Nothing is lost, everything is created: introduction to persistent data structures

Xavier Leroy

2023-03-09

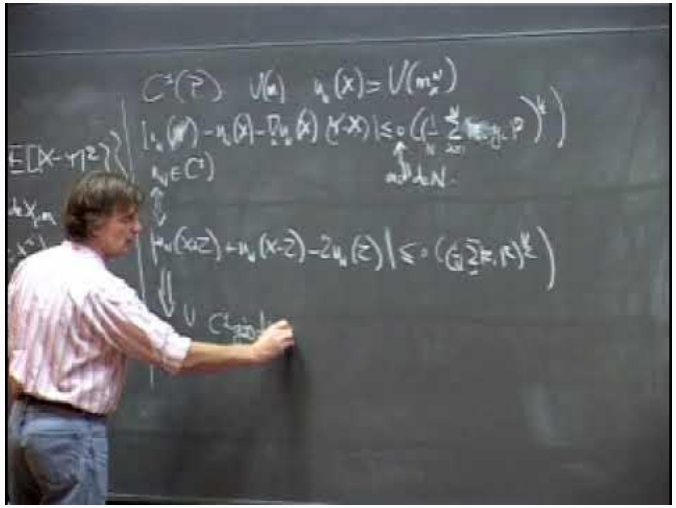
Collège de France, chair of Software sciences
xavier.leroy@college-de-france.fr

An algorithm is like a recipe!

Obvious consequence: input data and intermediate results are not preserved during computation.

Fundamental formalism: Turing machines.

An algorithm is like a mathematical definition



Computation is seen as a succession of steps that build the final result, without destroying input data nor intermediate results.

Fundamental formalisms: general recursive functions, lambda-calculus, rewriting systems, formal semantics.

“Declarative” (as opposed to “imperative”) programming: functional programming, logic programming, constraint programming, …

Which algorithms for declarative programming?

Declarative programming is often said to be inefficient because it cannot use ephemeral data structures (with in-place updates), such as arrays.

The persistent data structures that we will study in this course address this criticism:

- Their interfaces expose only operations that do not update structures in place, but return new, updated structures.
- Their implementations match (or get close to) the complexity of the best known ephemeral structures.

Persistence matters for imperative programming too!

Even for imperative programs and classical algorithms, persistent data structures are useful:

- They make it easy to checkpoint and backtrack computations.
- We can keep the full history of the data structure.
- The lack of in-place updates enables memory sharing among the various versions of the structure, hence a compact representation of its history.

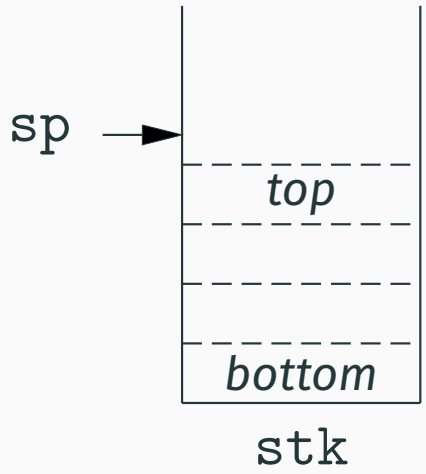
First example: stacks

A stack

"Last-in first-out" operations:

`init`	empty the stack
`push`(v)	push v on top of the stack
`top`	return the top value
`pop`	pop the top value.

An ephemeral stack implemented as an array

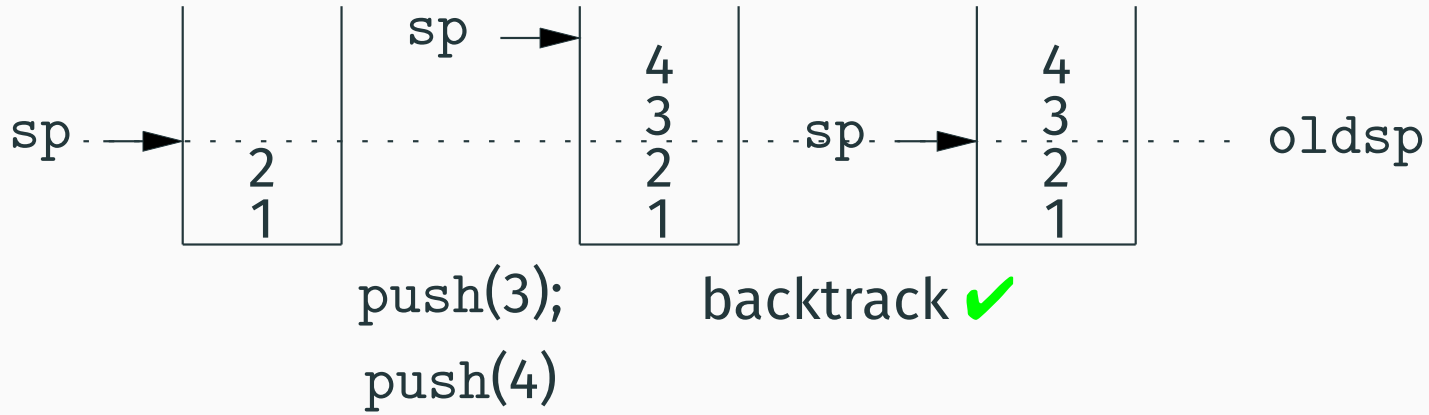



```
int stk[SIZE];
int sp;

void init(void) { sp = 0; }

void push(int v) {
  assert (sp < SIZE);
  stk[sp] = v; sp = sp + 1;
}
int top(void) {
  assert (sp > 0); return stk[sp - 1];
}
void pop(void) {
  assert (sp > 0); sp = sp - 1;
}
```

Checkpointing and backtracking

Cheap but incomplete approach:

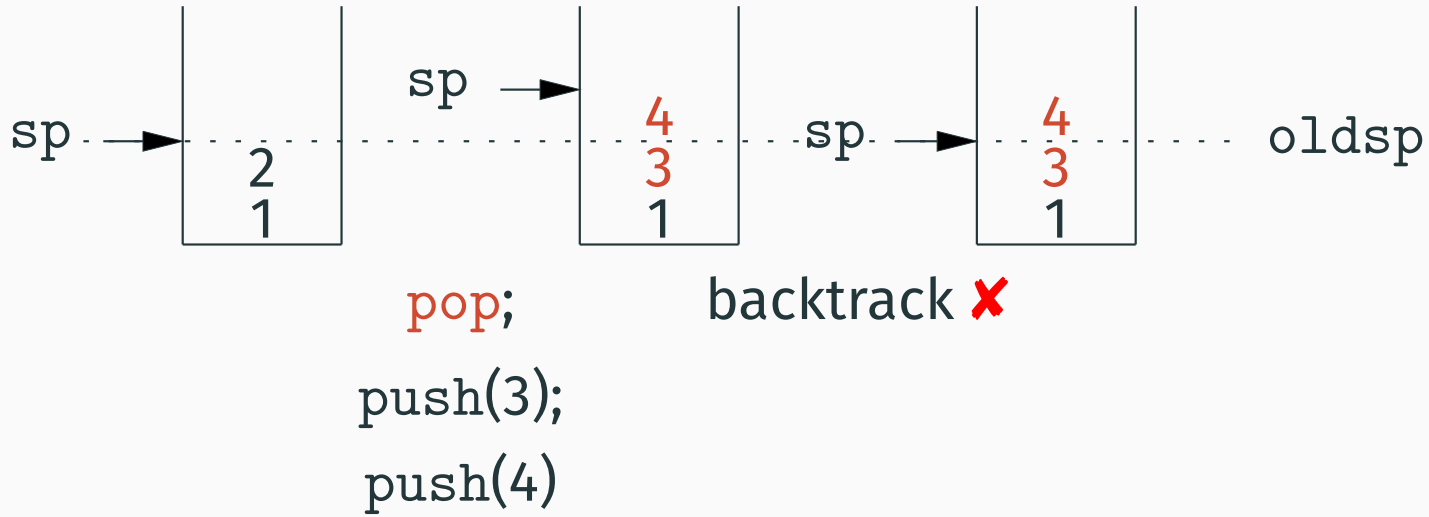
- Setting a checkpoint: `oldsp = sp;` (save `sp`)
- Backtracking: `sp = oldsp;` (restore `sp`)



Checkpointing and backtracking

Cheap but incomplete approach:

