Persistent and Retroactive Data Structures

Deliverables

Retroactive Data Structures

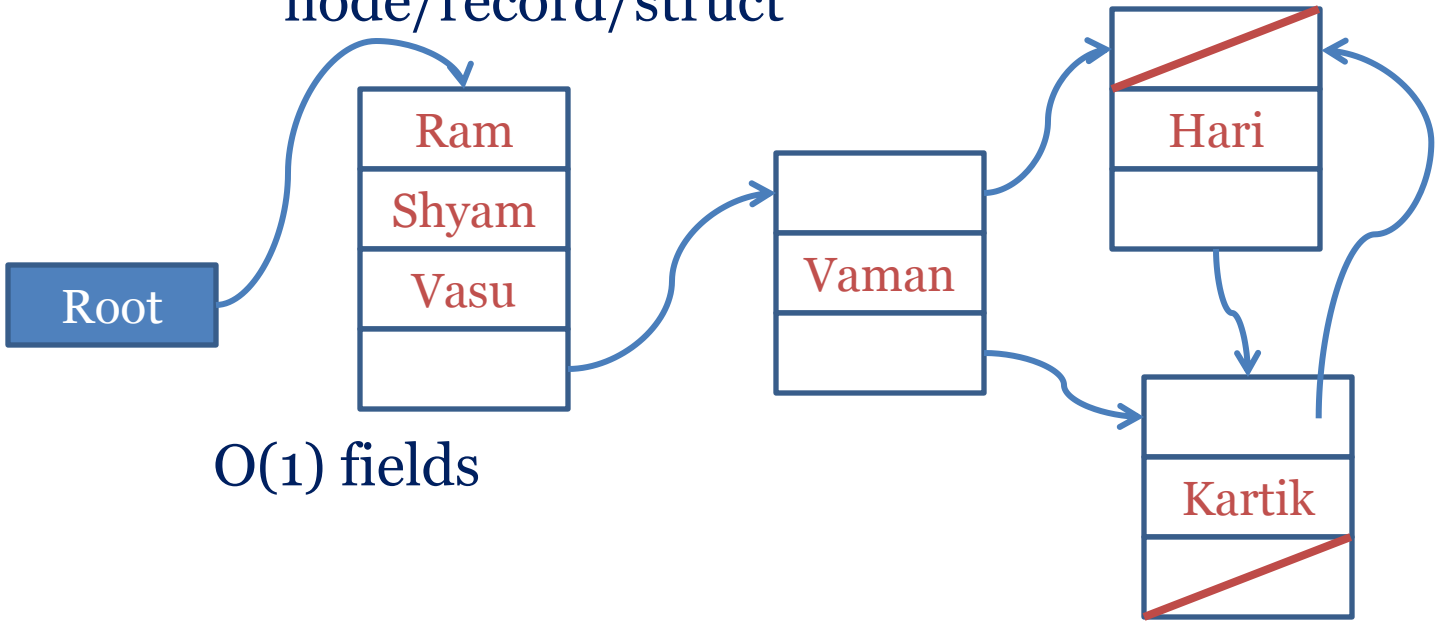
Persistent Data Structures

Different Types of Retroactivity

Succinct Data Structures

Pointer Machine/Model

- node/record/struct



-
- O(1) fields
- Field= data item or pointers to Node

Operations on pointer Machine

- x,y,z are fields of root
- x= new node
- x= y.field
- x.field=y
- x=y+z (data operations)
- Destroy x (If there is no pointer pointing to it)

- Any data structure like linked list, BST, stack, queue etc can be represented with this.

Temporal Data Structures

It is like we can travel in the past and see our past.

If still not satisfied, then we can go to past and change our past.

Never satisfied, then go to past, change the past and based on that present will also be changed.

First property is satisfied by partial persistent data structures.

Second Property is satisfied by Full persistent Data structures

Third Property is satisfied by retroactive Data structures.

