Persistent data structures, second lecture

Balanced trees + path copying = persistent dictionaries

Xavier Leroy

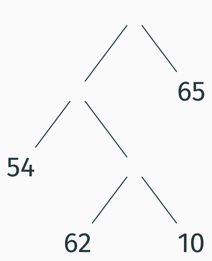
2023-03-16

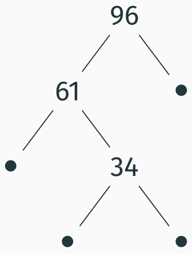
Collège de France, chair of Software sciences
xavier.leroy@college-de-france.fr

Binary search trees

Binary trees

A tree = a leaf OR a node carrying two sub-trees.





Carries information at the leaves or at the nodes.

Information = element $\Longrightarrow$ finite sets

Information = (key, value) pair $\Longrightarrow$ finite maps, dictionaries

The algebraic view

With elements $x \in X$ at leaves:

$$\begin{aligned} A ::=\ & [x] && \text{leaf} \\ \mid\ & \langle A_1, A_2 \rangle && \text{node} \end{aligned}$$

With elements $x \in X$ at nodes:

$$\begin{aligned} A ::=\ & \bullet && \text{leaf} \\ \mid\ & \langle A_1, x, A_2 \rangle && \text{node} \end{aligned}$$

Direct translation to algebraic types
(OCaml, Haskell, …) or inductive types (Coq, Agda, …):

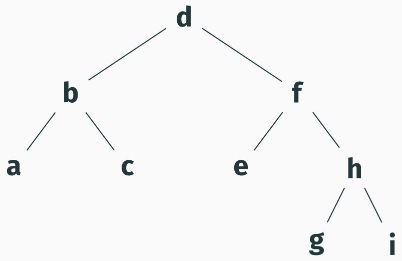
```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Binary search trees (BST)

A binary search tree =

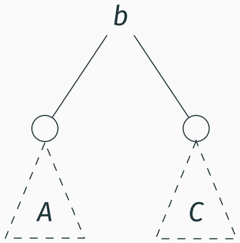
a binary tree with elements at nodes

elements strictly increase from left to right (infix traversal).



(Note: we don't draw branches leading to leaves.
What looks like the leaf **a** is the trivial node $\langle \bullet, \mathbf{a}, \bullet \rangle$)

Binary search trees



Inductive invariant: for every node $\langle A, b, C\rangle$,
the elements contained in the left sub-tree A are $< b$
the elements contained in the right sub-tree C are $> b$

Binary search in a BST

Membership in a finite set:

$$\mathtt{mem}(x, \bullet) = \mathtt{false}$$

$$\mathtt{mem}(x, \langle A, b, C\rangle) = \begin{cases} \mathtt{mem}(x, A) & \text{if } x < b \\ \mathtt{true} & \text{if } x = b \\ \mathtt{mem}(x, C) & \text{if } x > b \end{cases}$$

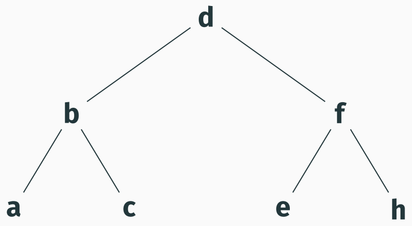
Looking up a key in a dictionary:

$$\mathtt{find}(x, \bullet) = \mathtt{None}$$