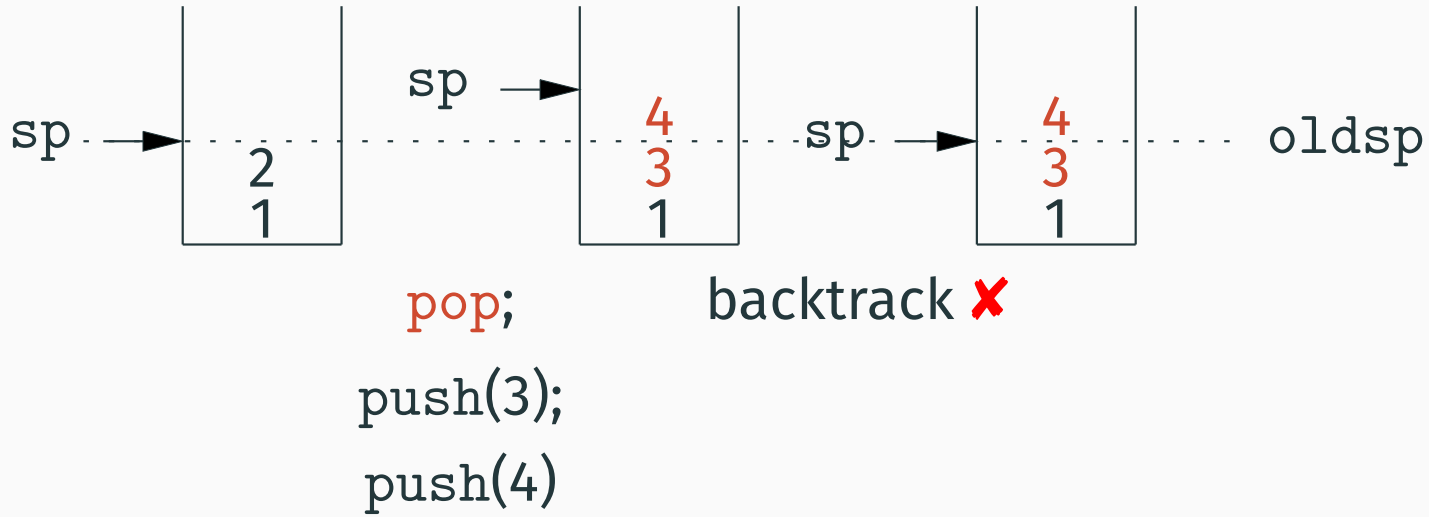
- Setting a checkpoint: `oldsp = sp;` (save `sp`)
- Backtracking: `sp = oldsp;` (restore `sp`)



Checkpointing and backtracking

Cheap but incomplete approach:

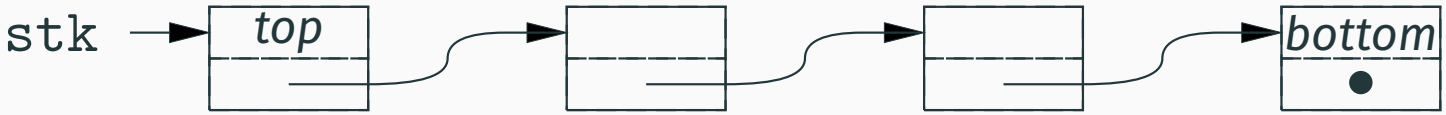
- Setting a checkpoint: `oldsp = sp;` (save `sp`)
- Backtracking: `sp = oldsp;` (restore `sp`)



Correct but expensive approach:

copy `stk`[0 . . . `sp`[to/from another array.

A persistent stack implemented by a linked list

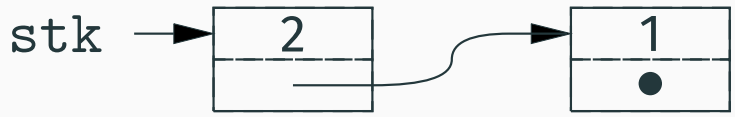



```
class Stack {
  private int hd; private Stack tl;
  static Stack empty = null;
  static Stack push(int v, Stack s)
  { Stack t = new Stack(); t.hd = v; t.tl = s; return t; }
  static int top(Stack s) { return s.hd; }
  static Stack pop(Stack s) { return s.tl; }
}
```

Change of interface: now, `push`, `pop` return the new stack as a result instead of modifying the stack given as argument.

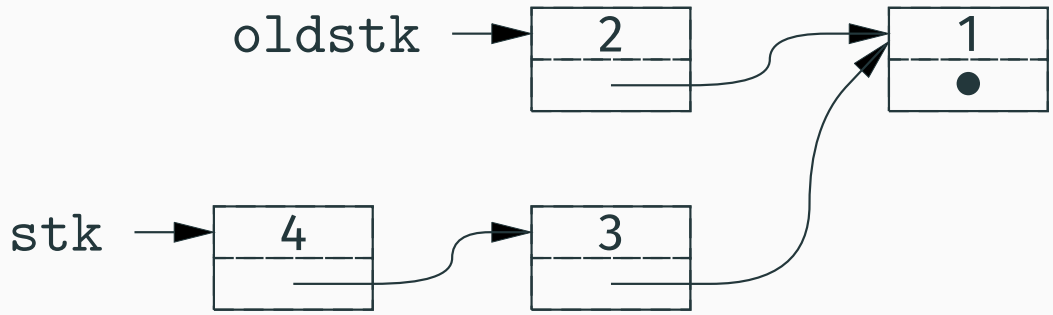
We rely on garbage collection to reclaim list cells that become unreachable after `pop`.

Persistent stacks and backtracking



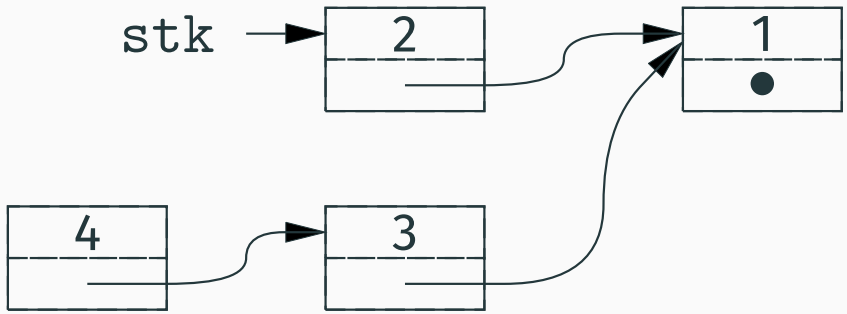

```
stk = Stack.push(2, Stack.push(1, Stack.empty));
```

Persistent stacks and backtracking




```
stk = Stack.push(2, Stack.push(1, Stack.empty));
// Checkpoint
oldstk = stk;
// Work
stk = Stack.push(4, Stack.push(3, Stack.pop(stk)));
```

Persistent stacks and backtracking




```
stk = Stack.push(2, Stack.push(1, Stack.empty));
// Checkpoint
oldstk = stk;
// Work
stk = Stack.push(4, Stack.push(3, Stack.pop(stk)));
// Backtrack
stk = oldstk;
```

Pure functional implementation of persistent stacks

In Lisp, Scheme, etc: using primitive operations over lists.

`empty` $\equiv$ `nil` `push` $\equiv$ `cons` `top` $\equiv$ `car` `pop` $\equiv$ `cdr`

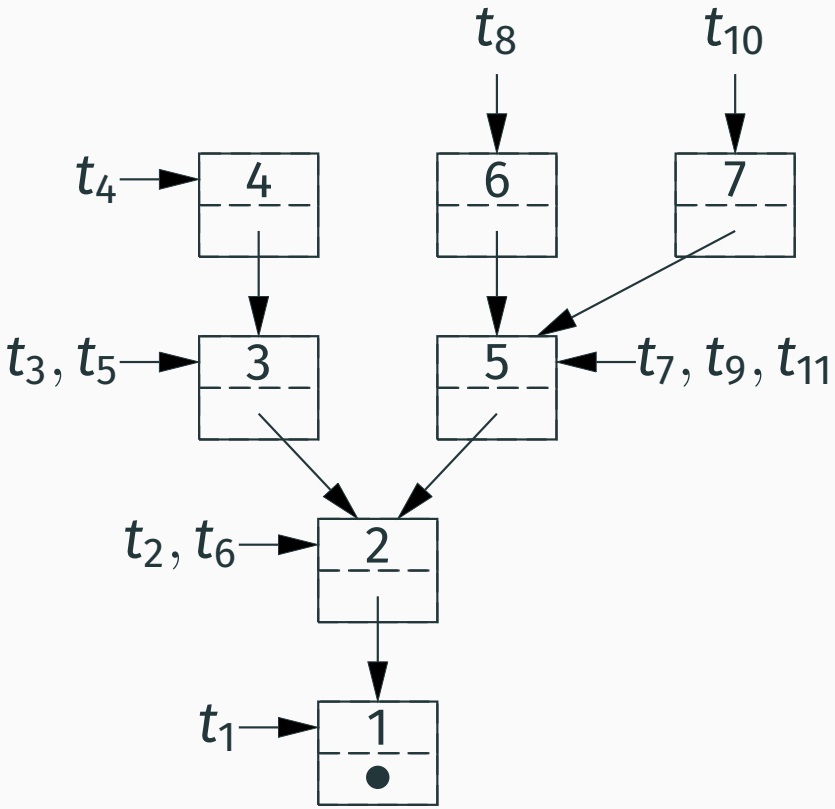
In OCaml, Haskell, etc: using an algebraic type.

```
type 'a stack = Empty | Stack of 'a * 'a stack
let empty = Empty
let push v s = Stack(v,s)
let top = function Stack(v,_) -> v | _ -> assert false
let pop = function Stack(_,s) -> s | _ -> assert false
```