Persistence

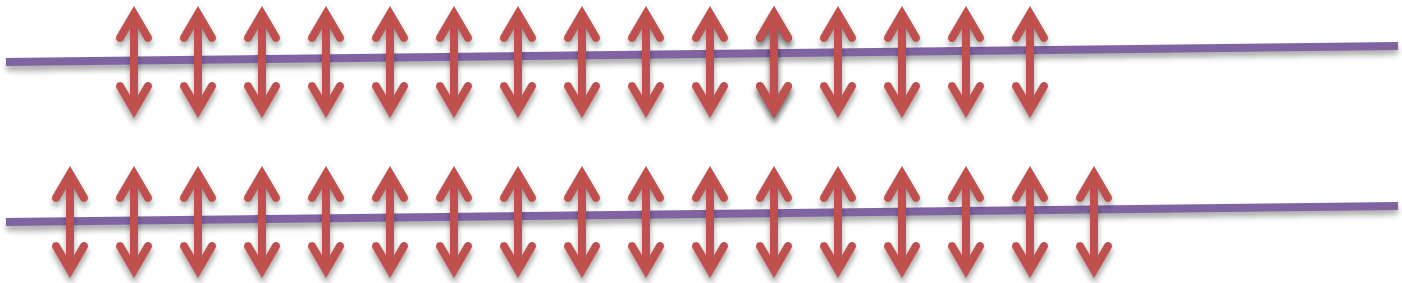
It keeps all versions of Data structure

Data structure operations are done relative to a specified version.

Update creates and returns a new version of the Data structure. It never modifies the existing version.

Partial Persistence

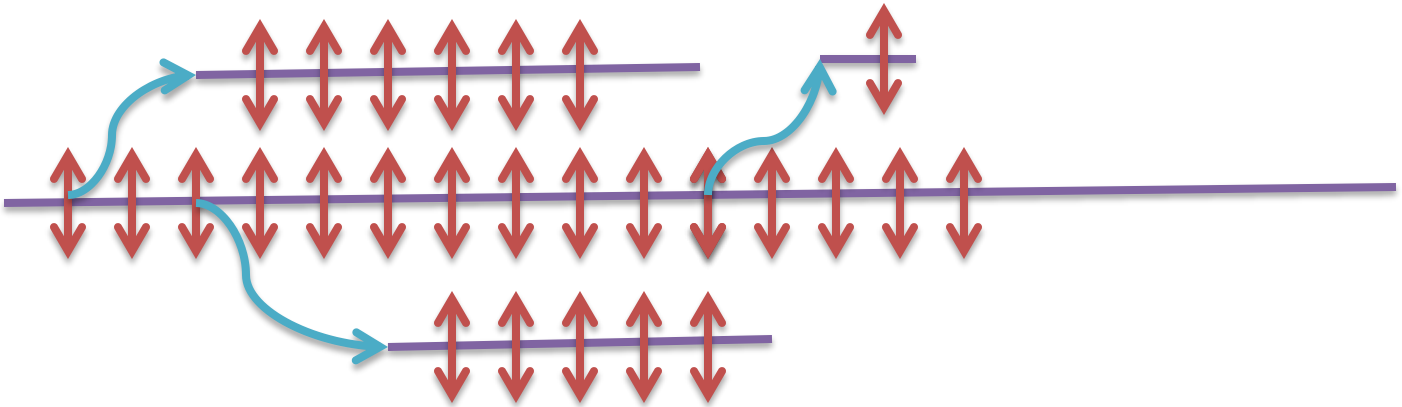
- Updates only Latest Version
- It can query any version
- Versions are linearly ordered.



- Updating the last version creates a new version.