$$\mathtt{find}(x, \langle A, (k, v), C\rangle) = \begin{cases} \mathtt{find}(x, A) & \text{if } x < k \\ \mathtt{Some}(v) & \text{if } x = k \\ \mathtt{find}(x, C) & \text{if } x > k \end{cases}$$

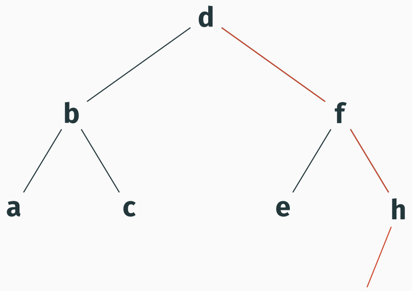
Time: $\mathcal{O}(h)$ where h is the height of the tree, i.e. the maximal length of a branch.

Insertion in a BST: imperative version



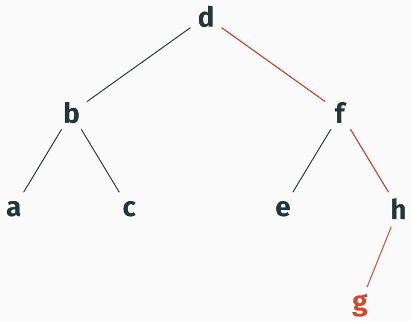
1. Search for the element to be inserted (here, **g**).

Insertion in a BST: imperative version



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

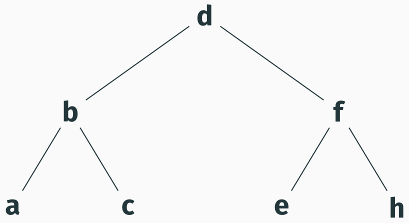
Insertion in a BST: imperative version



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

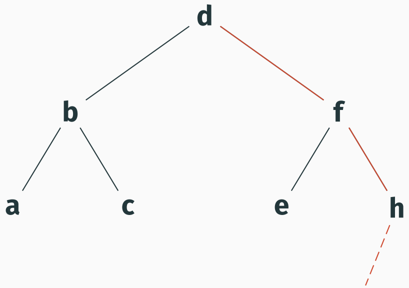
Time: $\mathcal{O}(h)$, space: $\mathcal{O}(1)$.

Insertion in a BST: persistent version



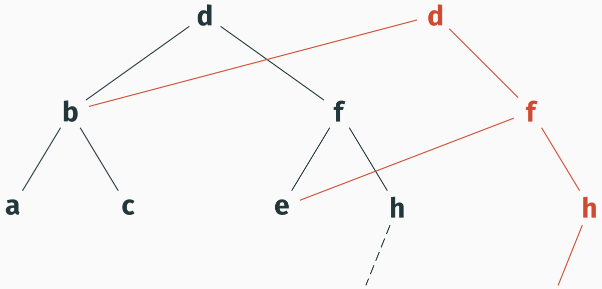
1. Search for the element to be inserted (here, **g**).

Insertion in a BST: persistent version



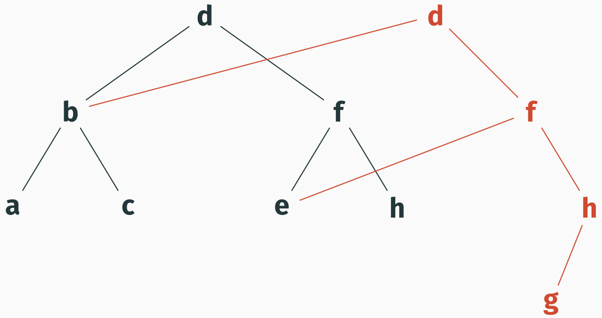
1. Search for the element to be inserted (here, **g**).

Insertion in a BST: persistent version



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



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 $\langle \bullet, \mathbf{g}, \bullet \rangle$.

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

Insertion in a BST: algebraic presentation

Adding an element to a finite set:

$$\mathtt{add}(x, \bullet) = \langle \bullet, x, \bullet \rangle$$

$$\mathtt{add}(x, \langle A, b, C\rangle) = \begin{cases} \langle \mathtt{add}(x, A), b, C\rangle & \text{if } x < b \\ \langle A, x, C\rangle & \text{if } x = b \\ \langle A, b, \mathtt{add}(x, C)\rangle & \text{if } x > b \end{cases}$$

Inserting a (key, value) pair in a dictionary:

$$\mathtt{ins}(k, v, \bullet) = \langle \bullet, (k, v), \bullet \rangle$$

$$\mathtt{ins}(k, v, \langle A, (k', v'), C\rangle) = \begin{cases} \langle \mathtt{ins}(k, v, A), (k', v'), C\rangle & \text{if } k < k' \\ \langle A, (k, v), C\rangle & \text{if } k = k' \\ \langle A, (k', v'), \mathtt{ins}(x, C)\rangle & \text{if } k > k' \end{cases}$$

Insertion in a BST: pure functional implementation