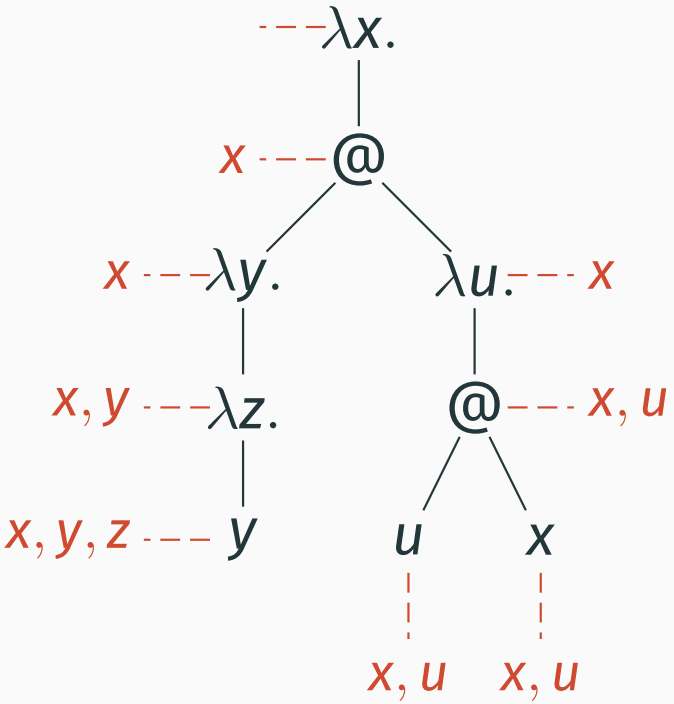
Persistent stacks and memory sharing

A stack produced by `push` or `pop` shares all memory blocks except one with the previous stack. This makes it possible to keep all successive states of the stack in a memory-efficient way: N blocks for any sequence of N `push` and M `pop`.

$$
\begin{aligned}
t_1 &= \mathtt{push}(1, \mathtt{empty}) \\
t_2 &= \mathtt{push}(2, t_1) \\
t_3 &= \mathtt{push}(3, t_2) \\
t_4 &= \mathtt{push}(4, t_3) \\
t_5 &= \mathtt{pop}(t_4) \\
t_6 &= \mathtt{pop}(t_5) \\
t_7 &= \mathtt{push}(5, t_6) \\
t_8 &= \mathtt{push}(6, t_7) \\
t_9 &= \mathtt{pop}(t_8) \\
t_{10} &= \mathtt{push}(7, t_9) \\
t_{11} &= \mathtt{pop}(t_{10})
\end{aligned}
$$



Application: annotating an AST with environments



Annotate each node of an abstract syntax tree with its environment, that is, the set of variables in scope at this point.

Environment ≈ stack
Entering a variable scope ≈ push
Leaving a variable scope ≈ pop

	Arrays	Lists	BSTs (→ 2nd lecture)
Sharing	none	maximal	high
Total space	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Lookup time	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Algebraic specifications

In mathematics, an algebraic structure is a set equipped with operations that satisfy identities (equations).

Example: a group is a set G with three operations:
a constant 1, a binary operation $\cdot$, a unary operation $^{-1}$,
verifying the identities

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$1 \cdot x = x = x \cdot 1$$
$$x \cdot x^{-1} = 1 = x^{-1} \cdot x$$

Algebraic specifications

(Guttag and Horning, *The Algebraic Specification of Abstract Data Types*, 1978.)

In computer science, an algebraic abstract type is an abstract type (= type name + operations) specified by equations involving the operations.

Example: stacks (operations $\texttt{empty}$, $\texttt{push}$, $\texttt{pop}$, $\texttt{top}$)

$$\texttt{top}\,(\texttt{push}(v, s)) = v$$

$$\texttt{pop}\,(\texttt{push}(v, s)) = s$$

If we add the $\texttt{enqueue}$ operation (insertion at the bottom of the stack):

$$\texttt{enqueue}(v, \texttt{empty}) = \texttt{push}(v, \texttt{empty})$$

$$\texttt{enqueue}(v, \texttt{push}(v', s)) = \texttt{push}(v', \texttt{enqueue}(v, s))$$

Algebraic specifications and persistence

$$\texttt{top}\ (\texttt{push}(v, s)) = v$$

$$\texttt{pop}\ (\texttt{push}(v, s)) = s$$

This equational specification style assumes a persistent interface for the abstract type: the `push`, `pop` operations produce new stacks, they do not modify (observably) any existing stack.

Algebraic specifications and persistence

$$\texttt{top}\ (\texttt{push}(v, s)) = v$$

$$\texttt{pop}\ (\texttt{push}(v, s)) = s$$

For an ephemeral structure (such as the array-based stack), we lose equations. At best we have program equivalences:

$$\begin{aligned} \texttt{push}(v);\ \texttt{pop}() &\approx \texttt{skip} \\ \texttt{push}(v);\ x := \texttt{top}() &\approx x := v;\ \texttt{push}(v) \end{aligned}$$

plus commutation rules with commands that do not depend on the state of the stack.

Algebraic specifications and functional implementations

$$\begin{aligned}
\texttt{top}\,(\texttt{push}(v, s)) &= v \\
\texttt{pop}\,(\texttt{push}(v, s)) &= s \\
\texttt{enqueue}(v, \texttt{empty}) &= \texttt{push}(v, \texttt{empty}) \\
\texttt{enqueue}(v, \texttt{push}(v', s)) &= \texttt{push}(v', \texttt{enqueue}(v, s))
\end{aligned}$$

It is easy to check that a pure functional implementation satisfies these equations. Taking the OCaml implementation as example, once definitions are expanded, it remains to show

```
(match Stack(v,s) with Stack(v,_) -> v | _ -> assert false) = v
(match Stack(v,s) with Stack(_,s) -> s | _ -> assert false) = s
```

This follows from the operational semantics of `match...with`.

Algebraic specifications and functional implementations

$$\texttt{top}\,(\texttt{push}(v, s)) = v$$

$$\texttt{pop}\,(\texttt{push}(v, s)) = s$$

$$\texttt{enqueue}(v, \texttt{empty}) = \texttt{push}(v, \texttt{empty})$$

$$\texttt{enqueue}(v, \texttt{push}(v', s)) = \texttt{push}(v', \texttt{enqueue}(v, s))$$

Symmetrically, we can often derive a functional implementation from the equations. Taking `enqueue` as an example: P.ex. pour `enqueue`:

```
let rec enqueue v s =
  match s with
  | Empty -> Stack(v, Empty)
  | Stack(v', s) -> Stack(v', enqueue v s)
```

The emergence of persistent data structures

Data structures = data + structural relationships

Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important structural relationships between the data elements.

In order to use a computer properly, we need to understand the structural relationships present within data, as well as the basic techniques for representing and manipulating such structure within a computer.

D. E. Knuth, The Art of Computer Programming, vol 1, chap 2, "Information structures", 1968.

1945–1960: The prehistory of data structures

Data stored in arrays, either as “amorphous masses of numerical value”, or with a bit of structure:

- sorted array + binary search; (Mauchly, 1946)
- hash table; (A. Dumey et al, 1956–)
- “pointers” from one array to another
  (in early databases and knowledge bases).

1960–1970: a concept emerges; first breakthroughs

A data structure = an interface (set of operations)
with several implementations possible.

- Stacks, queues.
- Dictionaries, implemented using search trees:
  not balanced, (Windley et al, 1960; many others)
  self-balancing. (Adelson-Velskii et Landis, 1962; many others)
- Priority queues, implemented as heaps (Williams, 1964)

A program =

abstract algorithms

\+ appropriate, efficient data structures

(that can involve subtle algorithms themselves)

An abstract algorithm: Dijkstra's shortest path algorithm

> *In the course of the solution the nodes are subdivided into three sets [...] Consider all branches connecting the node just transferred to set A with nodes R in sets B or C [...] the node with minimum distance from P is transferred from set B to set A [...]*
>
> *(E.W. Dijkstra, A note on two problems in connexion with graphs, 1959.)*

The concrete implementation of those sets is left to the reader, as well as the efficient way to find "the node with minimal distance from *P*" among the nodes in *B*.