Full Persistence

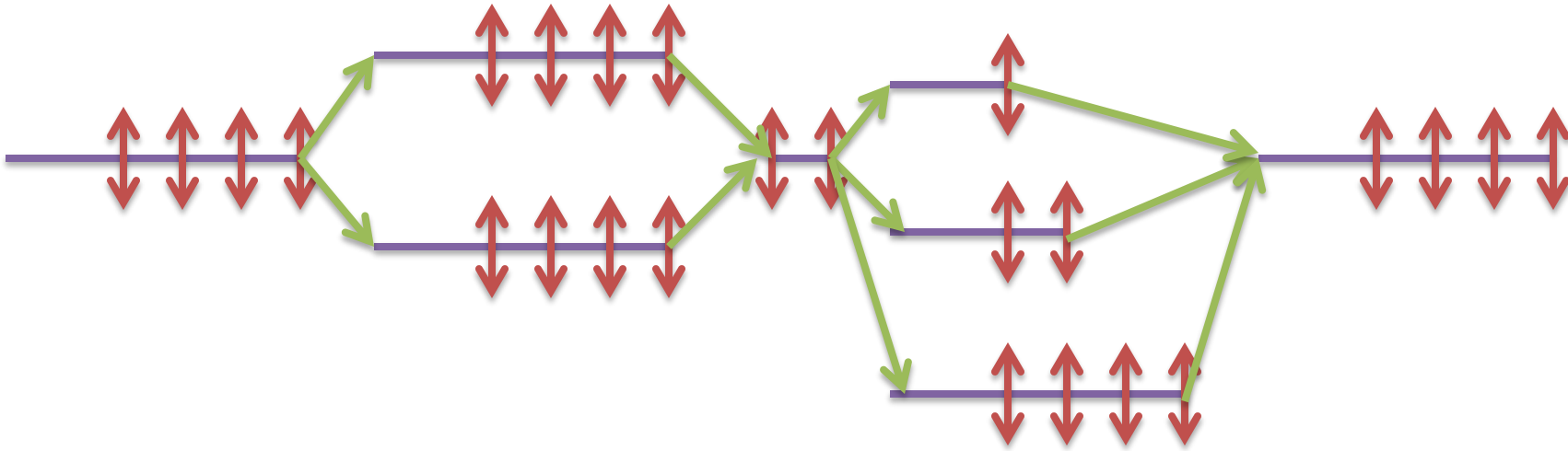
- It can update and query any version of the data structure
- Versions form a tree



- Any updated version can be further updated.

Confluent Persistence

- Can Combine more than one version into new version.
- Versions form a DAG (Directed Acyclic Graph)



