Persistent data structures, fourth lecture

How to make an ephemeral data structure persistent?

Xavier Leroy

2023-03-30

Collège de France, chair of Software sciences
`xavier.leroy@college-de-france.fr`

In this lecture

The persistent data structure we've seen so far have purely functional implementation + (possibly) suspensions for lazy evaluation.

Today we'll study persistent data structures with imperative implementations. Moreover, the persistent structure is derived semi-systematically from an ephemeral structure:

- Arrays $\rightarrow$ persistent arrays
  - By recording the history of updates (Baker's approach)
  - Using "fat elements" (O'Neill and Burton)
- Mutable trees $\rightarrow$ persistent trees
  - Using "fat nodes" (Driscoll, Sarnak, Sleator and Tarjan)

Full persistence, partial persistence

Partial persistence:

- the most recent version of the structure is the only one that can be modified; $\rightarrow$ a sequence of versions
- older versions are read-only.

(Note: this is less restrictive than single-threaded use as seen in the previous lecture.)

Full persistence:

- any version can be modified to create a new version.

$\rightarrow$ a tree of versions

Full persistence comes “for free” with purely functional implementations. But partial persistence suffices for many algorithms (e.g. geometric algorithms).

Persistent arrays, take 1: history of changes

Backtracking on a mutable array

A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

  push $(i, A[i])$ on S

  assign $A[i] := v$.

To backtrack:

  while S is not empty:

    pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array

A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

push $(i, A[i])$ on S

assign $A[i] := v$.

To backtrack:

while S is not empty:

pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array

A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

push $(i, A[i])$ on S

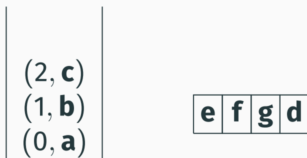
assign $A[i] := v$.

To backtrack:

while S is not empty:

pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array



A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

  push $(i, A[i])$ on S

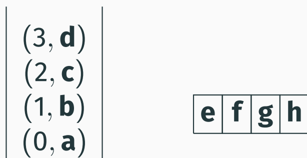
  assign $A[i] := v$.

To backtrack:

  while S is not empty:

    pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array



A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

push $(i, A[i])$ on S

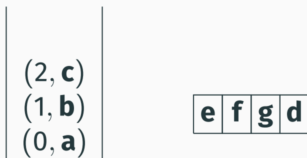
assign $A[i] := v$.

To backtrack:

while S is not empty:

pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array



A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

  push $(i, A[i])$ on S

  assign $A[i] := v$.

To backtrack:

  while S is not empty:

    pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array

A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

  push $(i, A[i])$ on S

  assign $A[i] := v$.

To backtrack:

  while S is not empty:

    pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array

A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

push $(i, A[i])$ on S

assign $A[i] := v$.

To backtrack:

while S is not empty:

pop (i, v) and assign $A[i] := v$.

Backtracking on a mutable array

A mutable array A plus a stack S containing pairs (index i, previous value of $A[i]$).

To write v in $A[i]$:

push $(i, A[i])$ on S

assign $A[i] := v$.

To backtrack:

while S is not empty:

pop (i, v) and assign $A[i] := v$.

Finding the initial value of an array element

No need to backtrack in full if all we need is to find the initial value of $A[i]$:

> for $k = 0, \ldots, |S| - 1$:
>
> > if the k-th entry of S is of the form (i, v):
> >
> > > return v
>
> return $A[i]$.

Finding the value of an array element at date t

To find the value of $A[i]$ at date t (i.e. after t writes in the array), it suffices to start the search at $k = t$ instead of $k = 0$:

> for $k = t, \ldots, |S| - 1$:
>   if the k-th entry of S is of the form (i, v):
>   return v
> return $A[i]$.

Hence, we managed to make the array A partially persistent:

- we can query its state at any date t
- and modify its state from the newest date t, obtaining date $t + 1$.

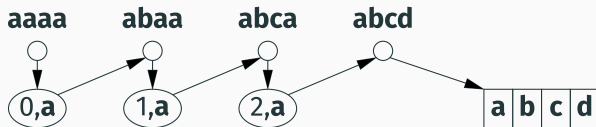
A chained representation of the history

Using references (= indirection cells with in-place mutation).

```
type 'a parray = 'a cell ref
and 'a cell =
  | Base of 'a array
  | Diff of int * 'a * 'a parray

let make size init = ref (Base (Array.make size init))

let rec get p i =
  match !p with
  | Base a -> a.(i)
  | Diff(j, v, q) -> if i = j then v else get q i
```

Note: `get` runs in time linear in the length of the chain of `Diff`.

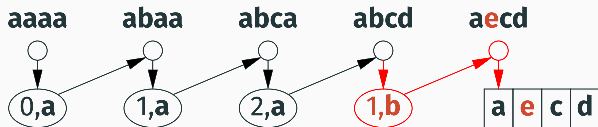
Modifying the base of the persistent array



We modify the array in place, and compensate by inserting a `Diff` node in the history:

```
let set p i v =
  match !p with
  | Base a ->
      let q = ref (Base a) in
      p := Diff(i, a.(i), q); a.(i) <- v; q
  | _ ->
      raise (Failure "full persistence not supported")
```

Modifying the base of the persistent array



We modify the array in place, and compensate by inserting a `Diff` node in the history:

```
let set p i v =
  match !p with
  | Base a ->
      let q = ref (Base a) in
      p := Diff(i, a.(i), q); a.(i) <- v; q
  | _ ->
      raise (Failure "full persistence not supported")
```

A partially persistent array

We obtain a partially persistent array, where

- `set` runs in time $\mathcal{O}(1)$ but applies only to the newest version of the array;
- `get` on the newest version takes time $\mathcal{O}(1)$;
- `get` on any version takes time $\mathcal{O}(w)$,
  where w is the total number of versions (= of writes);
- total space used is $\mathcal{O}(n + w)$, where n is the array size.