```
let rec add x t =
  match t with
  | Leaf -> Node(Leaf, x, Leaf)
  | Node(a, b, c) ->
      if x < b then
        Node(add x a, b, c)
      else if x > b then
        Node(a, b, add x c)
      else
        t
```

The path copying takes place when returning from recursive calls, by evaluating the expressions `Node(add x a, b, c)` or `Node(a, b, add x c)`.

Insertion in a BST: algebraic specification, verification

A simple equational specification:

$$\mathtt{mem}(x, \mathtt{add}(x, T)) = \mathtt{true} \tag{1}$$

$$\mathtt{mem}(x, \mathtt{add}(y, T)) = \mathtt{mem}(x, T) \qquad \text{if } x \neq y \tag{2}$$

A proof by structural induction over T.

Base case for (1): $\mathtt{mem}(x, \mathtt{add}(x, \bullet)) = \mathtt{true}$.

Inductive case for (1): unroll definitions and analyze 3 cases

$$\mathtt{mem}(x, \mathtt{add}(x, \langle A, b, C\rangle)) = \begin{cases} \mathtt{mem}(x, \mathtt{add}(x, A)) & \text{if } x < b \\ \mathtt{true} & \text{si } x = b \\ \mathtt{mem}(x, \mathtt{add}(x, C)) & \text{if } x > b \end{cases}$$

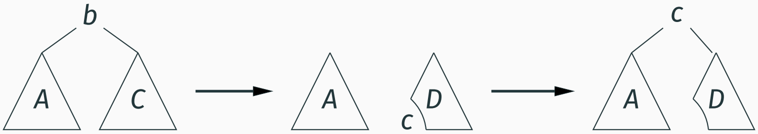
The result (1) follows from the induction hypothesis.

Exercise: prove (2).

Deletion in a BST

We search for the sub-tree $\langle A, b, C \rangle$ carrying the element b to be removed, then replace it by a tree that contains the same elements as A and C.

Idea: use the smallest element of C (or the largest element of A) as the root of this new tree.



Deletion in a BST

$$\mathtt{del}(x, \bullet) = \bullet$$

$$\mathtt{del}(x, \langle A, b, C\rangle) = \begin{cases} \langle \mathtt{del}(x, A), b, C\rangle & \text{if } x < b \\ \mathtt{join}(A, C) & \text{if } x = b \\ \langle A, b, \mathtt{del}(x, C)\rangle & \text{if } x > b \end{cases}$$

$$\mathtt{join}(A, \bullet) = \mathtt{join}(\bullet, A) = A$$

$$\mathtt{join}(A, C) = \langle A, \mathtt{min}(C), \mathtt{delmin}(C)\rangle \qquad \text{if } C \neq \bullet$$

$$\mathtt{min}(\langle \bullet, b, C\rangle) = b \qquad \mathtt{delmin}(\langle \bullet, b, C\rangle) = C$$

$$\mathtt{min}(\langle A, b, C\rangle) = \mathtt{min}(A) \quad \mathtt{delmin}(\langle A, b, C\rangle) = \langle \mathtt{delmin}(A), b, C\rangle$$

Other operations

Smallest element, largest element
$\Rightarrow$ usable as a priority queue.

If every sub-tree is annotated by its size:
which element has rank k? which rank has element x?
$\Rightarrow$ usable as an ordered sequence.

Set operations: union, intersection, difference, …

Set operations

Pattern: recurse on one of the two trees and split the other one.

$$\begin{aligned}
\texttt{union}(\bullet, T) &= \texttt{union}(T, \bullet) = T \\
\texttt{union}(\langle A, b, C\rangle, T) &= \langle \texttt{union}(A, A'), b, \texttt{union}(C, C')\rangle \\
&\qquad \text{where } (A', C') = \texttt{split}(T, b)
\end{aligned}$$

The function $\texttt{split}(T, b)$ returns two trees A' and C':
A' contains all elements of T that are $< b$
C' contains all elements of T that are $> b$.

Exercise: define `split`, set intersection, set difference.

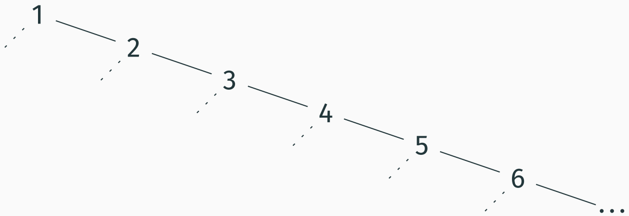
AVL trees

Balanced trees

BST operations are fast as long as the tree is balanced: its height h is small compared to the number of nodes n.

We aim for $h = \mathcal{O}(\log n)$.

However, some BSTs are completely unbalanced:

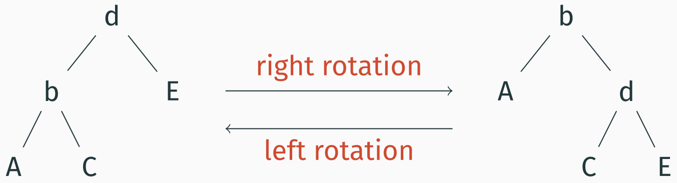


Such a tree appears naturally when inserting elements $1, 2, \ldots, n$ successively in this order. BST operations, then, take time $\mathcal{O}(n)$, as in the case of a sorted list.

Self-balancing trees

The operations (insertion, deletion, etc) should ensure that the tree remains balanced (height proportional to $\log n$) after any sequence of operations.

Idea: at any time, we can perform rotations over sub-trees of a BST.



Rotations preserve the BST property (elements increase from left to right), but can reduce imbalance.

AVL trees

Named after their authors, Georgii **A**delson-**V**elskii and Evgueni **L**andis.

G. Adelson-Velskii, E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR* 146 (1962), 263-266; English translation in *Soviet Mathematics Doklady* 3 (1962), 1259-1262.

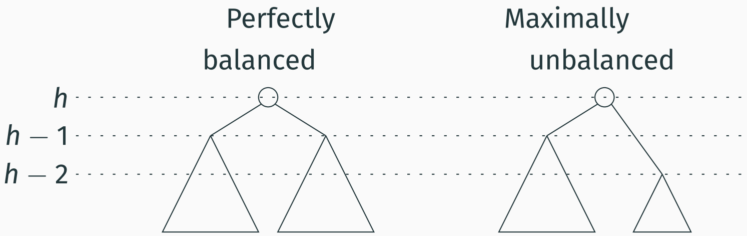
The AVL criterion: a balancing criterion based on the heights of sub-trees.

> *For each node $\langle A, b, C \rangle$, the heights of the sub-trees A, C differ by at most 1:*

$$|h(A) - h(C)| \leq 1$$

AVL trees

For each node $\langle A, b, C\rangle$, the heights of the sub-trees A, C differ by at most 1.

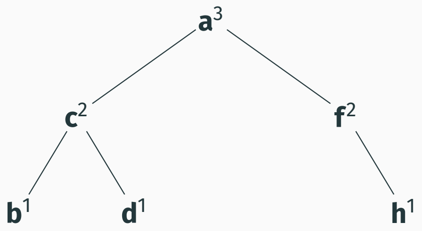


$N(h) = 1 + 2 \times N(h-1)$ $\quad$ $N(h) = 1 + N(h-1) + N(h-2)$

$N(h) = 2^h - 1$ $\quad$ $N(h) = 2F_h - 1$ (Fibonacci)

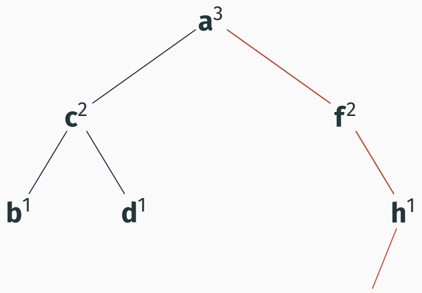
The height h is logarithmic in the size $n = N(h)$ of the tree: $h < \frac{3}{2}\log_2(n+1)$.

Insertion in an AVL tree: imperative version



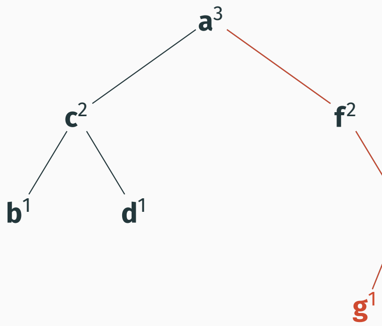
1. Search for the element to be inserted (here, **g**).

Insertion in an AVL tree: imperative version



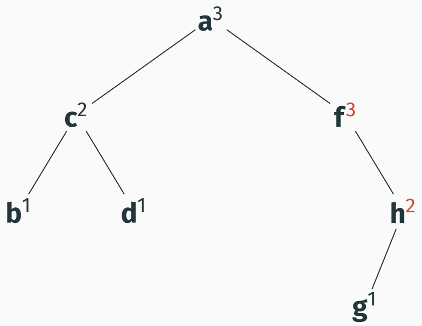
1. Search for the element to be inserted (here, **g**).

Insertion in an AVL tree: imperative version



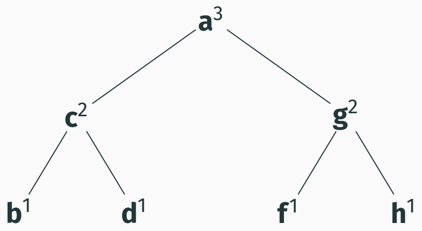
1. Search for the element to be inserted (here, **g**).
2. Replace the leaf by the node $\langle \bullet, \mathbf{g}, \bullet \rangle$.

Insertion in an AVL tree: imperative version



1. Search for the element to be inserted (here, $\mathbf{g}$).
2. Replace the leaf by the node $\langle \bullet, \mathbf{g}, \bullet \rangle$.
3. Go back up the path, updating the heights of nodes ...

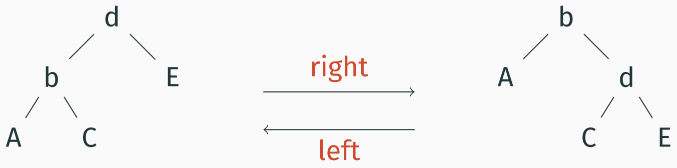
Insertion in an AVL tree: imperative version



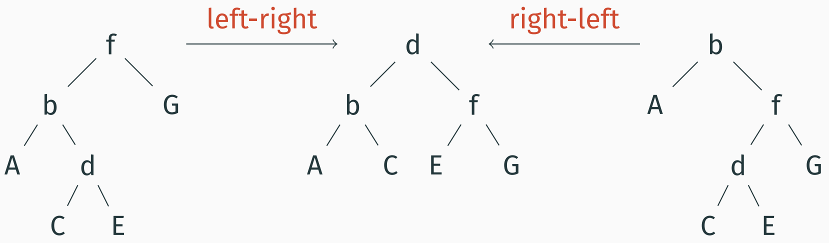
1. Search for the element to be inserted (here, **g**).
2. Replace the leaf by the node $\langle \bullet, \mathbf{g}, \bullet \rangle$.
3. Go back up the path, updating the heights of nodes …
4. … and performing rotations to restore the AVL criterion.

Rotations

Simple rotations: right (if *A* is too tall), left (if *E* is too tall).



Double rotations: if $\langle C, d, E\rangle$ is too tall.



Algebraic presentation

A smart constructor $\texttt{bal}(A, b, C)$ that builds a BST equivalent to $\langle A, b, C\rangle$, while performing the required rotations so that it is an AVL, assuming $|h(A) - h(C)| \leq 2$ initially.

$$\begin{aligned}
\texttt{bal}(A, b, C) &= \langle A, b, C\rangle \quad \text{if } |h(A) - h(C)| \leq 1 \\
\texttt{bal}(\langle A, b, C\rangle, d, E) &= \langle A, b, \langle C, d, E\rangle\rangle \\
&\qquad \text{if } h(A) > h(C), h(A) > h(E) \\