This efficient way came 4 years later: the heap data structure, invented by Williams in 1962.

Since 1970: the modern times of data structures

Systematic exploration guided by new needs and by new approaches to algorithm design:

- multi-dimensional structures: geometric algorithms, databases, …
- strings and pattern search;
- lock-free data structures for concurrency;
- randomized data structures.

Major developments in analysis of algorithms: worst case, average analysis, amortized analysis, expected-time analysis, …

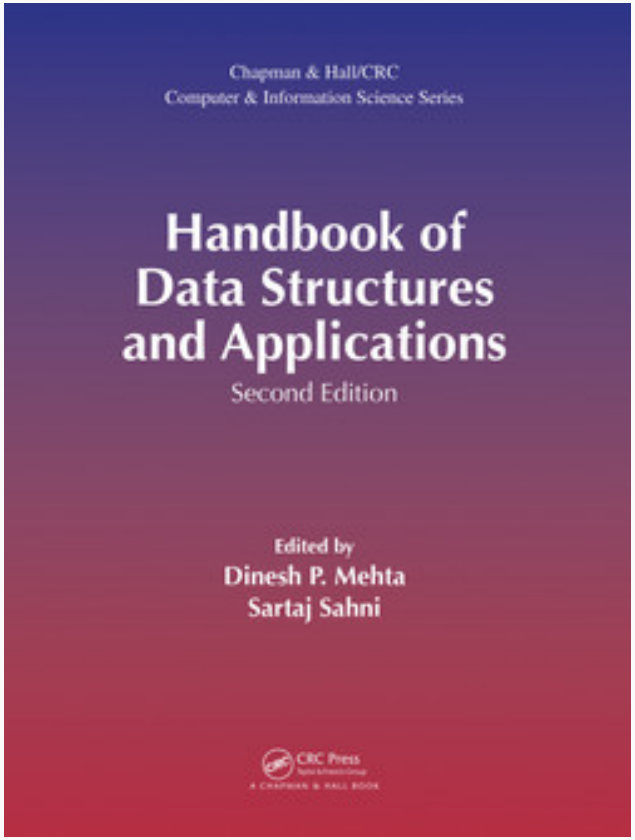
(See also: the lectures by B. Chazelle, J.-D. Boissonnat, C. Mathieu, R. Guerraoui and F. Magniez on the “chaire annuelle d’informatique et de sciences numériques”.)

Ephemeral structures and imperative programming

Just like most algorithms are presented like recipes, and most programs are written in imperative style, most data structures are ephemeral:

> *Operations on the structure (e.g. insertion in a dictionary) can modify in place the state of the structure, rendering unavailable the state of the structure before the operation.*

Handbook of Data Structures and Applications (CRC, 2005, 2018)



Out of 64 chapters, only 2 discuss data structures that are not ephemeral:

31. *Persistent data structures*
(Haim Kaplan)

40. *Functional data structures*
(Chris Okasaki)

Persistent data structures

> *All operations preserve the current state of the structure.*
> *If the structure needs updating, a "new" structure is produced and returned by the update operation.*
> *In other words: operations are presented like pure, side effect-free functions.*

Note: the implementation of the operations can use imperative features (e.g. mutable arrays).

Persistent data structures emerged in the 1980's in the context of computer graphics, where it is often convenient to have access to the full history of a data structure.
(Often called "searching in the past" or "in-the-past queries".)

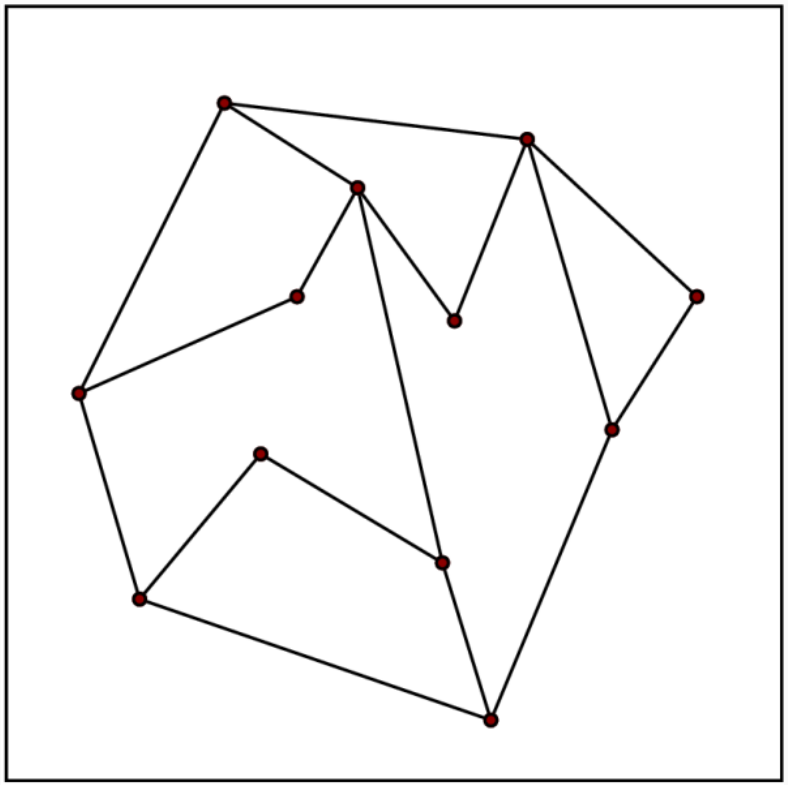
Purely functional data structures

These are persistent structures whose implementation uses no imperative features (no in-place updates) and can be written in a pure functional language.

Emerged in the 1990's to enable the use of efficient algorithms in purely functional programming.

Advanced example: planar point location

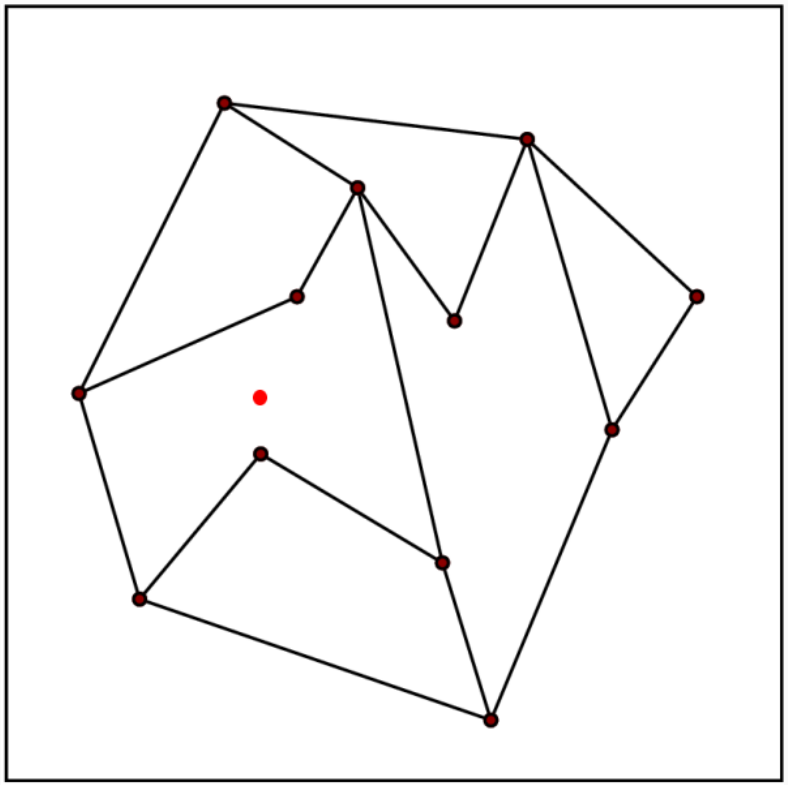
The planar point location problem



(Gfonsecabr, English Wikipedia)

Given line segments defining polygons, and k points $P_1, \ldots, P_k$, find quickly which polygon contains each point.

