Lemma: Decoding Time Improved CSA

An SA value can be decoded in \(O(\log \log n)\) time using the improved CSA

Proof (Sketch)

- on each level, odd SA values can be decoded using the recursive SA
- there are at most \(\log \log n\) levels
- on each level, even SA values can be decoded in one step, as the next SA value is odd

requires rank and select data structures
Temporal Data Structures

- data structure that allows updates
- queries only on the newest version
- what happens to old versions
Temporal Data Structures

- data structure that allows updates
- queries only on the newest version
- what happens to old versions

- keep old versions around
- in a “clever” way
- lecture based on: http://courses.csail.mit.edu/6.851/spring12/lectures/L01
Temporal Data Structures

- data structure that allows updates
- queries only on the newest version
- what happens to old versions

- keep old versions around
- in a “clever” way
- lecture based on: http://courses.csail.mit.edu/6.851/spring12/lectures/L01

Persistence

- change in the past creates new branch
- similar to version control
- everything old/new remains the same
Temporal Data Structures

- data structure that allows updates
- queries only on the newest version
- what happens to old versions

- keep old versions around
- in a “clever” way

lecture based on: http://courses.csail.mit.edu/6.851/spring12/lectures/L01

Persistence

- change in the past creates new branch
- similar to version control
- everything old/new remains the same

Retroactivity

- change in the past affects future
- make change in earlier version changes all later versions
Definition: Pointer Machine

- nodes containing \( d = O(1) \) fields
- one root node
- operations in \( O(1) \) time
  - new node
  - \( x = y.\text{field} \)
  - \( x.\text{field} = y \)
  - \( x = y+z \)
- access nodes by root.x.y...
Definition: Pointer Machine

- nodes containing \( d = O(1) \) fields
- one root node
- operations in \( O(1) \) time
  - new node
  - \( x = y.\text{field} \)
  - \( x.\text{field} = y \)
  - \( x = y+z \)
- access nodes by root.x.y...
Definition: Pointer Machine

- nodes containing $d = O(1)$ fields
- one root node
- operations in $O(1)$ time
  - new node
  - $x = y.field$
  - $x.field = y$
  - $x = y + z$
- access nodes by root.x.y...
Definition: Pointer Machine

- nodes containing \( d = O(1) \) fields
- one root node
- operations in \( O(1) \) time
  - new node
  - \( x = y.\text{field} \)
  - \( x.\text{field} = y \)
  - \( x = y + z \)
- access nodes by root.x.y... 

- add additional functionality to existing data structures
- is this a “useful” model? PINGO
- balanced binary search tree
- linked list
- ...

example on the board 📚
### Persistence

- keep all versions of data structure
- never forget an old version
- updates create new versions (e.g., insert/delete)
- all operations are relative to specific version

#### Definition: Partial Persistence

Only the latest version can be updated

- versions are linearly ordered
- old versions can still be queries
Persistence

- keep all versions of data structure
- never forget an old version
- updates create new versions e.g., insert/delete
- all operations are relative to specific version

Definition: Partial Persistence
Only the latest version can be updated
- versions are linearly ordered
- old versions can still be queries

Definition: Full Persistence
- Any version can be updated
- versions form a tree
- updates on old versions create branch
Persistence

- keep all versions of data structure
- never forget an old version
- updates create new versions e.g., insert/delete
- all operations are relative to specific version

**Definition: Partial Persistence**
Only the latest version can be updated

- versions are linearly ordered
- old versions can still be queries

**Definition: Full Persistence**
Any version can be updated

- versions form a tree
- updates on old versions create branch

**Definition: Confluent Persistence**
Like full persistence, but two versions can be combined to a new version

**Definition: Functional**
Nodes cannot be modified, only new nodes can be created
Lemma: Making DS Partially Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made partially persistent with

- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update
Lemma: Making DS Partially Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made partially persistent with

- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

Proof (Sketch: Idea)

- store original data and pointer (read only)
- store back pointers to latest version
- store \( \leq 2p \) modifications to fields
  - modification = \((\text{version}, \text{field}, \text{value})\)
  - version \( v \): apply modification with version \( \leq v \)

Proof (Sketch: Functionality)

