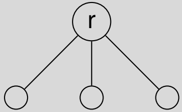
- deletes node v such that
- v's children are now children of v's parent
- cannot delete the root

insertchild(T, v, i, k)

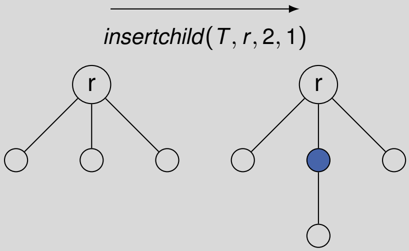
- insert new i-th child of node v such that
- the new node becomes parent of
- the previously i-th to $(i + k - 1)$-th child of v

- *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

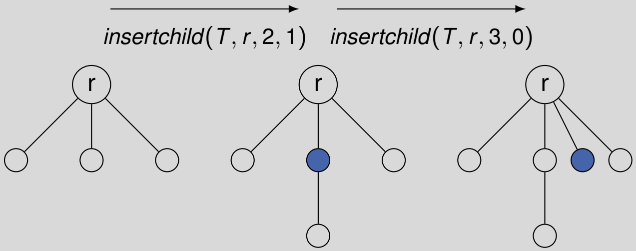
Example of *insertchild*



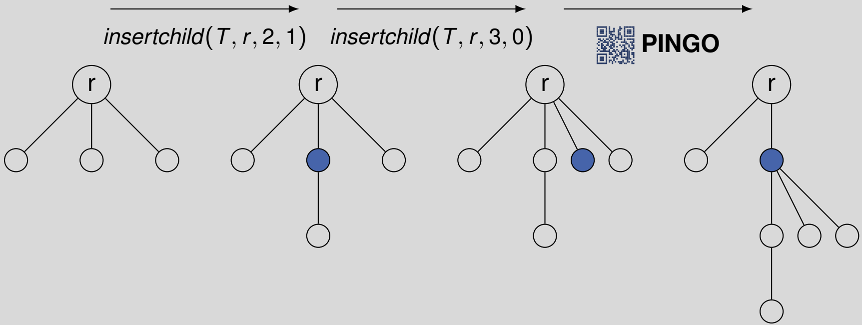
Example of *insertchild*



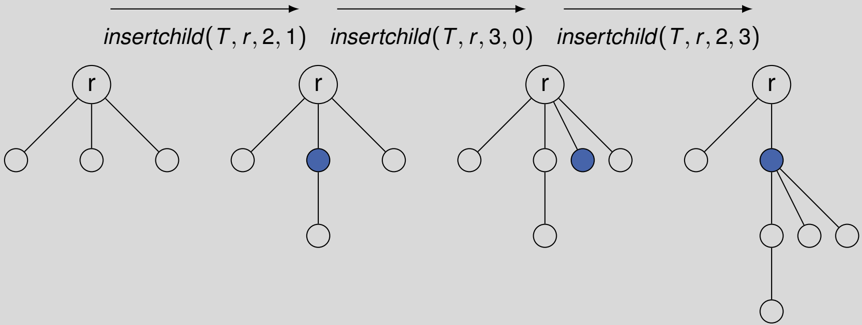
Example of *insertchild*



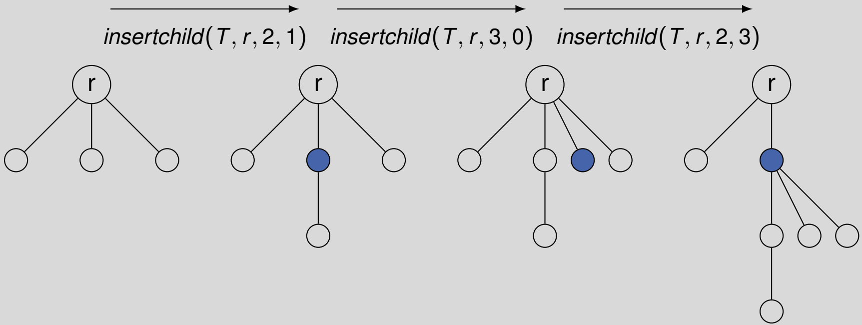
Example of *insertchild*



Example of *insertchild*



Example of *insertchild*



- which one is the hardest representation to insert and delete **PINGO**

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`

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

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

Dynamic DFUDS

Karlsruhe Institute of Technology

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 ℹ to make them balanced

Dynamic DFUDS

Karlsruhe Institute of Technology

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 ℹ to make them balanced

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

- find position where node is inserted
- if $i = \delta(v) + 1$ insert at end of subtree
- insert $(^k)$ ℹ $O(w)$ time if $k = O(w^2)$
- if $k > 1$ remove $k - 1$ left parentheses from v