Using a global rebuilding technique, we can improve these numbers: `get` in $\mathcal{O}(n)$ worst-case time and `set` in $\mathcal{O}(1)$ amortized time.

Global rebuilding of an array

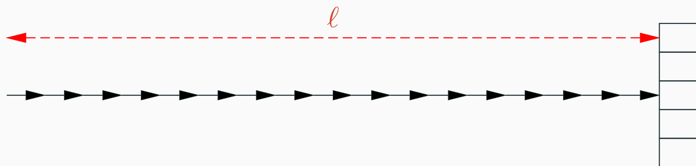
```
let rec to_array p =
  match !p with
  | Base a -> Array.copy a
  | Diff(i, v, q) -> let a = to_array q in a.(i) <- v; a

let rebuild p =
  p := Base (to_array p)
```

We get a new base array where all `Diff` have been applied.

After `rebuild p`, `get` operations on `p` take $\mathcal{O}(1)$ time, but the rebuilding cost $\mathcal{O}(n + \ell)$ time and $\mathcal{O}(n)$ space.

(ℓ = length of the chain of `Diff`)

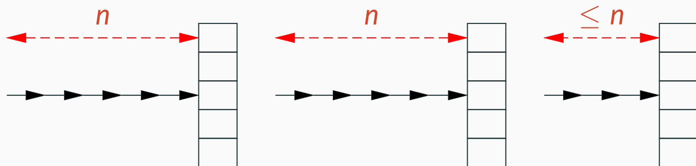
Rebuilding during a `get` operation



If a `get` operation traverses $\ell > n$ `Diff` nodes, it can rebuild $k = \lceil \ell / n \rceil - 1$ intermediate arrays, obtaining $k + 1$ chains of `Diff` of length $\leq n$.

This rebuilding takes time $\mathcal{O}(kn)$ = $\mathcal{O}(\ell)$, which is amortized over the ℓ `set` operations required to create such a long chain of `Diff`.

Rebuilding during a `get` operation



If a `get` operation traverses $\ell > n$ `Diff` nodes, it can rebuild $k = \lceil \ell/n \rceil - 1$ intermediate arrays, obtaining $k + 1$ chains of `Diff` of length $\leq n$.

This rebuilding takes time $\mathcal{O}(kn)$ = $\mathcal{O}(\ell)$, which is amortized over the ℓ `set` operations required to create such a long chain of `Diff`.

Towards full persistence

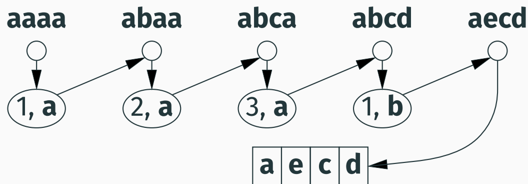
A `Diff` node can contain the old value of an array element, but also its new value!

```
let set p i v =
  match !p with
  | Base a ->
      let q = ref (Base a) in
      p := Diff(i, a.(i), q); a.(i) <- v; q
  | _ ->
      ref (Diff(i, v, p))
```

The array is now fully persistent.

We lose the guarantee that `get` always runs in $\mathcal{O}(1)$ time on the result of the last `set` performed.

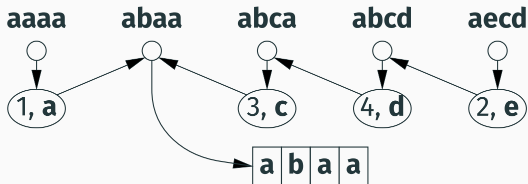
Re-rooting (H. Baker, 1991)



By reversing `Diff` nodes, a persistent array can always be moved to the `Base` state, without allocating a new array.

```
let rec reroot p =
  match !p with
  | Base a -> a
  | Diff(i, v, q) ->
      let a = reroot q in
      p := Base a; q := Diff(i, a.(i), p); a.(i) <- v; a
```

Re-rooting (H. Baker, 1991)



By reversing `Diff` nodes, a persistent array can always be moved to the `Base` state, without allocating a new array.

```
let rec reroot p =
  match !p with
  | Base a -> a
  | Diff(i, v, q) ->
      let a = reroot q in
      p := Base a; q := Diff(i, a.(i), p); a.(i) <- v; a
```

Full persistence by re-rooting

```
let set p i v =
  let a = reroot p in
  let q = ref (Base a) in
  p := Diff(i, a.(i), q);
  a.(i) <- v;
  q
```

The `get` and `set` operations now run in $\mathcal{O}(1)$ time on the result of the latest `set`, and in time $\mathcal{O}(w)$ on other versions of the array. (w = total number of versions)

Extension to other data structures

The "persistent structure = ephemeral structure + history" approach is not specific to arrays.

It also applies to other ephemeral structures, provided their operations are "reversible enough". For example:

- Hash tables (`add` – `remove`)
- Double-ended queues (dequeues) (`cons` – `tail`, `add` – `take`)

Other structures are not reversible enough for this approach:

- Single-ended queues (`tail` – ?, `add` – ?)
- Heaps (`delMin` – `add`, `add` – ?)

Persistent arrays, take 2: using “fat elements”

Arrays of fat elements

(M. E. O'Neill and F. W. Burton, *A new method for functional arrays*, JFP, 1997.)