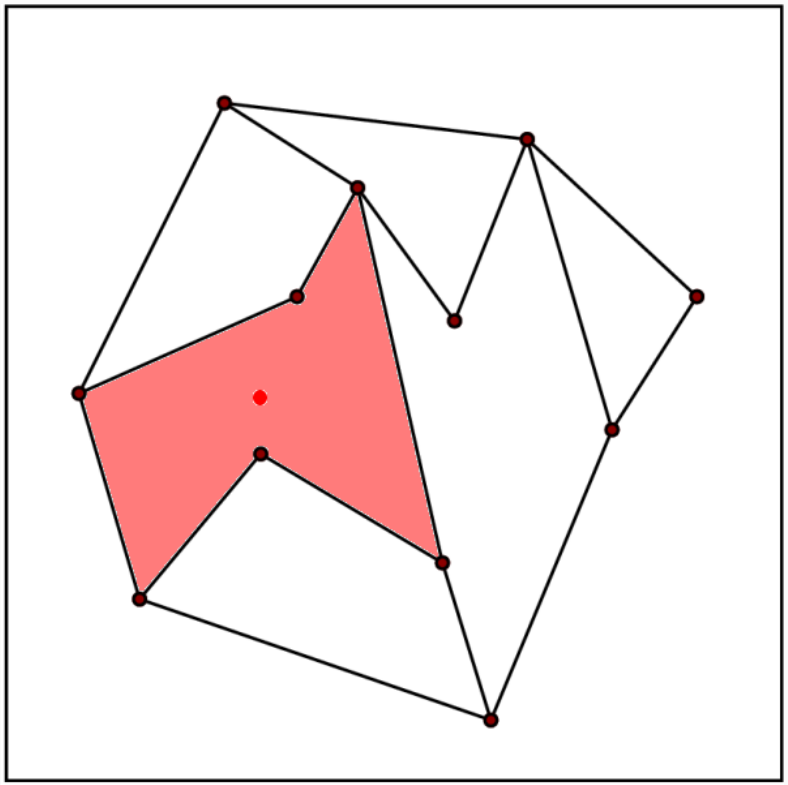
The planar point location problem



(Gfonsecabr, English Wikipedia)

Given line segments defining polygons, and k points $P_1, \ldots, P_k$, find quickly which polygon contains each point.

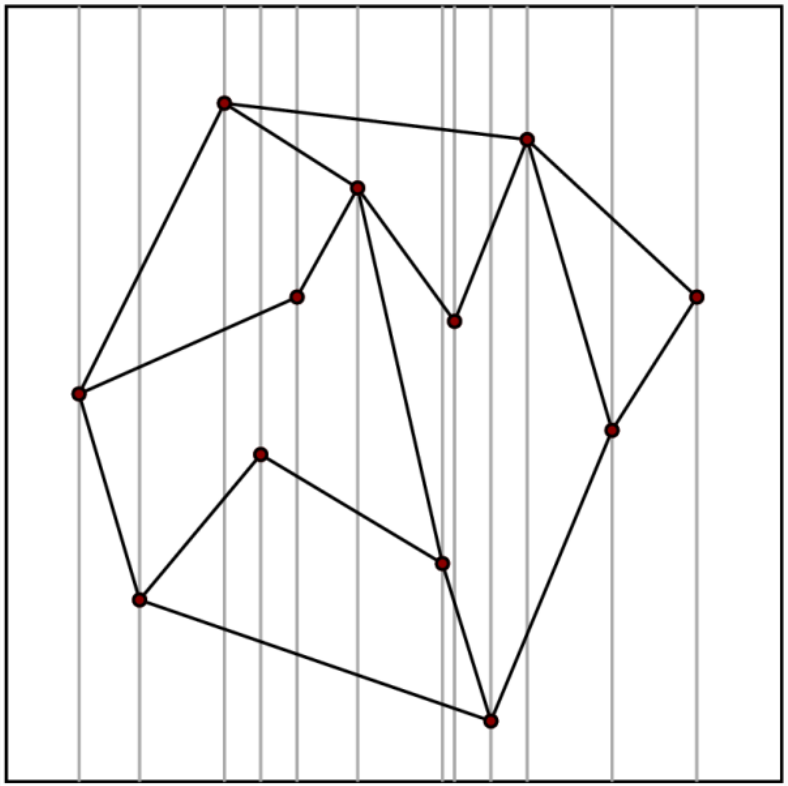
The planar point location problem



(Gfonsecabr, English Wikipedia)

Given line segments defining polygons, and k points $P_1, \ldots, P_k$, find quickly which polygon contains each point.

Partitioning in slabs (Dobkin et Lipton, 1976)



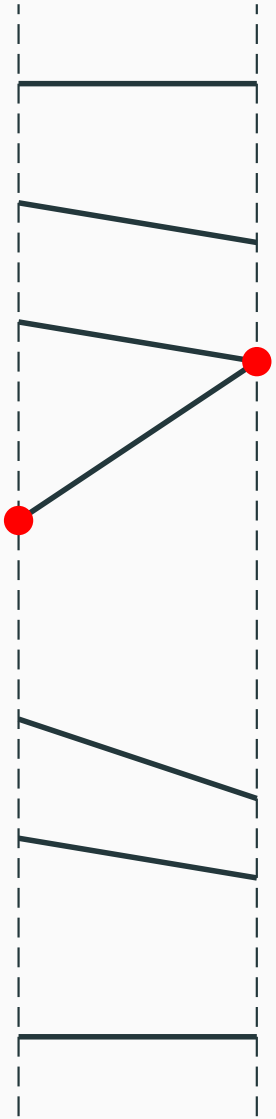
We sort the x coordinates of the segment endpoints.

This partitions the plane in vertical slabs.

Within a slab, the number of segments is constant.

Segments “enter” and “leave” at the frontier between two slabs.

Binary search within a slab



Inside a slab, segments do not intersect each other.
We can therefore sort the segments by vertical position, from lowest to highest.

Given a point P, binary search determines quickly the two segments S_i, S_j just above and just below P.

This suffices to identify the polygon that contains P.

Dobkin and Lipton's algorithm

Preprocessing of the n segments:

1. Sort the x coordinates of the segment endpoints $\rightarrow \mathcal{O}(n)$ slabs.
2. For each slab, build an array with the $\mathcal{O}(n)$ segments contained in the slab, and sort them by vertical position.

Space: $\mathcal{O}(n^2)$.

For each point $P = (x, y)$:

1. Find the slab containing P by binary search over x. sur x.
2. Find the two segments above and below P by binary search in the slab.

Time: $\mathcal{O}(\log n)$.

Reducing time and space for preprocessing

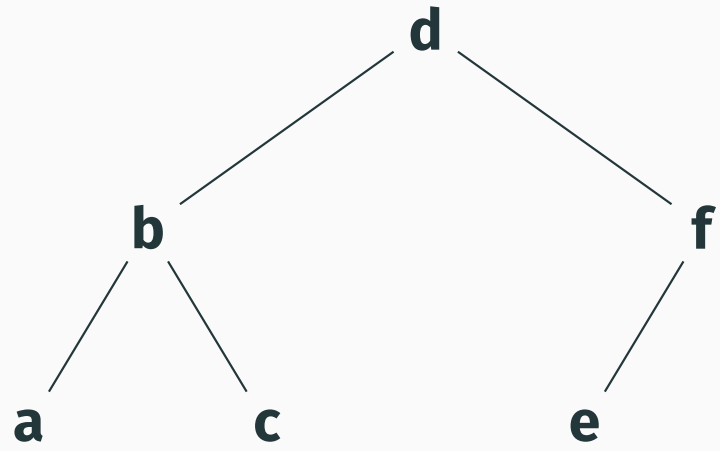
Each of the n segments "enters" and "leaves" a slab exactly once.

Therefore, two successive slabs share most of their segments, with the same relative vertical positions.

Idea: represent each slab no longer as a sorted array, but as a persistent structure with lookups in $\mathcal{O}(\log n)$ and memory sharing between successive versions.