- read version \( v \)
- look up all modifications \( \leq v \)
- if old version go through old version pointer
- write version
  - if node is not full add modification
  - if node \( n \) is full
    - create new node \( n' \)
    - copy latest version to data fields
    - copy back pointers to \( n' \)
    - for every node \( x \) such that \( n \) points to \( x \)
      - redirect its pointer to \( n' \)
      - for every node \( x \) pointing to \( n \) call recursive change of pointer to \( n' \)
Lemma: Making DS Partially Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made partially persistent with
- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

Proof (Sketch: Idea)
- store original data and pointer (read only)
- store back pointers to latest version
- store \( \leq 2p \) modifications to fields
  - modification = \((\text{version, field, value})\)
  - version \( v \): apply modification with version \( \leq v \)

Proof (Sketch: Functionality)
- read version \( v \)
  - look up all modifications \( \leq v \)
  - if old version go through old version pointer
Partial Persistence (1/3)

Lemma: Making DS Partially Persistent

Any pointer-machine data structure with $p = O(1)$ pointers to any node can be made partially persistent with
- $O(1)$ amortized factor overhead and
- $O(1)$ additional space per update

Proof (Sketch: Idea)
- store original data and pointer (read only)
- store back pointers to latest version
- store $\leq 2p$ modifications to fields
  - modification = $(\text{version}, \text{field}, \text{value})$
- version $v$: apply modification with version $\leq v$

Proof (Sketch: Functionality)
- read version $v$
  - look up all modifications $\leq v$
  - if old version go through old version pointer
- write version
  - if node is not full add modification
  - if node $n$ is full
    - create new node $n'$
    - copy latest version to data fields
    - copy back pointers to $n'$
    - for every node $x$ such that $n$ points to $x$ redirect its pack pointers to $n'$
    - for every node $x$ pointing to $n$ call recursive change of pointer to $n'$
Proof (Sketch: Space)
- adding only constant number of back pointers
- adding only constant number of modifications
- total additional space is $O(1)$

Proof (Sketch: Time)
Read is constant time.
Write requires amortized analysis.
Potential function $\Phi$ amortizes cost:
$$ \Phi(n) = \text{cost}(n) + \Delta \Phi $$

Proof (Sketch: Time cnt.)
Potential $\Phi = c \cdot P$ # modifications in latest version.
Change of potential by adding new modification.
Change of potential by creating new node.
Combined:
$$ \text{amortized cost} \leq c + c - 2cp + p \cdot \text{recursion} $$
First $c$: constant time checking.
Second $c$: adding new modification.
Remaining part if new node is created.
Total amortized time: $O(1)$.
Partial Persistence (2/3)

Proof (Sketch: Space)
- adding only constant number of back pointers
- adding only constant number of modifications
- total additional space is $O(1)$

Proof (Sketch: Time)
- read is constant time
- write requires amortized analysis
Partial Persistence (2/3)

Proof (Sketch: Space)
- adding only constant number of back pointers
- adding only constant number of modifications
- total additional space is $O(1)$

Proof (Sketch: Time)
- read is constant time
- write requires amortized analysis

- potential function $\Phi$
- $\text{amortizes}_{\text{cost}}(n) = \text{cost}(n) + \Delta \Phi$
Partial Persistence (2/3)

Proof (Sketch: Space)
- adding only constant number of back pointers
- adding only constant number of modifications
- total additional space is $O(1)$

Proof (Sketch: Time)
- read is constant time
- write requires amortized analysis
- potential function $\Phi$
- amortizes_cost($n$) = cost($n$) + $\Delta\Phi$

Proof (Sketch: Time cnt.)
- potential $\Phi = c \cdot \sum \#\text{modifications in latest version}$
- change of potential by adding new modification
- change of potential by creating new node
- combined:
  \[
  \text{amortized}_\text{cost} \leq c + c - 2cp + p \cdot \text{recursion}
  \]
- first $c$: constant time checking
- second $c$: adding new modification
- remaining part if new node is created
- total amortized time: $O(1)$
Lemma: Making DS Partially Persistent

Any pointer-machine data structure with $\leq p = O(1)$ pointers to any node can be made partially persistent with

- $O(1)$ amortized factor overhead and
- $O(1)$ additional space per update
Lemma: Making DS Partially Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made partially persistent with

- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

possible in \( O(1) \) worst case time [Bro96]
Lemma: Making DS Partially Persistent

Any pointer-machine data structure with $\leq p = O(1)$ pointers to any node can be made partially persistent with
- $O(1)$ amortized factor overhead and
- $O(1)$ additional space per update

- possible in $O(1)$ worst case time [Bro96]

- also possible for full persistence?
Differences

- versions are no longer numbers
- versions are nodes in a tree
Differences

- versions are no longer numbers
- versions are nodes in a tree

- can we represent versions in a linear fashion?