A variation on the "fat nodes" technique of Driscoll, Sarnak, Sleator and Tarjan (see later).

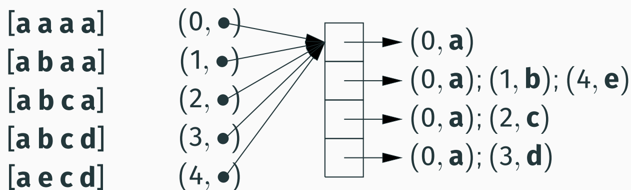
Idea: instead of representing a persistent array by

  a mutable array of values
+ a history of updates for all elements,

let's represent it by

a mutable array of fat elements
where one fat elements = a history for one element.

An array of fat elements



A fat element = an ordered set of pairs
(date of update, new value at that date).

A persistent array = a pair

date t at which we observe the array,

shared mutable array A of fat elements.

The value v of index i at date t =

the element (t', v) of $A[i]$ such that $t' \leq t$ and t' maximal.

An implementation in OCaml

```
module QSet = Set.Make(Q)
module QMap = Map.Make(Q)

type timestamp = Q.t
type 'a fat_element = 'a QMap.t
type 'a collection =
  { arr: 'a fat_element array;
    mutable timestamps: QSet.t }
type 'a parray = timestamp * 'a collection

let make size init =
  let t0 = Q.zero in
  let c = { arr = Array.make size (QMap.singleton t0 init);
            timestamps = QSet.singleton t0 } in
  (t0, c)
```

Reading an array element

```
let get_elt e t =
    match QMap.find_last_opt (fun t' -> Q.leq t' t) e with
    | None -> assert false
    | Some(_, v) -> v

let get (t, c) i =
    get_elt c.arr.(i) t
```

`QMap` is implemented by balanced binary search trees,
hence `get p i` runs in time $\mathcal{O}(\log w_i)$,
where w_i is the number of writes at index i.

Contrast with Baker's persistent arrays, where `get` takes time $\mathcal{O}(w)$, where w is the total number of writes.

Writing to the latest version

```
let set (t, c) i v =
  match QSet.find_first_opt (fun t' -> Q.gt t' t) c.timestamps
  with
  | None ->
      (* Updating the latest version of the array *)
      let t' = Q.(add t one) in
      c.timestamps <- QSet.add t' c.timestamps;
      let e = c.arr.(i) in
      let e' = QMap.add t' v e in
      c.arr.(i) <- e';
      (t', c)
  | Some ->
      raise (Failure "full persistence not supported yet")
```

We create a new version at date $t+1$, and add an entry $(t+1, v)$ to the i-th fat element. Time: $\mathcal{O}(\log w_i)$.

Analysis in the partial persistence case

After n writes, we can rebuild a new array in time $\mathcal{O}(n \log n)$
(n `get` requests over the old array).

This time and this space are amortized over the n preceding writes.

Then, each fat element has size $\leq n$. Therefore:

- reads take $\mathcal{O}(\log n)$ worst-case time
- writes take $\mathcal{O}(\log n)$ amortized time
- the array uses $\mathcal{O}(n + w)$ space if fat elements are represented by ephemeral BSTs, updated in place.

Using splay trees, we obtain accesses in $\mathcal{O}(1)$ time for elements that have been accessed recently.

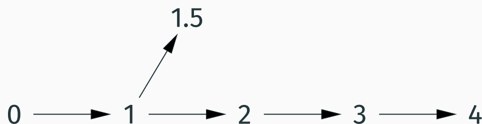
Writing to an arbitrary version

When writing to the version at date t, if versions at date $> t$ already exist, we pick a new date between t and the dates of these later versions.

It's a simple way to embed the partial order of versions in the total order of dates.

Writing to an arbitrary version

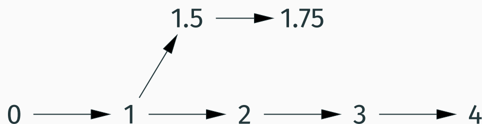
When writing to the version at date t, if versions at date $> t$ already exist, we pick a new date between t and the dates of these later versions.



It's a simple way to embed the partial order of versions in the total order of dates.

Writing to an arbitrary version

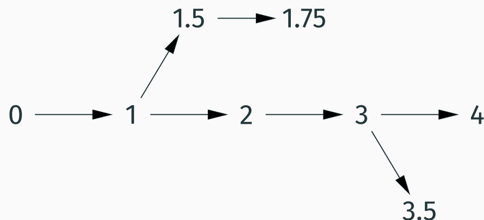
When writing to the version at date t, if versions at date $> t$ already exist, we pick a new date between t and the dates of these later versions.



It's a simple way to embed the partial order of versions in the total order of dates.

Writing to an arbitrary version

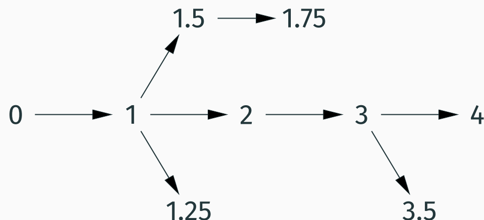
When writing to the version at date *t*, if versions at date $> t$ already exist, we pick a new date between *t* and the dates of these later versions.



It's a simple way to embed the partial order of versions in the total order of dates.

Writing to an arbitrary version

When writing to the version at date t, if versions at date $> t$ already exist, we pick a new date between t and the dates of these later versions.



It's a simple way to embed the partial order of versions in the total order of dates.

Intermediate dates

Given two dates t, t'', we need to create a date in between, e.g. $t' = (t + t'')/2$.