Functional Persistence

Never Modifies Nodes

Versions of Data structure represented by pointer

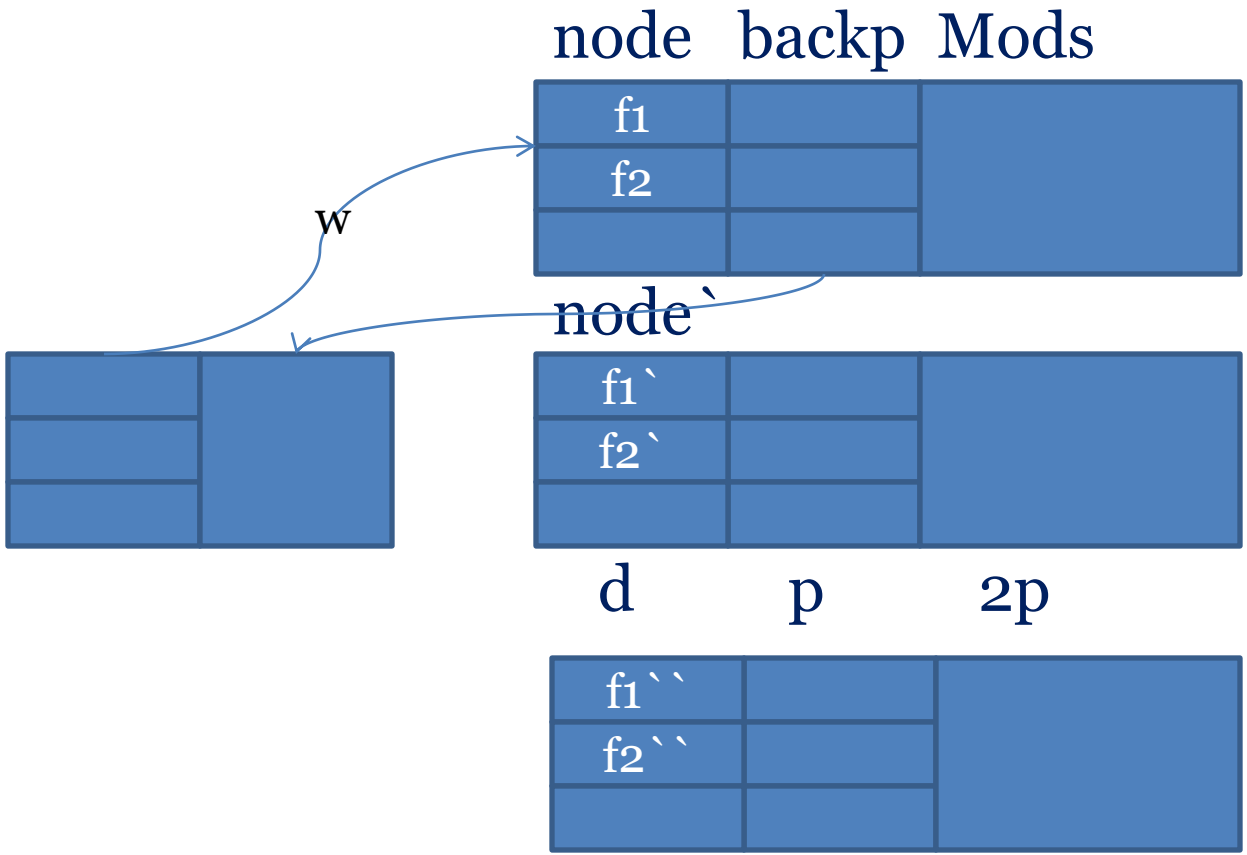
We save the previous data structure as it is, and creates the new one with the new copy of the entire data structure with modification.