PINGO
Full Persistence (1/4)

**Differences**
- versions are no longer numbers
- versions are nodes in a tree
- can we represent versions in a linear fashion?

PINGO
Differences

- versions are no longer numbers
- versions are nodes in a tree

can we represent versions in a linear fashion?

PINGO

ab cd ef g h i j k
(()(()(()()))()(()()))

b_a b_b e_b b_c b_d e_d \ldots
Differences
- versions are no longer numbers
- versions are nodes in a tree

- can we represent versions in a linear fashion?

PINGO

ab cd ef g h i j k
(((((())(()))))((()))())

versions change
update in constant time?
Order-Maintenance Data Structure

**Linked List**
- insert before or after element in $O(1)$ time
- check if $u$ is predecessor of $v$ in $n$ time
Order-Maintenance Data Structure

**Linked List**
- insert before or after element in $O(1)$ time
- check if $u$ is predecessor of $v$ in $n$ time

**Balanced Search Tree**
- insert before or after element in $O(\log n)$ time
- check if $u$ is predecessor of $v$ in $O(\log n)$ time
Order-Maintenance Data Structure

**Linked List**
- insert before or after element in $O(1)$ time
- check if $u$ is predecessor of $v$ in $n$ time

**Balanced Search Tree**
- insert before or after element in $O(\log n)$ time
- check if $u$ is predecessor of $v$ in $O(\log n)$ time

**Order-Maintenance DS [DS87]**
- insert before or after element in $O(1)$ time
- check if $u$ is predecessor of $v$ in $O(1)$ time
- how is
Order-Maintenance Data Structure

**Linked List**
- Insert before or after element in $O(1)$ time
- Check if $u$ is predecessor of $v$ in $n$ time

**Balanced Search Tree**
- Insert before or after element in $O(\log n)$ time
- Check if $u$ is predecessor of $v$ in $O(\log n)$ time

**Order-Maintenance DS [DS87]**
- Insert before or after element in $O(1)$ time
- Check if $u$ is predecessor of $v$ in $O(1)$ time
- How is

- Linearized version tree in order-maintenance DS
- Insert in $O(1)$ time
  - New version $v$ of $u$
  - After $b_u$
  - Before $e_u$
- Check order of versions in $O(1)$ time
- Maintain and check linearized version tree in $O(1)$ time
- Important for applying modifications to fields
Lemma: Making DS Fully Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made fully persistent with

- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

Proof (Sketch: Idea)

- store original data and pointer (read only)
- store back pointers to all versions
- store \( \leq 2(d + p + 1) \) modifications to fields

Proof (Sketch: Functionality)

- read version \( v \)
- look up all modifications \( \leq v \)
- if old version go through old version pointer
- write version if node is not full add modification
- the same if node is full?

PINGO

- if node \( n \) is full
- split node into two
- each new node contains half of modifications
- modifications are tree
- partition tree
- apply all modifications to “subtree”
- recursively update pointers
Lemma: Making DS Fully Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made fully persistent with
- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

Proof (Sketch: Idea)
- store original data and pointer (read only)
- store back pointers to all versions
- store \( \leq 2(d + p + 1) \) modifications to fields
  - modification = \((version, field, value)\)
- version \( v \): look at ancestors of \( v \)
Lemma: Making DS Fully Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made fully persistent with

- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

Proof (Sketch: Idea)

- store original data and pointer (read only)
- store back pointers to all versions
- store \( \leq 2(d + p + 1) \) modifications to fields
  - modification = (version, field, value)
- version \( v \): look at ancestors of \( v \)

Proof (Sketch: Functionality)

