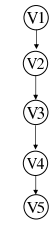
Persistent data structures

Ephemeral: A modification destroys the version which we modify.

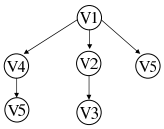
Persistent: Modifications are nondestructive. Each modification creates a new version. All version coexist.

We have a big data structure that represent all versions

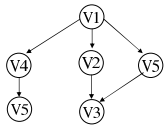
Partially persistent: Can access any version, but can modify only the most recent one.



fully persistent: Can access and modify any version any number of times .



confluently persistent: fully persistent and there is an operation that combines two or more versions to one new version.



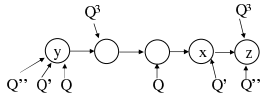
Purely functional: You are not allowed to change a field in a node after it is initialized.

This is everything you can do in a pure functional programming language

Example -- queues

Two operations

Q' = inject(x,Q)
(x,Q') = pop(Q)



Q' = inject(x,Q)
Q'' = inject(z,Q')
(y,Q^3) = pop(Q'')

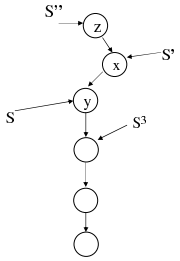
We have partial persistent,
We never want to store two different value in the same field

How do we make the queue fully persistent ?

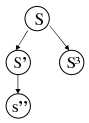
Example -- stacks

Two operations

S' = push(x,S)
(x,S') = pop(S)



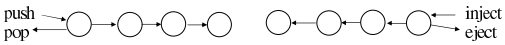
S' = push(x,S)
S'' = push(z,S')
(y,S^3) = pop(S)



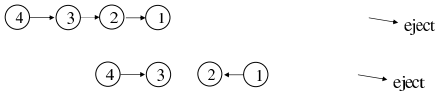
Stacks are automatically fully persistent.

Maybe we should use stacks

Stacks are easy.
We know how to simulate queues with stacks.
So we should be able to get persistent queues this way...



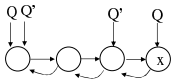
When one of the stacks gets empty we split the other



Example -- double ended queues

four operations

Q' = inject(x,Q)
(x,Q') = eject(Q)
Q' = push(x,Q)
(x,Q') = pop(Q)



(x,Q') = eject(Q)
Q'' = inject(z,Q')

Here its not even obvious how to get partial persistence ?

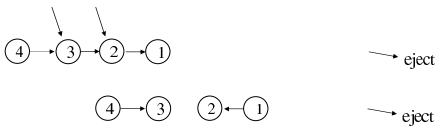
Deque by stack simulation (partial persistence)

$\Phi = | |S_l| - |S_r| |$

Where S is the "live" stack, the one which we can modify

Everything still works

When we do the reversal in order not to modify any other stack we copy the nodes !

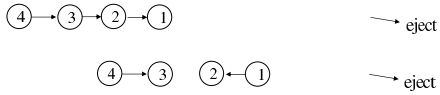


Deque by stack simulation (ephemeral analysis)

$\Phi = | |S_l| - |S_r| |$

Each operation changes the potential by O(1)

The amortized cost of the reverse is 0.



In a persistent setting it is not clear that this potential is well defined

Summary so far

Stacks are automatically fully persistent

Got partially persistent queues in O(1) time per pop/inject

Got partially persistent deques in O(1) amortized time per operation

How about fully persistent queues ?

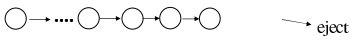
Partially persistent search trees, other data structures ?

Can we do something general ?

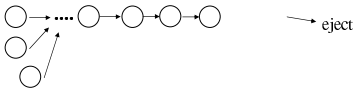
14

Deque by stack simulation (full persistence)

Can repeat the expensive operation over and over again



or



A sequence of n operations that costs $\Omega(n^2)$

13

How about search trees ?

All modifications occur on a path.

So it suffices to copy one path.

This is the path copying method.

16

Some easy observations

You could copy the entire data structure before doing the operation: $\Omega(n)$ time per update, $\Omega(nm)$ space.

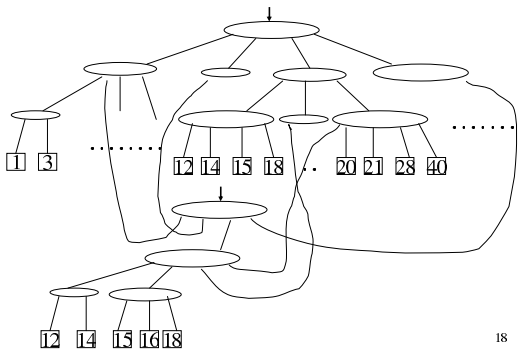
You could also refrain from doing anything just keep a log of the updates. When accessing version i perform first the i updates in order to obtain version i: $\Omega(i)$ time per access, O(m) space.

You could use a hybrid approach that would store the entire sequence of updates and in addition every k^{th} version for some suitable k. Either the space or the access time blows up by a factor of $\sqrt{m}$.

Can you do things more efficiently ?