In the previous three kind of Persistence we are only keeping a Memo or Meta Data of the updations done on the previous versions.

Partial Persistence

Any pointer machine Data structure with less than p pointers to any node in any version . Always O(1) pointers.

It can be made partially persistent with O(1) amortized multiplicative overhead and O(1) space per change.



Main points

- Store Back pointers for the Latest version only
- Allows storing of ≤ 2p Modifications. It is in the form of (Version, field, value). Assuming that p=O(1)
- To read node.field at version v, check for mods with time ≤ V
- When update changes node.field=x then
- (i) if node not full add mod(now, field,x)
- else create node`=node with last mod applied
- This new node is having empty mods and the old node is full with old mods.

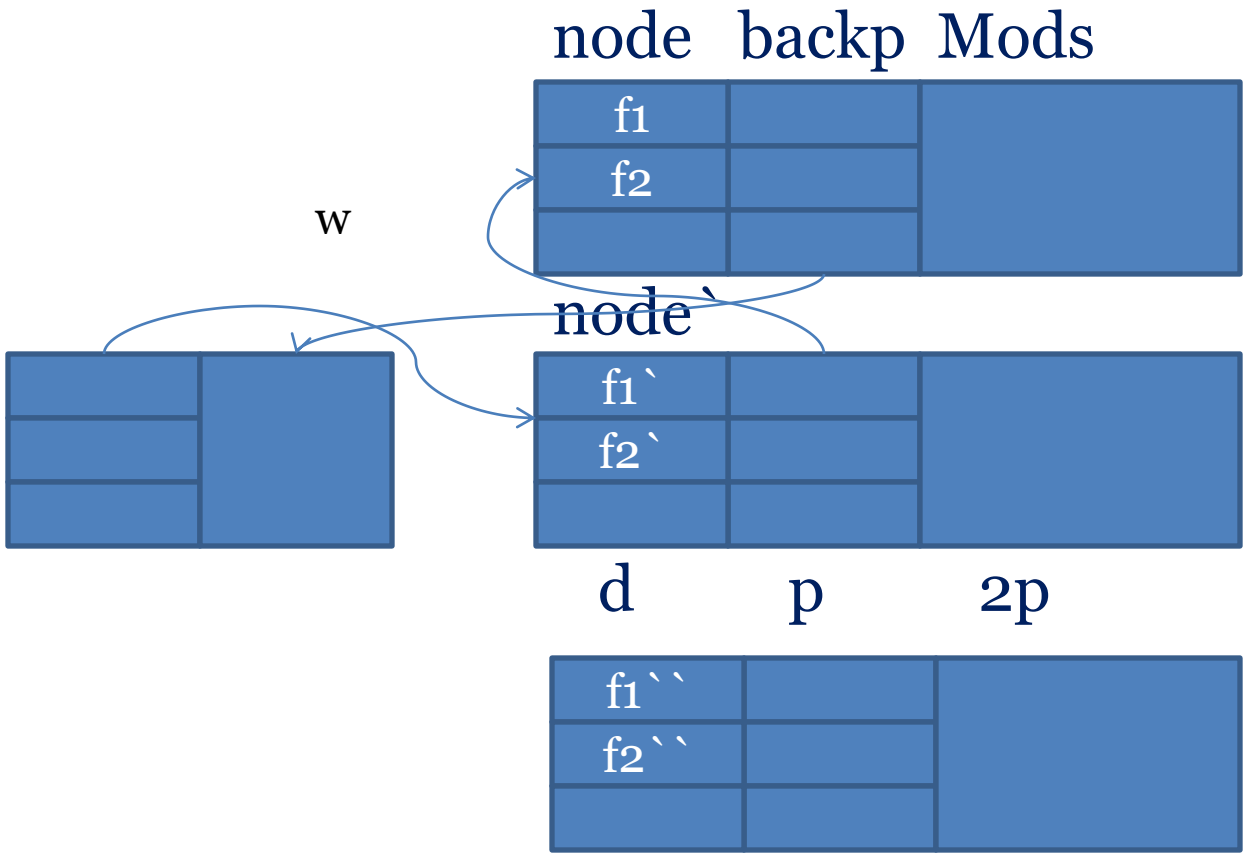
Partial Persistence

Now the pointers pointing to different fields of node from the root should point to the node` fields. And the back pointers of node` should point to the node fields whose back pointers are pointing to the root.

Also update the back pointers because these are to be maintained for the latest version only.

So root node is part of the returned version.

f1`, f2` are not the same as f1 and f2 because these are the values that are the latest ones after applying all the mods of previous node.



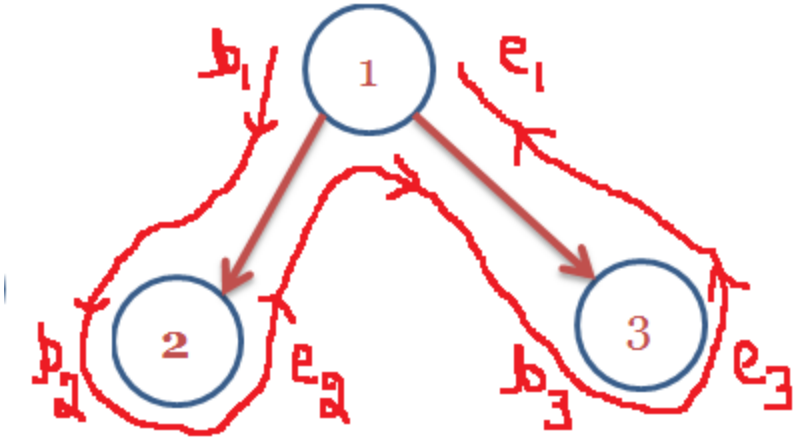
Analysis

- Potential $\Phi = C. \sum$ # mods in the latest version
- Amortized cost $\leq C + C - 2cp + p$ recursions
- Where C is computation cost and C is modification Cost and if we create a new node then our potential goes down by 2p*c because we are getting an empty node instead of a full node and there can be p recursions for changing the pointers till the root.
- Amortized cost $\leq 2C$
- because for p=1 it becomes C+C-2c(C+C-2c+1)+ 1 and same for all recursions.