- Simple solution: a date = a rational number
  or at least a dyadic fraction $p/2^q$.
  $\to$ Comparisons between dates are expensive
  ($\mathcal{O}(w)$ in the worst case).
- Clever solution: solving the order maintenance problem
  (see later).

Writing to an arbitrary version

When writing value v at an intermediate date $t' = (t + t'')/2$, it may be necessary to add two entries to the fat element:

- one for date t' with value v;
- aonther for date t'' with the value of the element at date t.

Example: consider $t = 1$ and $t'' = 2$:

Fat element	Value at date				
	0	1	2	3	
$(0, \mathbf{a}); (3, \mathbf{b})$	**a**	**a**	**a**	**b**	
$(0, \mathbf{a}); (1.5, \mathbf{c}); (3, \mathbf{b})$	**a**	**a**	**c**	**b**	✘
$(0, \mathbf{a}); (1.5, \mathbf{c}); (2, \mathbf{a}); (3, \mathbf{b})$	**a**	**a**	**a**	**b**	✔

Writing to an arbitrary version

```
let set (t, c) i v =
  match QSet.find_first_opt (fun t' -> Q.gt t' t) c.timestamps with
  | None ->
      (* Updating the latest version of the array *)
      ...
  | Some t'' ->
      (* Updating an earlier version *)
      let t' = Q.(div (add t t'') (of_int 2)) in
      c.timestamps <- QSet.add t' c.timestamps;
      let e = c.arr.(i) in
      let e1 =
        if QMap.mem t'' e then e else QMap.add t'' (get_elt t e) e in
      let e' = QMap.add t' v e1 in
      c.arr.(i) <- e';
      (t', c)
```

Analysis in the full persistence case

When the number of versions reach n, we can split the array in two arrays containing $\approx n/2$ versions each.

(See O'Neill and Burton for more details.)

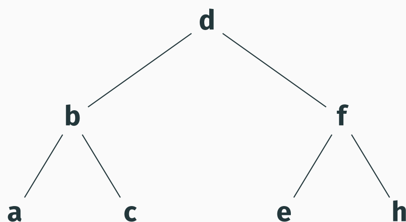
The previous analysis still applies:

- reads take $\mathcal{O}(\log n)$ worst-case time
- writes take $\mathcal{O}(\log n)$ amortized time
- the array uses $\mathcal{O}(n + w)$ space if fat elements are represented by ephemeral BSTs, updated in place

... provided we have an efficient solution to the order maintenance problem, that is, dates that fit in a fixed number of machine words and that can be compared in constant time.

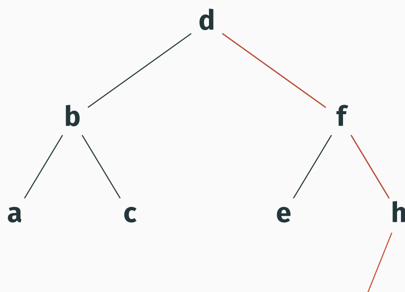
Persistent binary trees using fat nodes

Insertion in a BST: imperative version (from lecture 2)



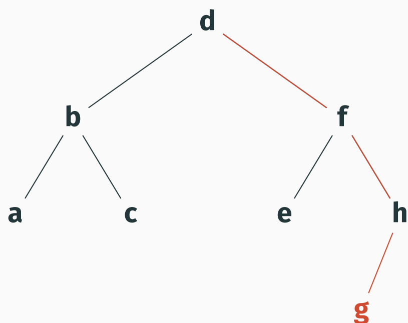
1. Search for the element to be inserted (here, **g**).

Insertion in a BST: imperative version (from lecture 2)



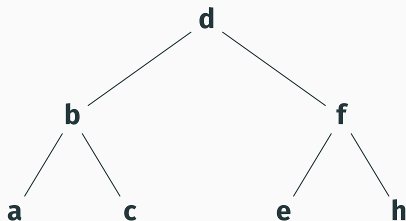
1. Search for the element to be inserted (here, **g**).

Insertion in a BST: imperative version (from lecture 2)



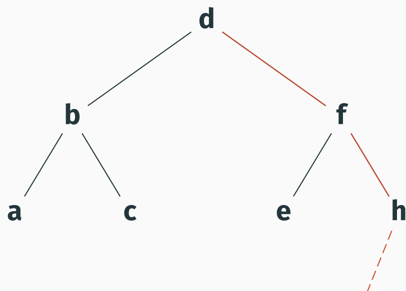
1. Search for the element to be inserted (here, **g**).
2. When reaching a leaf, replace it by the node $\langle \bullet, \mathbf{g}, \bullet \rangle$.

Insertion in a BST: persistent version (from lecture 2)



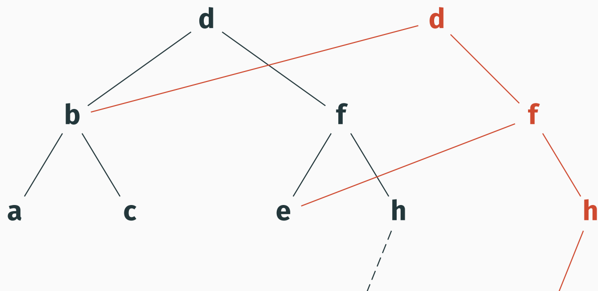
1. Search for the element to be inserted (here, **g**).