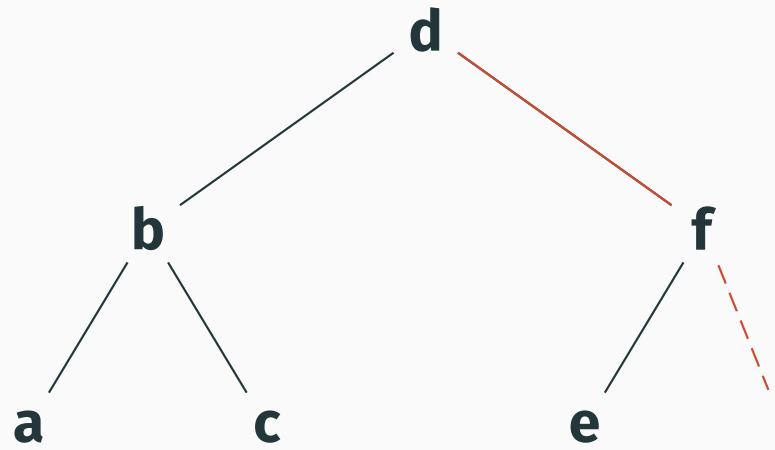
An appropriate data structure: a persistent balanced binary search tree (AVL tree, red-black tree, etc) ($\rightarrow$ 2nd lecture)

Persistent insertion in a binary search tree (BST)



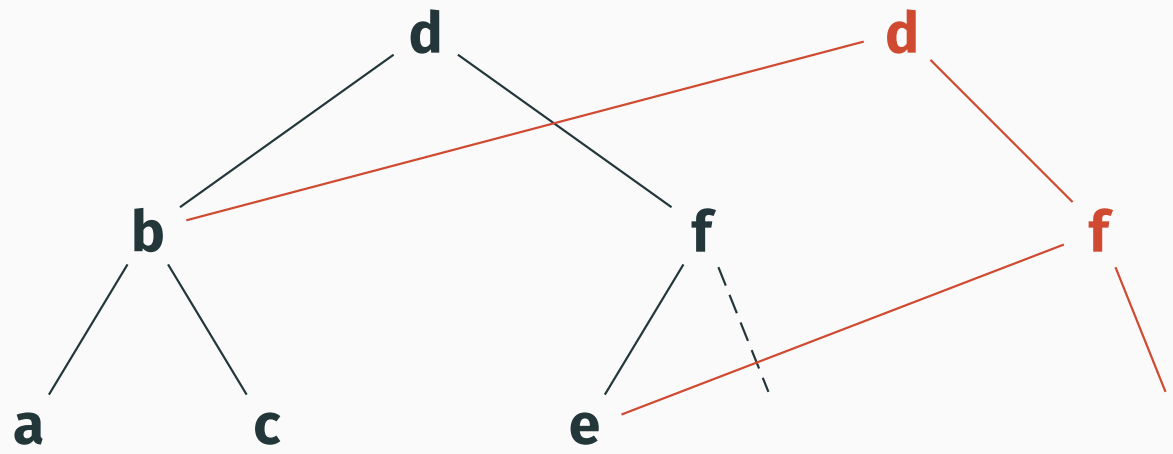
1. Search for the element to be inserted (here, **g**).

Persistent insertion in a binary search tree (BST)



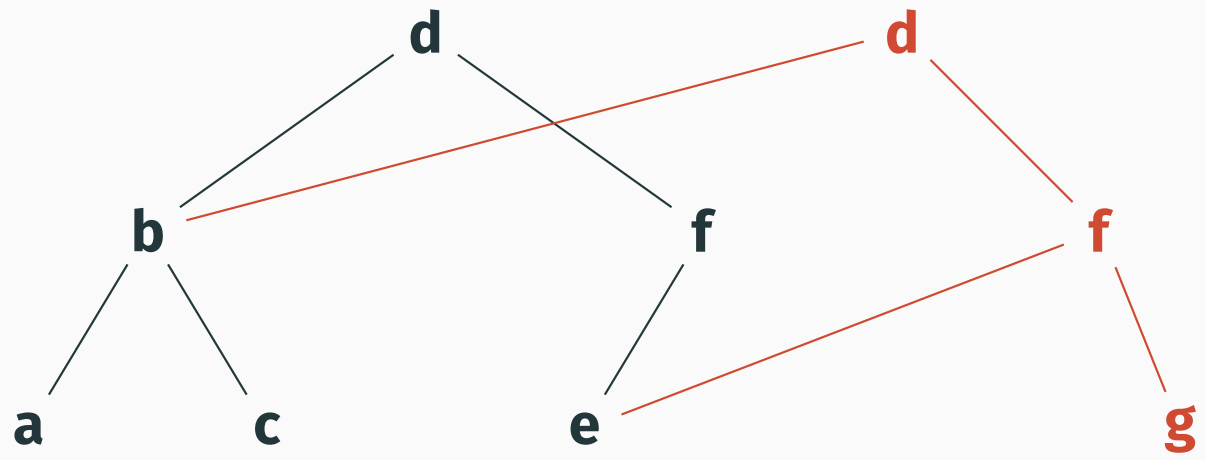
1. Search for the element to be inserted (here, **g**).

Persistent insertion in a binary search tree (BST)



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 original tree.

Persistent insertion in a binary search tree (BST)



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 original tree.
3. At the end of the copied path, add the node $\langle \bullet, \mathbf{g}, \bullet \rangle$.
4. If needed, rebalance the tree, preserving sharing as much as possible.

Time: $\mathcal{O}(\log n)$, space: $\mathcal{O}(\log n)$.

Improved algorithm for planar point location

Preprocessing of the n segments:

1. Scan segment endpoints by increasing x coordinates.
2. Enter / remove the segments in a persistent balanced BST, sorting segments by increasing vertical position.
3. Keep the intermediate states of the BST (= the slabs) in an array.

Time and space: $\mathcal{O}(n \log n)$.

(Each of the n segments enters and leaves the BST once, in time $\mathcal{O}(\log n)$.)

For each point $P = (x, y)$, binary search in the array of slabs, then in the BST corresponding to the slab.
Time: $\mathcal{O}(\log n)$.

Sarnak and Tarjan's algorithm

Sarnak and Tarjan (*Planar Point Location using Persistent Search Trees*, CACM, 1986) show how to reduce the space used by preprocessing from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$ by using a different implementation of persistent BSTs.

This implementation uses in-place mutations and relies on the fat nodes technique of Driscoll, Sarnak, Sleator, et Tarjan (1989).

It's a general technique to transform an ephemeral structure into a persistent structure: replace each field of each node by a journal of modifications of this field, i.e. a set of (modification date, new value) pairs.

→ 4th lecture

Example of a BST with fat nodes

Ephemeral BST:

t_0

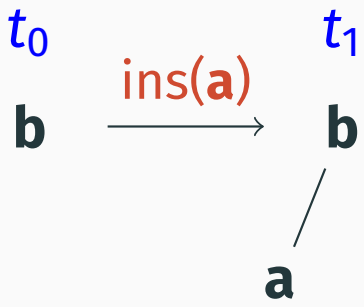
b

Persistent BST:

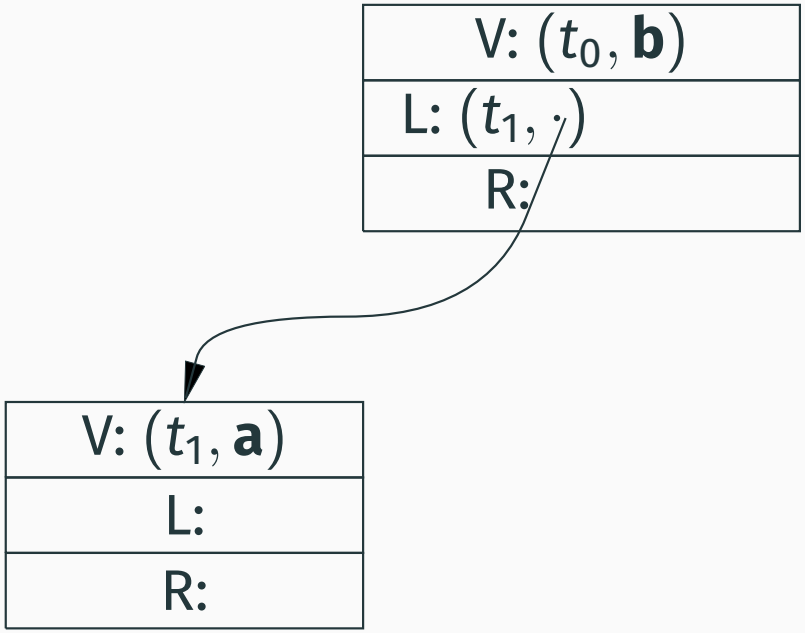
V: $(t_0, \mathbf{b})$
L:
R:

Example of a BST with fat nodes

Ephemeral BST:

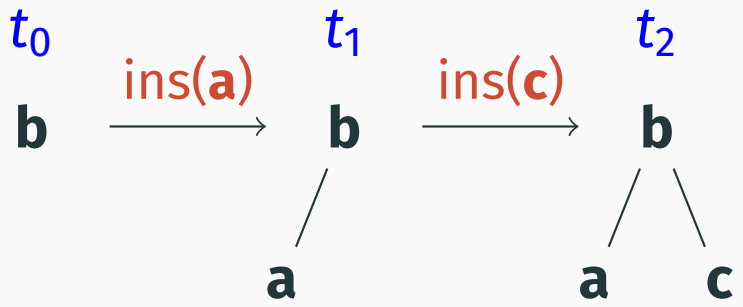


Persistent BST:

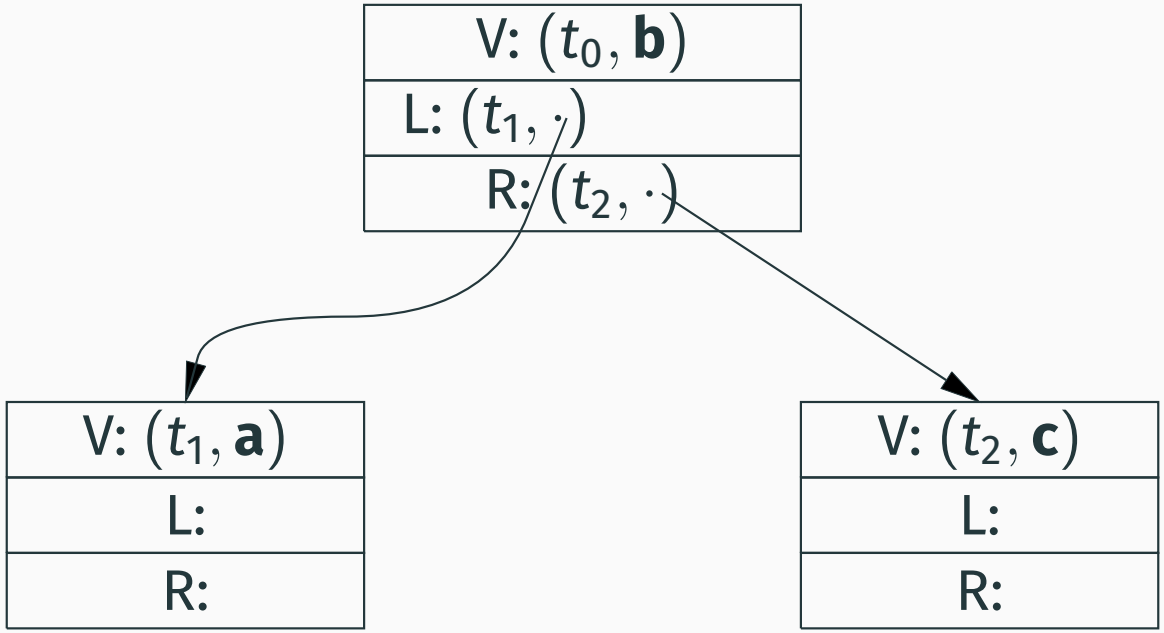


Example of a BST with fat nodes

Ephemeral BST:

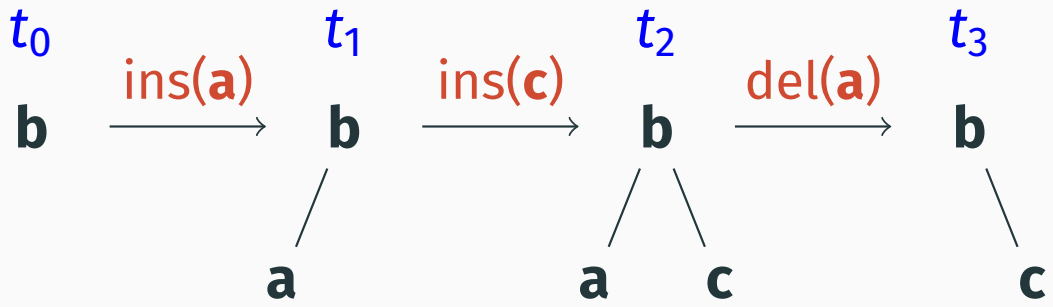


Persistent BST:

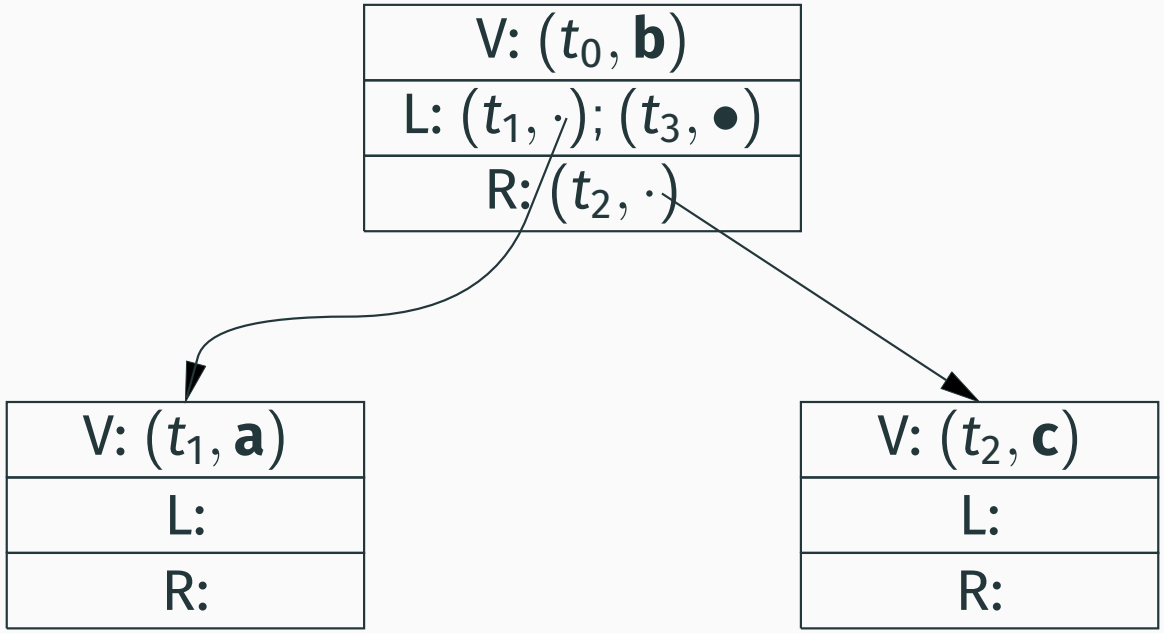


Example of a BST with fat nodes

Ephemeral BST:



Persistent BST:



Memory usage

Each update of a field consumes $O(1)$ space (to add an entry to the corresponding journal).

After c node creations and m field updates, the persistent data structure has size $\mathcal{O}(c + m)$.

For in-place insertion in a balanced AST, we have:

- one node creation and one field update in its parent;
- plus a small number of updates for rebalancing: $\mathcal{O}(1)$ amortized. ($\rightarrow$ 3rd lecture)

Same analysis for deletion.

After n insertions and n deletions, we therefore have a data structure of size $\mathcal{O}(n)$ instead of $\mathcal{O}(n \log n)$ in the previous approach.

Access time

In a simple implementation of fat nodes, accessing the value of a field at date t takes time $\mathcal{O}(\log w)$ where w is the number of modifications of the field.

This translates to $\mathcal{O}(\log^2 n)$ lookup time for each point in the planar point location problem.

Sarnak and Tarjan show how to keep $\mathcal{O}(\log n)$ lookup time using a combination of fat node updates and of node copying: it suffices to bound the size of journals, and to create a new fat node when the journal overflows. ($\rightarrow$ 4th lecture)

The result is an optimal data structure for the planar point location problem: space $\mathcal{O}(n)$ and lookup time $\mathcal{O}(\log n)$.

The emergence of purely functional programming

Lambda-calculus and general recursive functions

The pure lambda-calculus: (Church, 1935)

$$M, N ::= x \mid \lambda x.\, M \mid M\ N$$

Everything is encoded as functions: data structures (integers, Booleans, lists, …) and control structures (conditional, fixed-point operators for recursion).

General recursive functions: (Kleene, 1936)

One data type (tuples of integers) + operators that build functions $\mathbb{N}^p \to \mathbb{N}^q$ (succ, pred, projections, composition, primitive recursion, minimization).

Initially studied as computability formalisms (they are equivalent to Turing machines), not as programming languages.

Lisp (J. McCarthy et al, 1960)

Initially: a FORTRAN library for symbolic computation, manipulating S-expressions:

$$\begin{aligned} \mathit{sexp} &::= \mathit{atom} \mid (\mathit{sexp}\ .\ \mathit{sexp}) \\ \mathit{atom} &::= \mathit{number} \mid \mathit{symbol} \mid \texttt{nil} \end{aligned}$$

Quickly evolved into an applicative programming language to define recursive functions over S-expressions, these functions being represented as S-expressions themselves.

Lisp: the first applicative language

```
(define mapcar (fun lst)
  (if (null lst)
      nil
      (cons (fun (car lst)) (mapcar fun (cdr lst)))))
```

No expression/statement distinction; everything is an expression.

Execution (of an expression) = evaluation (computing its value).

Recursion s preferred over iteration.

No (or few) in-place updates: `cons` always returns a fresh list cell; memory is reclaimed by automatic garbage collection.

The beginning of a rich lineage of languages: Common Lisp, Scheme, Racket, Clojure, ...