Full persistence

- Here the problem is that we have a tree data structure and it will become difficult to find the relative ordering of the updates and keep the versioning information.
- We need to somehow linearize the tree and maintain begin and end of each subtrees of versions.
- We linearize tree of versions via inorder traversal marking begin(b1,b2…) and end(e1,e2….) of subtree.
- Store sequence of b's and e's in order maintenance Data structure

- (b1(b2e2)(b3e3)e1)



Full Persistence

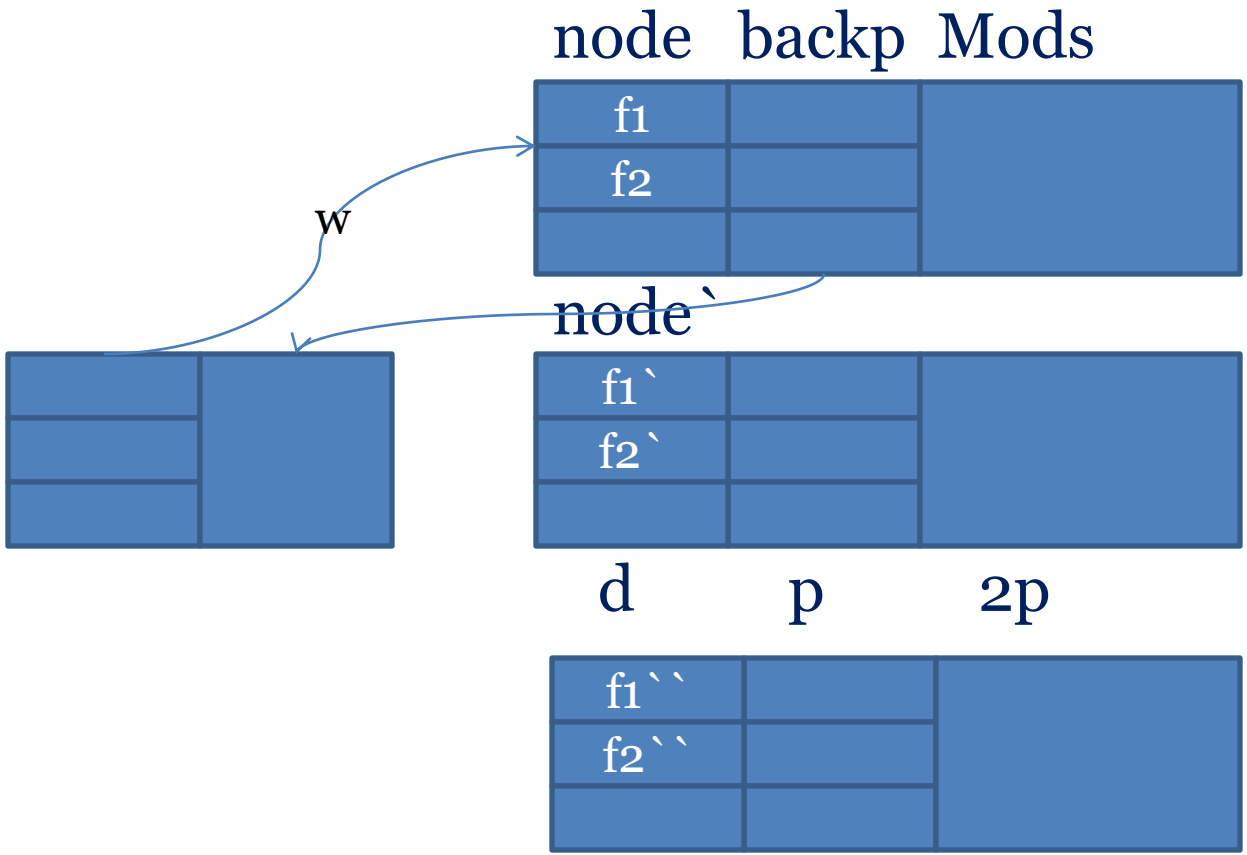
- Insert Item before/after specified item like linked list.
- Find relative order of 2 items in O(1) time per operation.
- Version v ancestor of w implies $b_v<b_w<e_w<e_v$
- It can tell which mods apply to specified version
- create child version of v via 2 inserts after b_v

Complexity Analysis

- Any pointer machine Data structure with less than p pointers to any node in any version . Always O(1) pointers.
- It can be made Fully persistent with O(1) amortized multiplicative overhead and O(1) space per change.