Insertion in a BST: persistent version (from lecture 2)



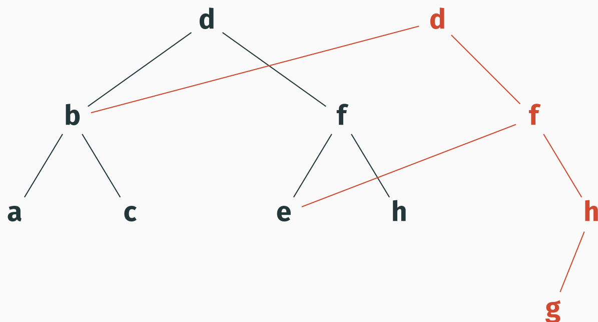
1. Search for the element to be inserted (here, **g**).

Insertion in a BST: persistent version (from lecture 2)



1. Search for the element to be inserted (here, **g**).
2. When reaching a leaf, copy the path from the root to this leaf, sharing sub-trees with the initial tree.

Insertion in a BST: persistent version (from lecture 2)



1. Search for the element to be inserted (here, **g**).
2. When reaching a leaf, copy the path from the root to this leaf, sharing sub-trees with the initial tree.
3. At the end of the copied path, add the node $\langle \bullet, \mathbf{g}, \bullet \rangle$.

Insertion in a BST

The purely functional implementation (using path copying) is as fast as the imperative implementation but less space efficient:

	Time	Space
Purely functional	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Imperative	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

If we perform n insertions starting with an empty tree, the set of all n versions of the tree uses $\mathcal{O}(n \log n)$ space with the purely functional implementation.

N. Sarnak and R. E. Tarjan (1986): we can get $\mathcal{O}(n)$ space by making the imperative implementation persistent.

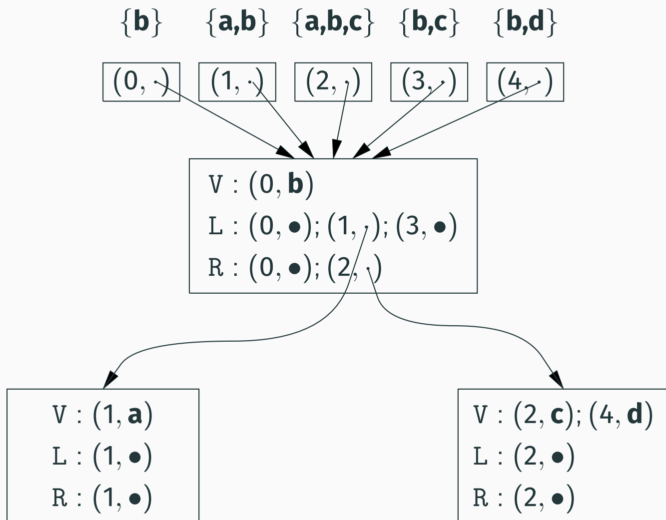
A BST with fat nodes

A fat node = a record

- `V` field: a history (date, value)
- `L` field: a history (date, left sub-tree)
- `R` field: a history (date, right sub-tree)
- balancing information: height, red/black color, etc.

At first, we shoot for partial persistence, hence the balancing information (which serves for updates only) does not need to be versioned and applies to the latest version only.

Example of a BST with fat nodes



Rebalancing via rotations after insertions

{**a**} {**a,b**} {**a,b,c**}

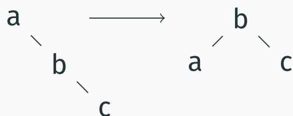
(0, ·) (1, ·) (2, ·)

V : (0, **a**)
L : (0, •)
R : (0, •); (1, ·); (2, •)

V : (1, **b**)
L : (1, •); (2, ·)
R : (1, •); (2, ·)

V : (2, **c**)
L : (2, •)
R : (2, •)

Corresponds to the rotation



Analysis

After n insertions, starting with an empty tree:

Searching for an element: time $\mathcal{O}(\log^2 n)$

(we traverse a branch of length $\log n$; at each node, we search the histories for the `V`/`L`/`R` fields, which are sets of size at most n).

Inserting an element: same time, $\mathcal{O}(1)$ space

(each insertion creates one node and updates one field,
then performs a small number of rotations:
at most 2 rotations for a red-black tree).

The n versions therefore fit in $\mathcal{O}(n)$ space.

Bounding the sizes of fat elements

Assume that each history (for `V`, `L`, `R` fields) contains at most k entries. Then, searching a history takes constant time, and searching in the whole BST takes $\mathcal{O}(\log n)$ time.

How to update a field `V`/`L`/`R` of a fat node?

- If the corresponding history is not full, we add one entry to it (constant time).
- Otherwise, we create a copy of the node with one-entry histories (the new values for `V`, `L`, `R`), and we recursively update the `L` or `R` field of the parent node.

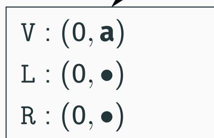
Example of updates with histories of size $k = 2$

$t = 0$

$\mathtt{V} : \ldots$

$\mathtt{L} : \ldots$

$\mathtt{R} : (0, \cdot)$



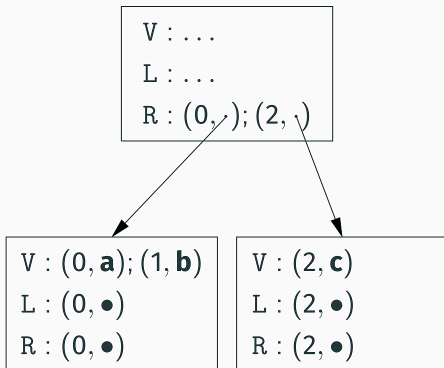
Example of updates with histories of size $k = 2$