Dynamic DFUDS

Karlsruhe Institute of Technology

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 ℹ to make them balanced

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

- find position where node is inserted
- if $i = \delta(v) + 1$ insert at end of subtree
- insert $(^k)$ ℹ $O(w)$ time if $k = O(w^2)$
- if $k > 1$ remove $k - 1$ left parentheses from v

$deletenode(T, v)$

- 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

Update Times and Dependencies

- LOUDS and BP can be updated in time $O(t_{\text{update}})$, 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

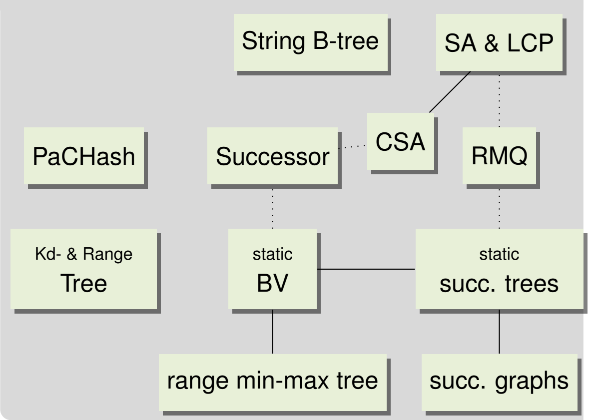
- range min-max trees needed for BP and DFUDS
- support operations in $O(\log n)$ time
- now range min-max trees must be dynamic
- we will see this later when introducing range min-max trees

Conclusion and Outlook

This Lecture

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

Advanced Data Structures



Conclusion and Outlook

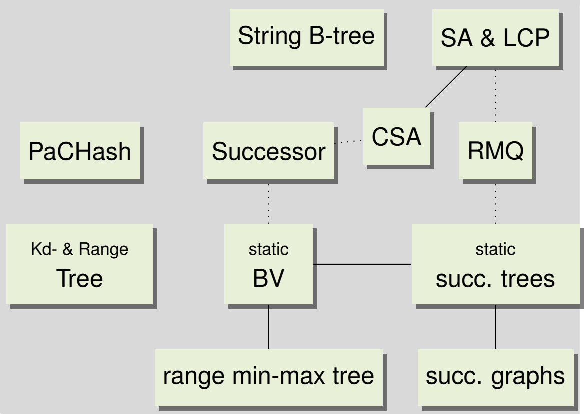
Karlsruhe Institute of Technology

This Lecture

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

- partial sum
- theoretical results for dynamic bit vectors

Advanced Data Structures



Conclusion and Outlook

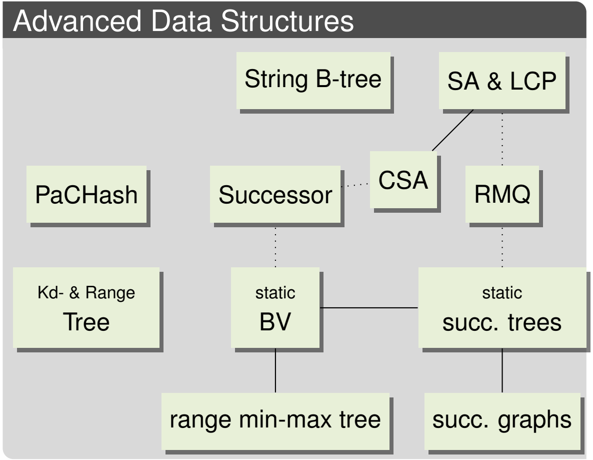
This Lecture

- 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



Bibliography I

[HM14] Meng He and J. Ian Munro. "Space efficient data structures for dynamic orthogonal range counting". In: *Comput. Geom.* 47.2 (2014), pages 268–281. DOI: 10.1016/j.comgeo.2013.08.007.

[KM99] S. Rao Kosaraju and Giovanni Manzini. "Compression of Low Entropy Strings with Lempel-Ziv Algorithms". In: *SIAM J. Comput.* 29.3 (1999), pages 893–911. DOI: 10.1137/S0097539797331105.

[Nav16] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016. ISBN: 978-1-10-715238-0.

[NS14] Gonzalo Navarro and Kunihiko Sadakane. "Fully Functional Static and Dynamic Succinct Trees". In: *ACM Trans. Algorithms* 10.3 (2014), 16:1–16:39. DOI: 10.1145/2601073.

[RRR01] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. "Succinct Dynamic Data Structures". In: *WADS*. Volume 2125. Lecture Notes in Computer Science. Springer, 2001, pages 426–437. DOI: 10.1007/3-540-44634-6_39.