- read version \( v \)
  - look up all modifications \( \leq v \)
  - if old version go through old version pointer
**Lemma: Making DS Fully Persistent**

Any pointer-machine data structure with \( p = O(1) \) pointers to any node can be made fully persistent with

- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

**Proof (Sketch: Idea)**

- store original data and pointer (read only)
- store back pointers to all versions
- store \( \leq 2(d + p + 1) \) modifications to fields
  - modification = \((\text{version}, \text{field}, \text{value})\)
- version \( v \): look at ancestors of \( v \)

**Proof (Sketch: Functionality)**

- read version \( v \)
  - look up all modifications \( \leq v \)
  - if old version go through old version pointer
- write version
  - if node is not full add modification
  - the same if node is full? PINGO
**Lemma: Making DS Fully Persistent**
Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made fully persistent with
- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

**Proof (Sketch: Idea)**
- store original data and pointer (read only)
- store back pointers to all versions
- store \( \leq 2(d + p + 1) \) modifications to fields
  - modification = \((\text{version, field, value})\)
- version \( v \): look at ancestors of \( v \)

**Proof (Sketch: Functionality)**
- read version \( v \)
  - look up all modifications \( \leq v \)
  - if old version go through old version pointer
- write version
  - if node is not full add modification
  - the same if node is full? PINGO
  - if node \( n \) is full
    - split node into two
    - each new node contains half of modifications
    - modifications are tree
    - partition tree
    - apply all modifications to “subtree”
    - recursively update pointers
Full Persistence (3/4)

Proof (Sketch: Space)
- if no split no additional memory
- if split $O(1)$ memory
Proof (Sketch: Space)

- if no split no additional memory
- if split $O(1)$ memory

Proof (Sketch: Time)

- applying versions in $O(1)$ time
- there are $\leq 2(d + p) + 1$ recursive pointer updates
- potential

$$\Phi = -c \cdot \sum \#empty \ modification \ slots$$
Full Persistence (3/4)

Proof (Sketch: Space)
- if no split no additional memory
- if split $O(1)$ memory

Proof (Sketch: Time)
- applying versions in $O(1)$ time
- there are $\leq 2(d + p) + 1$ recursive pointer updates
- potential

$$\Phi = -c \cdot \sum \#\text{empty modification slots}$$

Proof (Sketch: Time cnt.)
- if node is split $\Delta \Phi = -c \cdot 2(d + p + 1)$
- if node is not split $\Delta \Phi = c$
- combined:
  $$\text{amortized_cost} = c + c$$
  $$- 2c(d + p + 1)$$
  $$+ (2(d + p) + 1) \cdot \text{recursions}$$
- if node is split constants cancel each other out
Lemma: Making DS Fully Persistent

Any pointer-machine data structure with $\leq p = O(1)$ pointers to any node can be made fully persistent with

- $O(1)$ amortized factor overhead and
- $O(1)$ additional space per update
Lemma: Making DS Fully Persistent

Any pointer-machine data structure with \( \leq p = O(1) \) pointers to any node can be made fully persistent with

- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

- versions are represented by tree
- tree has pointers to order-maintenance DS
- order-maintenance DS has pointers to tree
Lemma: Making DS Fully Persistent

Any pointer-machine data structure with \( p = O(1) \) pointers to any node can be made fully persistent with
- \( O(1) \) amortized factor overhead and
- \( O(1) \) additional space per update

- versions are represented by tree
- tree has pointers to order-maintenance DS
- order-maintenance DS has pointers to tree
- de-amortization is open problem
Confluent Persistence

- hard because concatenation
- linked list concatenate with itself
- after $u$ version length $2^u$

more information:
Conclusion and Outlook

This Lecture
- partial and full persistent data structures

Advanced Data Structures

- String B-tree
- SA & LCP
- Successor
- CSA
- RMQ
- static/dynamic BV
- static/dynamic succ. trees
- range min-max tree
- succ. graphs
Conclusion and Outlook

This Lecture
- partial and full persistent data structures

Next Lecture
- retroactive data structures

Advanced Data Structures

- String B-tree
- SA & LCP
- Successor
- CSA
- RMQ
- static/dynamic
  - BV
- static/dynamic
  - succ. trees
- range min-max tree
- succ. graphs
Bibliography I