$t = 0, 1$

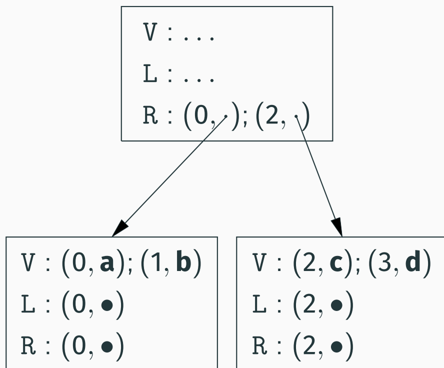
```
V : ...
L : ...
R : (0, ·)
```

```
V : (0, a); (1, b)
L : (0, •)
R : (0, •)
```

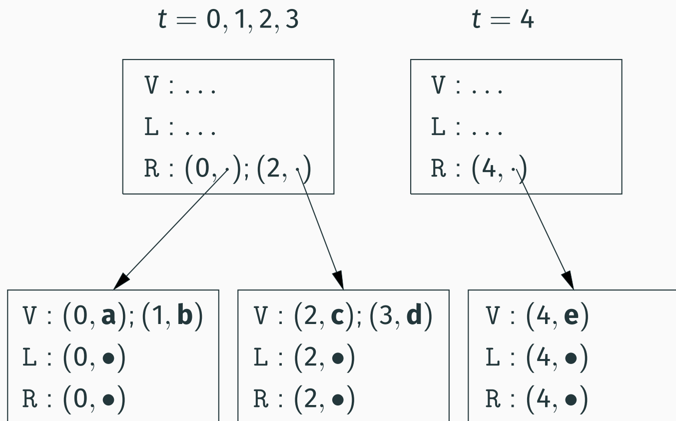
Example of updates with histories of size $k = 2$



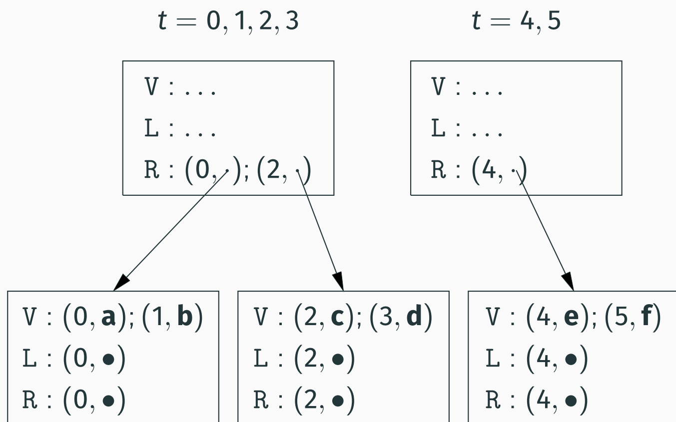
Example of updates with histories of size $k = 2$



Example of updates with histories of size $k = 2$



Example of updates with histories of size $k = 2$



Each update to the V field of the right child takes, on an average, 2 updates or block creations $\rightarrow$ time and space $\mathcal{O}(1)$ amortized.

Amortized analysis (Sarnak and Tarjan, 1986)

In the Sarnak-Tarjan model, each fat node has `3` fields `V`, `L`, `R` containing their most recent values, and k "generic" history slots containing earlier values tagged `V`, `L`, `R`, along with their dates.

The potential for the data structure is

$$\Phi = \text{number of nodes} - \frac{\text{number of free history slots}}{k}$$

Allocation of a new node or copy of an existing node:
number of nodes $+1$, number of slots $+k$, hence $\Delta\Phi = 0$.

Writing without node copying: $\Delta\Phi = 1/k$.

For each BST insertion, $\Delta\Phi$ is bounded by a constant, and the amortized space is $\mathcal{O}(1)$.

Towards full persistence

Three main changes:

1. A field update can require two additions to its history, as we saw for O'Neill and Burton's arrays.
2. When the history is full, instead of creating a new, empty node, we must split the node in two half-full nodes.
3. We also need to version the balancing information (red/black colors). Up to $\log n$ colors can change at each insertion.

   (This bothers Driscoll, Sarnak, Sleator and Tarjan (1989) so much that they outline a lazy recoloring algorithm that avoids this problem.)

Making an ephemeral structure persistent

Making Data Structures Persistent

A famous article by Driscoll, Sarnak, Sleator and Tarjan (1989) that shows how to start from

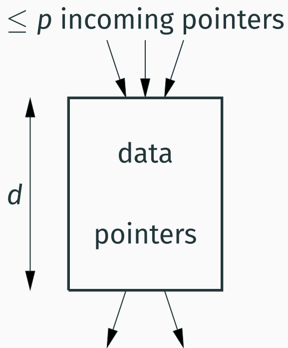
an almost-arbitrary ephemeral data structure
(a directed graph with bounded-degree vertices)

and transform it into

a structure with the same operations,
but partially or fully persistent.

Same ingredients as for red-black trees:
fat nodes with fixed-size histories
+ node copying or node splitting.

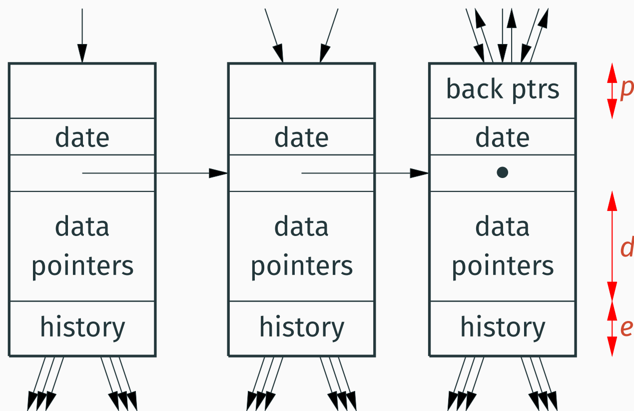
A node of the ephemeral structure



Each node contains d fields.