In classic Lisp, which programming style, which algorithms?

Quite naturally: purely functional programming using lists as the main data structure.

- Code readability, code reusability.
- Suboptimal complexity (lists $\rightarrow \mathcal{O}(n)$).
- But n is often small, especially in the 1960's!
- Symbolic computation algorithms are expensive anyway.

As an attempt to increase performance: a few imperative features.

- `setq` to change the value of a symbol (assignment);
- `rplaca`, `rplacd` to update a list cell in place (mutation);
- arrays (with in-place updates).

ML and its offsprings (Milner et al, 1978–)

Initially: Lisp + static typing and type abstraction

```
absrectype * tree = * + * tree # * tree
  with leaf n = abstree(inl n)
   and node (t1, t2) = abstree(inr(t1, t2))
   and isleaf t = isl(reptree t)
   and leafval t = outl(reptree t) ? failwith ‘leafval‘
   and leftchild t = fst(outr(reptree t) ? failwith ‘leftchild‘
   and rightchild t = snd(outr(reptree t) ? failwith ‘leftchild‘
```

Later extended with algebraic types and pattern matching (HOPE, Prolog).

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
let rec sum = function Leaf n -> n | Node(l, r) -> sum l + sum r
```

Many offsprings: Standard ML, Caml, OCaml, F#, etc.

Imperative features in ML

Same trade-off as in Lisp: pure functional programming by default + imperative features when needed.

> *At the same time, imperative features were important for practical reasons; no-one had experience of large useful programs written in a pure functional style. In particular, an exception-raising mechanism was highly desirable for the natural presentation of tactics.*
>
> *(Milner et al, The Definition of Standard ML)*

Not just exceptions, but also mutable state presented as references (indirection cells with in-place updates) and arrays.

```
let x = ref 0 in x := !x + 2
```

Haskell (Hudak, Peyton Jones, Wadler, et al, 1992-)

Haskell $\approx$ ML + lazy evaluation + type classes + much more.

A convergence of 1980's research work on lazy evaluation (on-demand evaluation) of expressions, as opposed to the strict evaluation used in Lisp and ML.

- Supports defining more code fragments as function. E.g. the `ifthenelse` function:

```
ifthenelse True  a b = a
ifthenelse False a b = b
```

- Facilitates the definition and handling of infinite data structures such as streams (= infinite lists).

Haskell: the triumph of purely functional programming?

> *Laziness prevented us from sinning.*
>
> *(attributed to S. Peyton Jones)*

Lazy evaluation makes it nearly impossible to guess when an expression is evaluated. This makes side effects unusable, including I/O and assignments.

Hence, a return to the roots of Lisp 1960:

- Purely functional programming.
- Equational reasoning.
- Program derivation via calculation.

Haskell: the return of imperative programming

> *In short, Haskell is the world's finest imperative programming language*
>
> *(S. Peyton Jones, 2000)*

Very pressing needs: input/output, interfacing with other languages, mutable arrays (for numerical codes), references (for fast unification algorithms), …

$\rightarrow$ Imperative features come back in controlled ways that remain compatible with lazy, non-strict evaluation:

- Haskell: monads
- Clean, Linear Haskell: linear types.

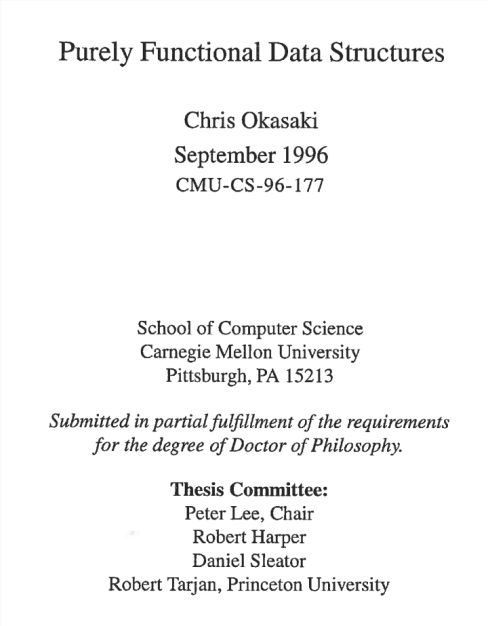
The emergence of purely functional algorithms

In the 1990's, a realization came (at long last): a purely functional program can be algorithmically efficient provided it uses appropriate purely functional or persistent data structures.

Early users: proof assistants and other verification tools:

- for stronger correctness guarantees (no interferences);
- to reduce memory requirements (data sharing).

Chris Okasaki's PhD work



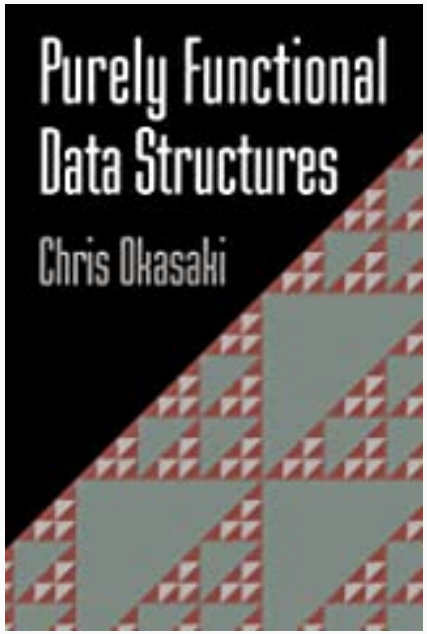
Purely Functional Data Structures

Chris Okasaki
September 1996
CMU-CS-96-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Peter Lee, Chair
Robert Harper
Daniel Sleator
Robert Tarjan, Princeton University



Systematic reformulation of advanced data structures and algorithms in purely functional style.

Novel uses of lazy evaluation, with matching novel techniques for amortized analysis.

Journal of Functional Programming, 1993-2008

FUNCTIONAL PEARLS

Efficient sets—a balancing act

STEPHEN ADAMS
Electronics and Computer Science Department, University of Southampton, UK

FUNCTIONAL PEARL

The Zipper

GÉRARD HUET
INRIA Rocquencourt, France

FUNCTIONAL PEARLS

Diets for fat sets

MARTIN ERWIG
FernUniversität Hagen, Praktische Informatik IV. 58084 Hagen, Germany
(*e-mail:* erwig@fernuni-hagen.de)

FUNCTIONAL PEARL

Red-black trees in a functional setting

CHRIS OKASAKI*
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213, USA

FUNCTIONAL PEARL

A fresh look at binary search trees

RALF HINZE
Institute of Information and Computing Sciences, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* ralf@cs.uu.nl)

Simple and efficient purely functional queues and deques

CHRIS OKASAKI
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
(*e-mail:* cokasaki@cs.cmu.edu)

FUNCTIONAL PEARL

Three algorithms on Braun trees

CHRIS OKASAKI*
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
(*e-mail:* cokasaki@cs.cmu.edu)

FUNCTIONAL PEARL

Explaining binomial heaps

RALF HINZE
Institut für Informatik III, Universität Bonn Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

Red-black trees with types

STEFAN KAHRS
University of Kent at Canterbury, Canterbury, Kent, UK

Finger trees: a simple general-purpose data structure

RALF HINZE
Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

ROSS PATERSON
Department of Computing, City University, London EC1V OHB, UK
(*e-mail:* ross@soi.city.ac.uk)

Course outline

Lectures

1. Introduction.
2. Balanced trees + path copying = persistent dictionaries.
3. Laziness matters! Reconciling amortization and persistence.
4. How to make an ephemeral data structure persistent?
5. Numerical representations and non-regular types.
6. From formal derivation to navigation in a structure: contexts, zippers, fingers.
7. In search of the lost array: theoretical limits and conclusions.

Seminar talks

1. Tobias Nipkow (T.U. München)
   Verification of functional data structures:
   Correctness and complexity.
2. Jean-Christophe Filliâtre (CNRS)
   Structures de données semi-persistantes.
3. KC Sivaramakrishnan (IIT Madras et Tarides)
   Mergeable replicated data types.
4. Arthur Charguéraud (Inria)
   Transience : comment allier persistance et performance.
5. Pierre-Etienne Meunier (Coturnix)
   Une algèbre de modifications, ou: le contrôle de versions pour tous.

References

References

Two reference texts, useful for the whole course:

- Chris Okasaki, *Purely Functional Data Structures*,
  Cambridge University Press, 1998, 2009.

- Haim Kaplan, *Persistent Data Structures*,
  chap. 31 of *Handbook of data structures and applications*,
  Chapman&Hall / CRC Press, 2005,
  http://www.cs.tau.ac.il/~haimk/papers/persistent-survey.ps