2-INF-237 Vybrané partie z dátových štruktúr

2-INF-237 Selected Topics in Data Structures

- Instructor: Broňa Brejová
- E-mail: brejova@fmph.uniba.sk
- Office: M163

- Course webpage: http://compbio.fmph.uniba.sk/vyuka/vpds/

Pointer machine model of computation

- No arrays, no pointer arithmetic
- Constant-size nodes, holding data (numbers, characters, etc) and pointers to other nodes
- Pointers can be assigned:
  – address of a newly created node
  – copy of another pointer
  – null
- Root node with all global variables
  all variables of the form root.current.left.x

Persistent data structures

- partial persistence: update only current version, query any old version, linearly ordered versions
- full persistence: update any version, versions form a tree
- fully functional: never modify nodes, only create new
- backtracking: query and update only current version, revert to an older version
- retroactive: insert updates to the past, delete past updates, query at any past time relative to current set of updates

General transformation for partial persistency

Arbitrary data structure

with $f(n)$ update, $g(n)$ query, $O(n)$ space

Partially persistent version

$O(n/k + f(n))$ update, $O(kf(n) + g(n))$ query

Store whole structure after k updates

Simulate updates from nearest checkpoint

Often balanced at $k \approx \sqrt{n}$

General transformation with fat nodes

Assumes pointer machine model, adds $O(\log n)$ factor overhead

Fat node: binary search tree with time as keys

Each BST node holds a node of the original d.s.

Query of original node at time t: predecessor search in BST

Update of original node: insert a new maximum to BST

Some BSTs allow $O(1)$ maximum inserts and maximum queries

General transformation with fat nodes

Arbitrary pointer machine data structure

with $f(n)$ update, $g(n)$ query

Partially persistent version

$O(f(n))$ update, $O(g(n))$ current query, $O(g(n) \log n)$ past query

General transformation for backtracking

Arbitrary pointer machine data structure

with $f(n)$ update, $g(n)$ query

Version with backtracking

$O(f(n))$ update, $O(g(n))$ query, $O(1)$ amortized backtrack

Use fat nodes with stacks instead of BST

Update adds a new version on the stack, prepays its removal

Query accesses top of the stack

Backtrack pops all stacks until it finds correct time stamp

Each item popped only once, prepaid

What about stacks that do not need change?

General transformation with node copying

Assumes pointer machine and

each node of original structure has **in-degree** $O(1)$

True e.g. for BSTs, but not for union-find

Better partial persisence: remove $O(\log n)$ overhead

Main idea:

avoid searching for time t separately in each fat node

link together versions belonging to the same time

but not all time stamps present in each node

General transformation with node copying

Arbitrary pointer machine data structure

with at most $p = O(1)$ incoming pointers per node

and $f(n)$ update, $g(n)$ query

Partially persistent version

$O(f(n))$ amortized update, $O(g(n))$ query

General transformation with node copying

New node:

– original node

– p reverse pointers for current version only

– $2p$ mods (version, field, value)

multiple such nodes for an original node

Version:

– time t and original root node at time t

– array of roots (not pointer machine) or BST for roots or user supplies root

Read node at time t:

apply all mods with version $< t$

$O(1)$ overhead

General transformation with node copying

Update node, i.e. change $n.x$ from z to y

If node not full, add a new mod

Otherwise add a new node n' with latest version of n

Other nodes may have back pointers to n, change to n'

– to do so, follow forward pointers from n'

Recursively change pointers to n to point to n' in the newest version

– keep pointer to n in the old version

– found using back pointers

Add back pointer from y to n'

Remove back pointer from z to n

Φ: total number of mods in the latest versions of nodes

Retroactive data structures

- Insert(t,updateOp(args))
- Delete(t)
- Query(t,queryOp(args))

  partially retroactive allows query only at present

  fully retroactive query at any time

Commutative and invertible updates

Easy case:

- Order of updates can be permuted
- Each update has an inverse operation
- e.g. search: maintain a set under insert, delete, find (hashing)
- e.g. maintain array, update $A[i] += x$, query $A[i]$
- but not heap insert, delete_min
- Partial retroactivity simply applies updates in the present

Decomposable search problems

Search problem:

maintain a set S with insert and delete

support query(x, S)

Decomposable search problem

query$(x, A \cup B) =$ query$(x, A) \square$ query(x, B)

– operation $\square$ computable in $O(1)$

– possibly require that A and B are disjoint

Examples:

– exact set membership, nearest neighbor, predecessor

– for disjoint sets also rank

Full retroactivity for decomposable search problems

- Item a present in set S during interval (b_a, e_a)

  assume each item inserted only once

- Binary search tree, keys are items, values (b_a, e_a)

- Segment tree, leaves are times when operation inserted

  – each node: data str. for decomposable search for a subset of S

  – element a in nodes in canonical decomposition of (b_a, e_a)

- Retroactive update:

  – find interval (b_a, e_a) in BST and update to (b'_a, e'_a)

  – delete interval (b_a, e_a) from segment tree

  – insert new interval (b'_a, e'_a) to segment tree

  – segment tree with leaf insert/deletes, needs rebalancing (how?)

Full retroactivity for decomposable search problems

- Query at time t:
  - find a leaf for predecessor of t
  - search in each node on the path to root
  - combine results using $\square$
- Overall $\log n$ factor overhead for each update and query as well as for space

Arbitrary data structure

$f(n)$ update, $g(n)$ query

Totally retroactive version

$O(f(n) \log n)$ amortized update, $O(g(n) \log n)$ query

Review: Scapegoat trees

- Lazy amortized binary search trees
- Do not require balancing information stored in nodes
- Insert and delete $O(\log n)$ amortized
  search $O(\log n)$ worst-case
- Invariant: keep the height of the tree at most $\log_{3/2} n$
- When invariant not satisfied, completely rebuild a subtree