Each field is either raw data or a pointer to another node.

The corresponding fat nodes (partial persistence case)



Histories contain up to e entries. Each entry is a triple (field name, date, new value).

Only the last fat node of the chain can be modified and contains back pointers.

Writing to a field

Same approach as in the partially-persistent red-black tree example:

- If the history is not full, add an entry to it.
- Otherwise, create a new node and write a new pointer in the parent nodes.

Main difference: we must use back pointers (to locate the parent nodes) and update them too.

Writing to a field

Writing value v in field f of fat node x at date i:

- If x already contains a value for f at date i, update it.
- Otherwise, if the history is not full, add the entry (f, i, v).
- Otherwise, create a new node y, with empty history, field f initialized to v, other fields initialized to the values they have in x at date $i - 1$.
- In the latter case, use the back pointers of x to update incoming pointers: if field g of node z points to x at date $i - 1$, write to $z.g$ a pointer to y at date i.
- In all cases, if f has pointer type, update the back pointers of the blocks pointed by f at dates $i - 1$ and i.

Amortized analysis

Each write takes amortized time $\mathcal{O}(1)$, and therefore amortized space $\mathcal{O}(1)$, provided that the histories are large enough: $e \geq p$.

Intuition: each time we copy a node, we gain e free history entries, which compensate for the p updates of incoming pointers that are required.

Extension to full persistence

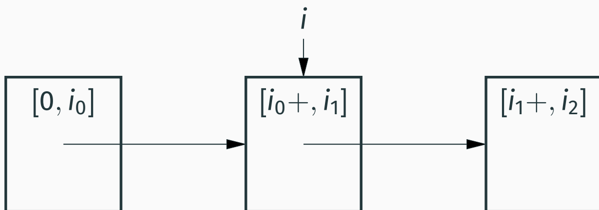
Three main changes:

1) Since we can write to any version, we need back pointers for all versions, not just the latest.
$\rightarrow$ Back pointers are “versioned” like any other pointer.

2) A write to a field can require two insertions in its history.

3) Instead of copying a full node, obtaining a full node + an empty node, we split a full node, obtaining two half-full nodes.

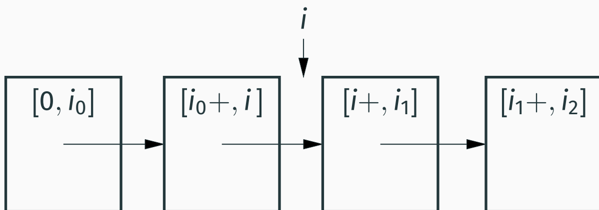
Chaining and splitting nodes



Each node of the ephemeral structure corresponds to a linked list of fat nodes, each being valid for an interval of dates.

Splitting a node at date i inserts a new node containing the history from date $i+$, leaving the history up to date i in the old node.

Chaining and splitting nodes



Each node of the ephemeral structure corresponds to a linked list of fat nodes, each being valid for an interval of dates.

Splitting a node at date i inserts a new node containing the history from date $i+$, leaving the history up to date i in the old node.

Updating pointers

Splitting a node can invalidate both forward pointers and back pointers! We need a rather complicated iterative algorithm:

- Identify pointers that have become incorrect
  ($\approx$ pointers whose validity interval is no longer included in the validity interval of the pointed node).
- Rewrite these pointers so that they point to the correct version of the node.
- These writes can trigger further node splits.

Writes still take $\mathcal{O}(1)$ amortized time and space,
provided that $e \geq d + p$.

The order maintenance problem

The order maintenance problem

Consider an ordered list with two operations:

- Given two list elements, determine their relative poisition: which element precedes the other in list order.
- Insert a new element just after a given element.

We'd like both operations to run in constant amortized time.

A naive implementation

A circular, mutable, linked list of cells c with two fields:

- $L(c)$: an integer label in $[0, M)$ with $M = 2^{128}$ for instance.
- $S(c)$: pointer to the next cell.

Determine the relative position of cells c_1 and c_2:

Compare their labels $L(c_1), L(c_2)$.

Insert a new element just after cell c:

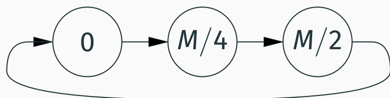
If $L(S(c)) \geq L(c) + 2$, label the new cell

with the average of $L(c)$ and $L(S(c))$.

Otherwise, renumber cells in the neighborhood of c

and try again.

A naive implementation



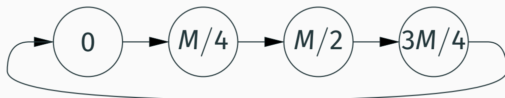
A circular, mutable, linked list of cells c with two fields:

- $L(c)$: an integer label in $[0, M)$ with $M = 2^{128}$ for instance.
- $S(c)$: pointer to the next cell.

Determine the relative position of cells c_1 and c_2:
Compare their labels $L(c_1), L(c_2)$.

Insert a new element just after cell c:
If $L(S(c)) \geq L(c) + 2$, label the new cell
with the average of $L(c)$ and $L(S(c))$.
Otherwise, renumber cells in the neighborhood of c
and try again.

A naive implementation



A circular, mutable, linked list of cells c with two fields:

- $L(c)$: an integer label in $[0, M)$ with $M = 2^{128}$ for instance.
- $S(c)$: pointer to the next cell.