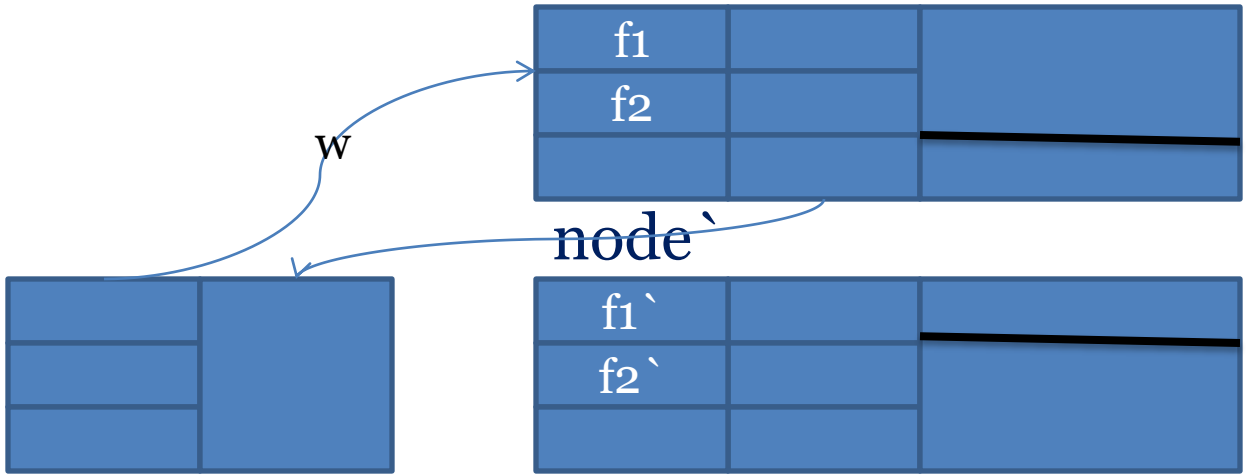
How it works

- Store Back pointers for all versions.
- Allows storing of ≤ 2(d+p+1) Modifications per node. where d is the no of fields and is a constant. It is the maximum out degree from the root and p is the maximum indegree in the form of back pointers. It is in the form of
- (Version, field, value). Assuming that p=O(1)
- To read node.field at version v, check for mods with nearest ancestor of v.