\texttt{bal}(\langle A, b, \langle C, d, E\rangle\rangle, f, G) &= \langle\langle A, b, C\rangle, d, \langle E, f, G\rangle\rangle \\
&\qquad \text{if } \max(h(C), h(E)) > h(A) > h(G)
\end{aligned}$$

(Plus: 3 symmetric cases.)

We can also define $\texttt{bal}^*(A, b, C)$ with no precondition on $h(A), h(C)$ by iterating `bal` until the resulting tree is an AVL.

Insertion in an AVL tree: algebraic presentation

$$\mathtt{add}(x, \bullet) = \langle \bullet, x, \bullet \rangle$$

$$\mathtt{add}(x, \langle A, b, C \rangle) = \begin{cases} \mathtt{bal}(\mathtt{add}(x, A), b, C) & \text{if } x < b \\ \langle A, x, C \rangle & \text{if } x = b \\ \mathtt{bal}(A, b, \mathtt{add}(x, C)) & \text{if } x > b \end{cases}$$

Translates straightforwardly to a pure functional implementation, in time and space $\mathcal{O}(\log n)$ where n is the size of the tree.

Similar presentation and implementation for deletion in an AVL.

Other balancing criterion

Other criterion for height balancing:

A relaxed AVL criterion:

$$|h(A) - h(C)| \leq K \qquad \text{for each sub-tree } \langle A, b, C \rangle$$

Example: $K = 2$ for the `Set` and `Map` OCaml libraries
$\rightarrow$ fewer rotations than with AVL trees, but longer branches.

Red-black trees (later in this lecture).

Other balancing criterion

Weight balancing:

$$\frac{1}{K} \leq \frac{w(A)}{w(C)} \leq K \qquad \text{for each sub-tree } \langle A, b, C \rangle$$

$w(A) = 1 + |A|$ is the weight of A (1 + number of elements).

Example: $K = 4$ for the `Data.Map` Haskell library.

Delicate rotation criteria; several implementations are wrong.
(Hirai et Yamamoto, *Balancing weight-balanced trees*, JFP 21(3), 2011.)

Note: AVL trees are not weight-balanced, since we can have $w(A) \sim 2^h$ (perfectly balanced) and $w(C) \sim 2F_h \sim c.\varphi^h$ (maximally unbalanced), hence $w(A)/w(C) \to \infty$ when $h \to \infty$.

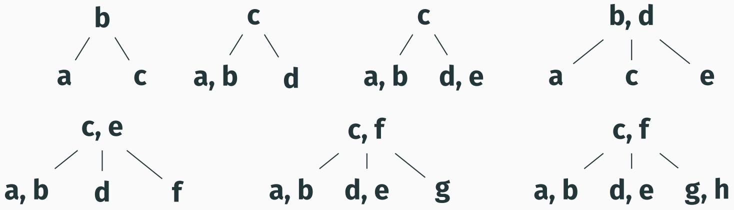
2-3 trees

2-3 trees

A different approach to balancing:
perfect trees + variable-arity nodes.

- All leaves are at the same level.
- Nodes carry either 2 sub-trees and 1 element
  or 3 sub-trees and 2 elements.

Examples of 2-3 trees with 3, 4, …, 8 elements:



Generalization: B-trees

A generalization of 2-3 trees to trees with a high branching degree:

- All leaves are at the same level.
- Intermediate nodes carry between $k/2$ and k sub-trees.
- The topmost node carries between 2 and k sub-trees.

k is chosen large enough so that one node = one disk block (typically).

2-3 trees: algebraic presentation, lookup algorithm

$$\begin{array}{rll} A, C, E ::= & \bullet & \text{leaf} \\ \mid & \langle A, b, C \rangle & \text{2-node} \\ \mid & \langle A, b, C, d, E \rangle & \text{3-node} \end{array}$$

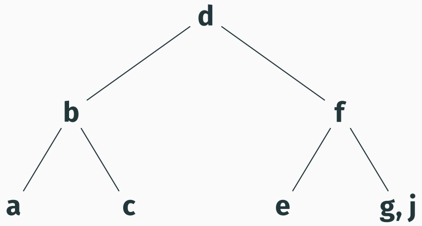
Lookup in a 2-3 tree: by binary and ternary search

$$\mathtt{mem}(x, \bullet) = \mathtt{false}$$

$$\mathtt{mem}(x, \langle A, b, C \rangle) = \begin{cases} \mathtt{mem}(x, A) & \text{if } x < b \\ \mathtt{true} & \text{if } x = b \\ \mathtt{mem}(x, C) & \text{if } x > b \end{cases}$$

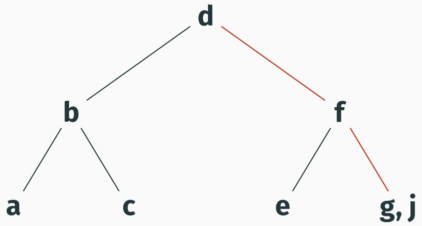
$$\mathtt{mem}(x, \langle A, b, C, d, E \rangle) = \begin{cases} \mathtt{true} & \text{if } x = b \text{ or } x = d \\ \mathtt{mem}(x, A) & \text{if } x < b \\ \mathtt{mem}(x, C) & \text{if } x > b \text{ and } x < d \\ \mathtt{mem}(x, E) & \text{if } x > d \end{cases}$$