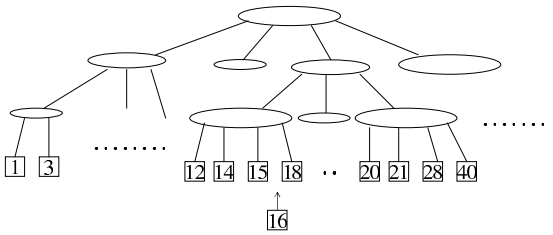
15

Example -- path copying



18

Example -- path copying



17

Application -- planar point location

Suppose that the Euclidian plane is subdivided into polygons by n line segments that intersect only at their endpoints.

Given such polygonal subdivision and an on-line sequence of query points in the plane, the planar point location problem, is to determine for each query point the polygon containing it.

Measure an algorithm by three parameters:

1) The preprocessing time.

2) The space required for the data structure.

3) The time per query.

20

Path copying -- analysis

Gives fully persistent search trees!

O(log n) time for update and access

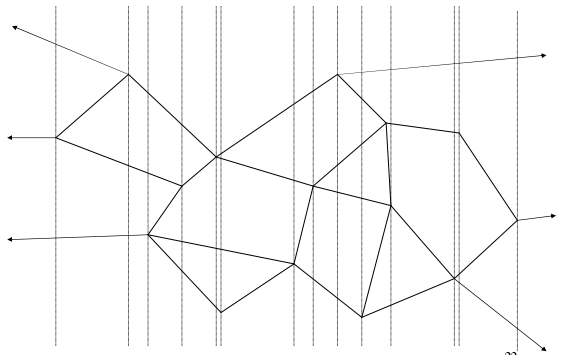
O(log n) space per update

Want the space bound to be proportional to the number of field modifications that the ephemeral update did.

In case of search trees we want the space consumption of update to be O(1) (at least amortized).

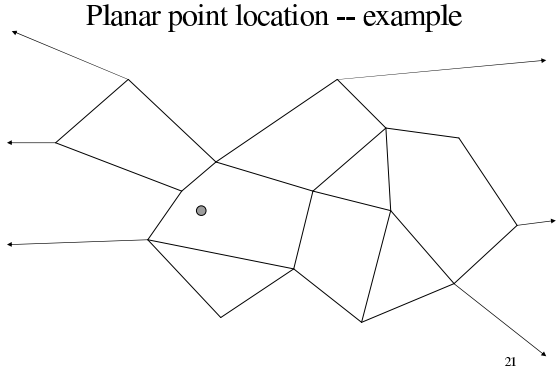
19

Planar point location -- example



22

Planar point location -- example



21

Solving planar point location (Cont.)

To answer query

first find the appropriate slab

Then search the slab to find the polygon

24

Solving planar point location (Cont.)

Partition the plane into vertical slabs by drawing a vertical line through each endpoint.

Within each slab the lines are totally ordered.

Allocate a search tree per slab containing the lines at the leaves with each line associate the polygon above it.

Allocate another search tree on the x-coordinates of the vertical lines

23

Planar point location -- analysis

Query time is O(log n)

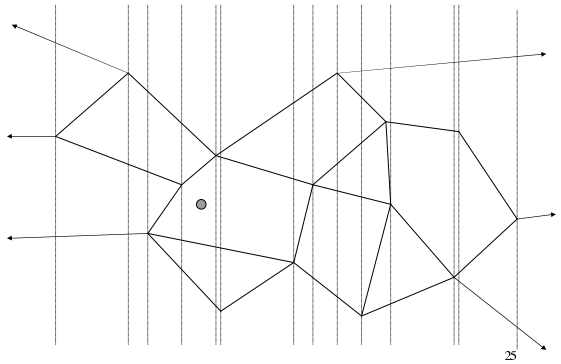
How about the space ?

$\Omega(n^2)$

And so could be the preprocessing time

26

Planar point location -- example



25

Planar point location & persistence

So how do we improve the space bound ?

Key observation: The lists of the lines in adjacent slabs are very similar.

Create the search tree for the first slab.

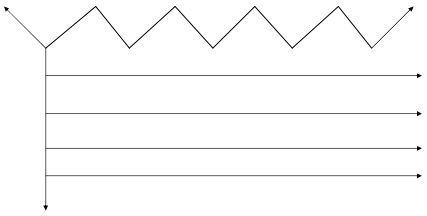
Then obtain the next one by deleting the lines that end at the corresponding vertex and adding the lines that start at that vertex

How many insertions/deletions are there alltogether ?

2n

28

Planar point location -- bad example



Total # lines O(n), and number of lines in each slab is O(n).

27

Making data structures persistent (DSST 89)

We will show a general technique to make data structures partially and later fully persistent.

The time penalty of the transformation would be O(1) per elementary access and update step.

The space penalty of the transformation would be O(1) per update step.

In particular, this would give us an O(n) space solution to the planar point location problem

30

Planar point location & persistence (cont)

Updates should be persistent since we need all search trees at the end.

Partial persistence is enough

Well, we already have the path copying method, lets use it. What do we get ?

O(nlogn) space and O(nlog n) preprocessing time.

We shall improve the space bound to O(n).

29