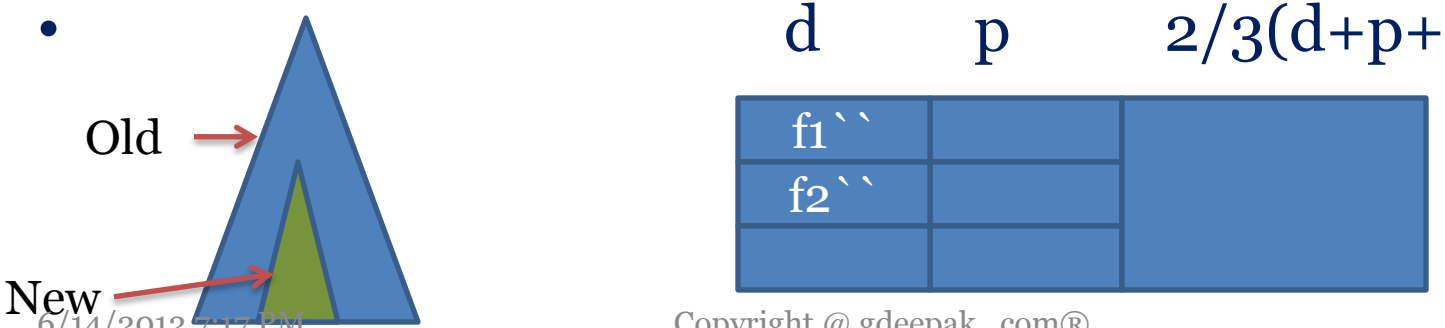


- When update changes node.field=x then
- (i) if node not full add mod(now,field,x)
- when node is full (We can't just create a new node and make the old one non-functional, because update can be applied to any version, not only the latest version). So we need to keep all the nodes functional. In actual we are having a tree of mods and we are splitting it into two partitions of 1/3 and 2/3 size.
- split into two nodes where each is 1/3 and 2/3 full of the original node. It makes copy of the 1/3 modifications applied and rest empty. New node represents the sub tree after the partition
- Recursively update 2d+2p+1 pointers and back pointers similar to partial persistence

- node backp Mods



- d p 2/3(d+p+1)



Potential Analysis

- Potential $\Phi = -C. \sum$ # empty mod slots in both version
- Amortized cost $\leq c. (2(1/3(d+p+1))+2(2/3(d+p+1)))$
- $\leq c(2d+2p+2)$
- In fact here we have an extra c as compared to 2d+2p+1
- So $\Phi = -c\,(2d+2p+1) + c(2d+2p+2) = c$
- So it is O(1) amortized
- Deamortization
- Partial persistence O(1) worst case
- Full persistence Open
-

Retroactive Data structures

- Data structure formed by sequence of updates
- Allow changes to that sequence
- Maintains linear timeline

- insert(x) insert(y) delete-min

- t=0 1 2 … now

Operations

- Insert(t, “operation”) Retroactive do operations at time t
- Delete(t) Retroactively undo operations at time t
- Query(t,”operation”) Execute a query at time t

- Time is specified as index or via order maintenance Data structure

Retroactivity

- Partial Retroactivity: Query only in present(t)
- Full retroactivity: Query at any time

- Easy Cases for partial retroactivity
- Commutative updates : x,y=y,x
- Insert(t,x) is similar to inserting in present
- Invertible updates $x,x^{-1} = \varphi$
- Delete(t,operation) is similar to Insert(now,operation^{-1})

Partial Retroactivity

- Mainly we need to see how we do an operation at some time t in the past and that can do a chain reaction to all other operations after that.
- If we have the properties of commutativity and invertibility that retroactivity may be easy in few cases.
- For example hashing
- As in a dictionary whether you are inserting in past is same as inserting now.

Full retroactivity

- Search Problem: maintain set S of objects subject to insert, delete, query(X,S)
- Search problems have insert and delete as commutativity and invertibility and queries are queries only. So search problems are relatively easy.

Decomposable Search Problem

- Query(x, A∪ B) = f (query (x,A) , query(x,B))

- For example if we have a set of points in 2d space and we need to find the nearest neighbours.
- We can decompose the problems into two spaces and find the nearest neighbours of each one and then in one comparison can find the nearest neighbour of both of them. So Right hand side in many cases may be done in O(1) time.
- Full retroactivity in O(lgn) overhead.

Succinct Data Structures

A bit vector is an array with one array element per domain element. The value is 1 if that is the member of the set under representation and 0 otherwise. So, if the domain is {A,Z} then the bit vector representation can be

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Succinct Data Structures

The find is extremely fast. Simply index position corresponding to the element to be found.

value=1: element ϵ set; value=0: it is not. Time is O(1).

Also, add and delete are O(1)

Actual elements are not stored, so the representation is space compact, only if the elements in the set are relatively dense.

The space required by a bit vector is O(M), where M is the domain size.

Time proportional M to initialize a bit vector.

An example of Time/Space trade-off.

find: O(1) add: O(1) delete: O(1) new: O(1)

Questions, Comments and Suggestions

Question 1

If we can go back and see our past it is called

A) Partial Persistent Data Structures

B) Full Persistent Data Structures

C) Retroactive Data Structures

D) Succinct Data Structures

Question 2

- DAG kind of structure is formed in
- A) Full persistence
- B) Confluent Persistence
- C) Partial Persistence

Question 3

What are the issues in Full persistence Data Structures.