Insertion in a 2-3 tree: imperative version



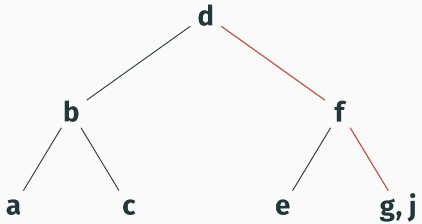
1. Search for the element to be inserted (here, **h**).

Insertion in a 2-3 tree: imperative version



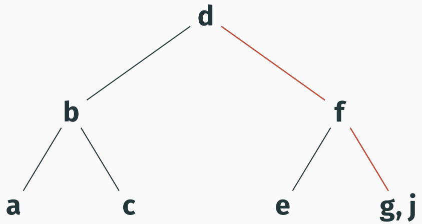
1. Search for the element to be inserted (here, **h**).

Insertion in a 2-3 tree: imperative version



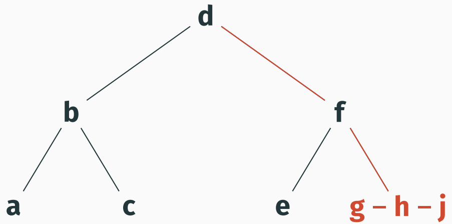
1. Search for the element to be inserted (here, **h**).
2. If we end on a 2-node, turn it into a 3-node:
   $\langle \bullet, \mathbf{g}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.

Insertion in a 2-3 tree: imperative version



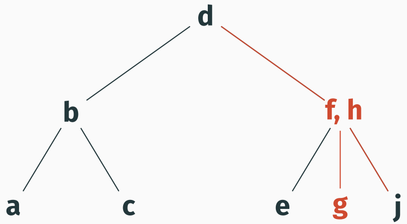
1. Search for the element to be inserted (here, **h**).
2. If we end on a 2-node, turn it into a 3-node:
   $\langle \bullet, \mathbf{g}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. If we end on a 3-node $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, “explode” it into two related 2-nodes $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$
   and insert all this in the 2-node above.

Insertion in a 2-3 tree: imperative version



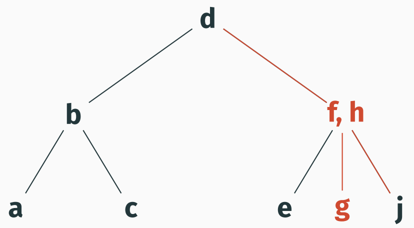
1. Search for the element to be inserted (here, **h**).
2. If we end on a 2-node, turn it into a 3-node:
   $\langle \bullet, \mathbf{g}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. If we end on a 3-node $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, "explode" it into two related 2-nodes $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$
   and insert all this in the 2-node above.

Insertion in a 2-3 tree: imperative version



1. Search for the element to be inserted (here, **h**).
2. If we end on a 2-node, turn it into a 3-node:
   $\langle \bullet, \mathbf{g}, \bullet \rangle \to \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \to \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. If we end on a 3-node $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, “explode” it into two related 2-nodes $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$
   and insert all this in the 2-node above.

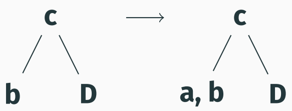
Insertion in a 2-3 tree: imperative version



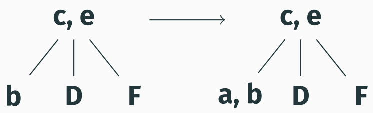
1. Search for the element to be inserted (here, **h**).
2. If we end on a 2-node, turn it into a 3-node:
   $\langle \bullet, \mathbf{g}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. If we end on a 3-node $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, "explode" it into two related 2-nodes $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$
   and insert all this in the 2-node above.
4. If the node above is a 3-node, explode it too and iterate.

All cases of insertion in leftmost position in a 2-3 tree of height 2

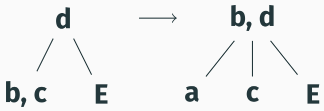
2-2 path becomes 2-3:



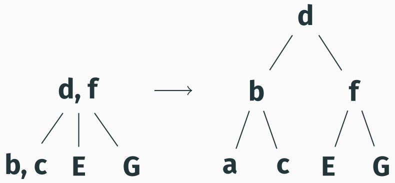
3-2 path becomes 3-3:



2-3 path becomes 3-2:



3-3 path becomes 2-2-2:



Note: similar to carry propagation when incrementing a base-2 number.

A purely functional, finely-typed implementation

Using generalized algebraic data types (GADTs) from Haskell and OCaml to enforce the height invariant.