Determine the relative position of cells c_1 and c_2:
Compare their labels $L(c_1), L(c_2)$.

Insert a new element just after cell c:
If $L(S(c)) \geq L(c) + 2$, label the new cell
with the average of $L(c)$ and $L(S(c))$.
Otherwise, renumber cells in the neighborhood of c
and try again.

The renumbering criterion of Dietz and Sleator (1987)

Define the gap between two cells c_1, c_2 as

$$g(c_1, c_2) \stackrel{def}{=} (L(c_2) - L(c_1)) \bmod M$$

If we need to insert after c and $g(c, S(c)) = 1$:

Search the list ahead of c to find an interval of (minimal) width j such that $g(c, S^j(c)) > j^2$.

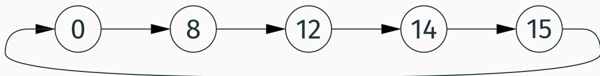
(It's always possible if the list contains at most $\sqrt{M} = 2^{64}$ elements.)

Renumber the j cells in the interval, spacing them equally:

$$L(S^k(c)) := \left(L(c) + \left\lfloor \frac{k}{j} \times g(c, S^j(c)) \right\rfloor\right) \bmod M$$

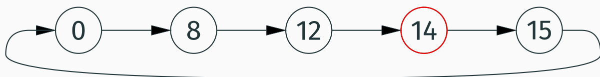
An example of renumbering

We take $M = 16$, and we insert after element 14.



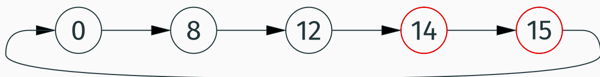
An example of renumbering

We take $M = 16$, and we insert after element 14.



An example of renumbering

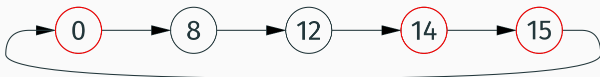
We take $M = 16$, and we insert after element 14.



For $j = 1$, the gap is 1, too small.

An example of renumbering

We take $M = 16$, and we insert after element 14.

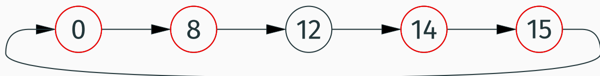


For $j = 1$, the gap is 1, too small.

For $j = 2$, the gap is 2, too small.

An example of renumbering

We take $M = 16$, and we insert after element 14.



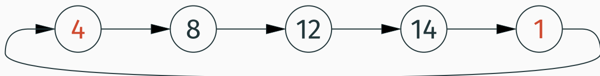
For $j = 1$, the gap is 1, too small.

For $j = 2$, the gap is 2, too small.

For $j = 3$, the gap is 10, that's enough!

An example of renumbering

We take $M = 16$, and we insert after element 14.



For $j = 1$, the gap is 1, too small.

For $j = 2$, the gap is 2, too small.

For $j = 3$, the gap is 10, that's enough!

Renumbering: $14 + 3 = 1$, $14 + 2 \times 3 = 4$.

Renumbering and relative cell positions

Renumberning can “wrap around” the list and change the labels of the first elements. Hence, we need a different way to order cells.

Determine the relative position of cells c_1 and c_2:

Compare $g(base, c_1)$ with $g(base, c_2)$ (the gaps with the base).

Renumbering preserves relative positions!

	Labels					Gaps with the base				
Before renumbering	0	8	12	14	15	0	8	12	14	15
After renumbering	4	8	12	14	1	0	4	8	10	13

Amortized analysis; a two-level representation

(P. F. Dietz, D. Sleator, *Two Algorithms for Maintaining Order in a List*, STOC 1987).

A complicated analysis shows that insertion takes time $\mathcal{O}(\log n)$, where n is the size of the list.

We can achieve $\mathcal{O}(1)$ amortized using a two-level representation:

- Each element of the circular list points to a sub-list of integers with at least $\lceil \log n \rceil$ bits, e.g. 64 bits, containing at most $\log n$ elements.
- Insertion in sub-lists does not renumber, it just takes the average of the two labels.
- When insertion in a sub-list is not possible, we split it in two halves, renumber these halves, and insert the second half in the main list.

Summary

Making an ephemeral structure persistent

An approach initiated by seminal work of the 1980's.

It lacks the algebraic elegance of the purely functional approach to persistence, but can be more efficient in specific use cases:

- partial persistence;
- semi-persistence; ($\rightarrow$ J. C. Filliâtre's seminar)
- quasi-linear use.

Full persistence is (excessively?) complicated to obtain.

The approach is interesting not so much to reduce time but to reduce space ($\mathcal{O}(n \log n) \rightarrow \mathcal{O}(n)$).

References

References

A dense but complete overview of the approach:

- H. Kaplan, *Persistent Data Structures*, chap. 31 of *Handbook of data structures and applications*, Chapman&Hall / CRC Press, 2005.

The main articles discussed in this lecture:

- M. E. O'Neill and F. W. Burton, *A new method for functional arrays*, J. Func. Prog. 7(5), 1997.
- N. Sarnak and R. E. Tarjan, *Planar Point Location using Persistent Search Trees*, Commun. ACM 29(7), 1986.
- J. Driscoll, N. Sarnak, D. Sleator and R. E. Tarjan, *Making Data Structures Persistent*, J. Comput. Syst. Sci. 38(1), 1989.
- P. Dietz and D. Sleator, *Two Algorithms for Maintaining Order in a List*, proceedings of STOC 1987, ACM.