```
type zero = Zero
type 'a succ = Succ of 'a

type _ tree =
  | Leaf : zero tree
  | Two : 'h tree * elt * 'h tree -> 'h succ tree
  | Three : 'h tree * elt * 'h tree * elt * 'h tree
                                              -> 'h succ tree
```

The `'h` parameter in type `'h tree` is the height of the tree, encoded in types as a Peano integer:
`zero` (= 0), `zero succ` (= 1), `zero succ succ` (= 2), etc.

A purely functional, finely-typed implementation

Using generalized algebraic data types (GADTs) from Haskell and OCaml to enforce the height invariant.

```
type zero = Zero
type 'a succ = Succ of 'a

type _ tree =
  | Leaf : zero tree
  | Two : 'h tree * elt * 'h tree -> 'h succ tree
  | Three : 'h tree * elt * 'h tree * elt * 'h tree
                                    -> 'h succ tree
```

The types of constructors `Leaf`, `Two`, `Three` enforce the height invariant.

A purely functional, finely-typed implementation

Using generalized algebraic data types (GADTs) from Haskell and OCaml to enforce the height invariant.

```
type zero = Zero
type 'a succ = Succ of 'a

type _ tree =
  | Leaf : zero tree
  | Two : 'h tree * elt * 'h tree -> 'h succ tree
  | Three : 'h tree * elt * 'h tree * elt * 'h tree
                                      -> 'h succ tree

type set = Set : 'h tree -> set
```

A finite set (type `set`) is an `'h tree` for some `'h` (existential quantification).

Type-checking insertion

The result of insertion is not just a `tree` but an `etree`: either `Ok` if the result is a well-formed 2-3 tree, or `Split` if it's the outcome of exploding a node.

```
type _ etree =
  | Ok: 'h tree -> 'h etree
  | Split: 'h tree * elt * 'h tree -> 'h etree
```

The difference between `Split` and `Two` is that `Two` is a tree of height $h + 1$, while `Split` is considered as having height h.

Implementing insertion

```
let rec add_t : type h. elt -> h tree -> h etree =
  fun x t ->
  match t with
  | Leaf -> Split(Leaf, x, Leaf)
  | Two(a, b, c) ->
      if x = b then Ok t else
      if x < b then two1(add_t x a, b, c)
      else two2(a, b, add_t x c)
  | Three(a, b, c, d, e) ->
      if x = b || x = d then Ok t else
      if x < b then three1(add_t x a, b, c, d, e)
      else if x < d then three2(a, b, add_t x c, d, e)
      else three3(a, b, c, d, add_t x e)
```

Smart constructors to absorb explosions

```
  two1: 'h etree * elt * 'h tree -> 'h succ etree
  two2: 'h tree * elt * 'h etree -> 'h succ etree
three1: 'h etree * elt * 'h tree * elt * 'h tree -> 'h succ etree
three2: 'h tree * elt * 'h etree * elt * 'h tree -> 'h succ etree
three3: 'h tree * elt * 'h tree * elt * 'h etree -> 'h succ etree
```

Definitions guided by the types and the BST invariant.

For instance:

```
let two1 (ll, x, r) =
    match ll with
    | Ok l -> Ok (Two(l, x, r))
    | Split(a, b, c) -> Ok (Three(a, b, c, x, r))
let three1 (ll, x, m, y, r) ->
    match ll with
    | Ok l -> Ok (Three(l, x, m, y, r))
    | Split(a, b, c) -> Split(Two(a, b, c), x, Two(m, y, r))
```

Implementing insertion

At the top level, for sets instead of raw trees:

```
let add : elt -> set -> set =
  fun x (Set t) ->
    match add_t x t with
    | Ok t’ -> Set t’
    | Split(a, b, c) -> Set (Two(a, b, c))
```

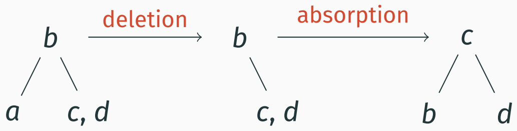
The second case increases height by one, but this is hidden by the `Set` constructor, which quantifies existentially over the height.

Deletion in a 2-3 tree

Just like insertion can "explode" a 3-node, deletion can "implode" a 2-node because its height needs to decrease by 1.

$$\langle \bullet, a, \bullet, b, \bullet \rangle \to \langle \bullet, b, \bullet \rangle \ ✔ \qquad \langle \bullet, a, \bullet \rangle \to \bullet \ ✘$$

We must absorb this height reduction further "up" the tree:



Deletion in a 2-3 tree

The result of deletion is not just a `tree` but an `etree`:

```
type _ etree =
  | Ok: 'h tree -> 'h etree
  | Short: 'h tree -> 'h succ etree
```

`Short t` is a short tree `t`, with height h instead of the expected $h+1$. This wrong height must be absorbed by smart constructors:

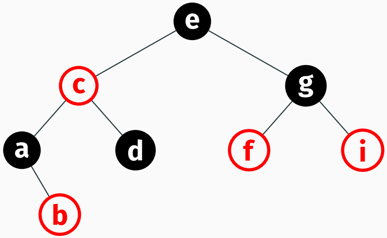
```
  two1: 'h etree * elt * 'h tree -> 'h succ etree
  two2: 'h tree * elt * 'h etree -> 'h succ etree
three1: 'h etree * elt * 'h tree * elt * 'h tree -> 'h succ etree
three2: 'h tree * elt * 'h etree * elt * 'h tree -> 'h succ etree
three3: 'h tree * elt * 'h tree * elt * 'h etree -> 'h succ etree
```

Deletion follows the same pattern as insertion. (Exercise.)

Red-black trees

Red-black trees

Binary search trees where each node has a color: red ou black.



Two invariants over colors:

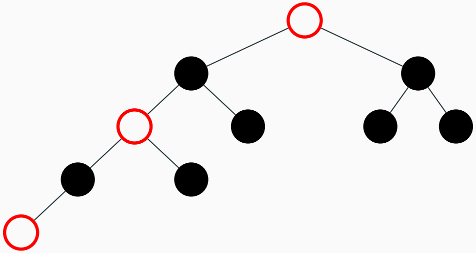
1. The children of a red node are not red.
2. All paths from the root to a leaf contain the same number of black nodes.
   This number is called the black height of the tree.

Red-black trees are balanced

1. The children of a red node are not red.
2. All paths from the root to a leaf contain the same number of black nodes.

Let h be the black height of the tree. Then, all paths from the root to a leaf have lengths h to $2h + 1$. This ensures that the tree contains at least $2^h - 1$ nodes, and therefore that it is balanced.

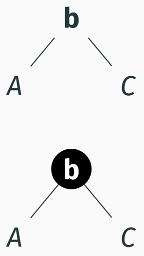
This can be proved by considering maximally-unbalanced trees:



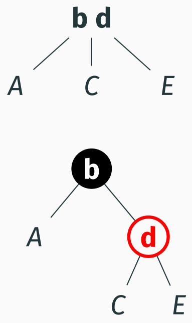
Red=black trees and 2-3-4 trees

Red-black trees were introduced by L. J. Guibas and R. Sedgewick (1978) as a simplified representation for 2-3-4 trees, that is, B-trees of degree 4, where each node carries from 2 to 4 sub-trees and from 1 to 3 keys.

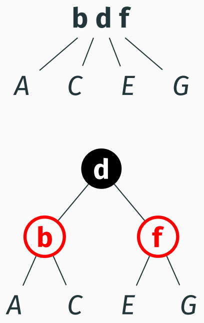
Node 2:



Node 3:

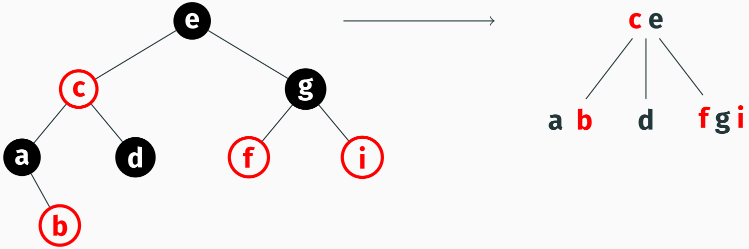


Node 4:



Red=black trees and 2-3-4 trees

Symmetrically, we recover a 2-3-4 tree from a red-black tree by lifting the red nodes to the same level as their black parents:



Algebraic presentation

Color:	$k ::= \mathbf{R} \mid \mathbf{B}$	red, black
Red-black tree:	$A, C ::= \bullet \mid k\langle A, b, C\rangle$	

Standard binary search, ignoring colors:

$$\mathtt{mem}(x, \bullet) = \mathtt{false}$$

$$\mathtt{mem}(x, k\langle A, b, C\rangle) = \begin{cases} \mathtt{mem}(x, A) & \text{if } x < b \\ \mathtt{true} & \text{if } x = b \\ \mathtt{mem}(x, C) & \text{if } x > b \end{cases}$$

Performs the same comparisons as binary/ternary/quaternary search in the corresponding 2-3-4 tree.

Insertion in a red-black tree

Same approach as for AVL trees:

- Search for the element x to be inserted.
- If we reach a leaf, we replace it with $\mathbf{R}\langle\bullet, x, \bullet\rangle$.
- Restore the color invariants on the way "up" (`bal` function).
- Recolor the root to black. (This fixes red-red violations at the top.)

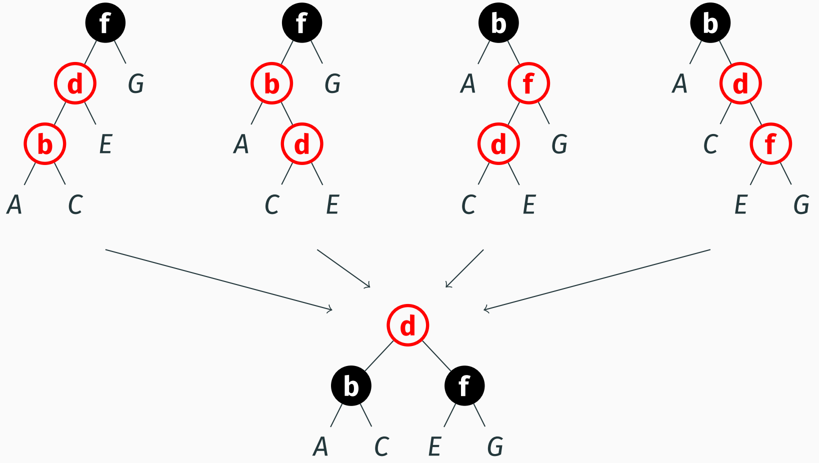
$$\texttt{add}(x, \bullet) = \mathbf{R}\langle\bullet, x, \bullet\rangle$$

$$\texttt{add}(x, k\langle A, b, C\rangle) = \begin{cases} \texttt{bal}(k, \texttt{add}(x, A), b, C) & \text{if } x < b \\ k\langle A, x, C\rangle & \text{if } x = b \\ \texttt{bal}(k, A, b, \texttt{add}(x, C)) & \text{if } x > b \end{cases}$$

Restoring color invariants by rotations and recoloring

In algorithmic textbooks: 8 cases or more.

C. Okasaki (1999): 4 cases are enough.



Rebalancing in algebraic notation

$$
\begin{aligned}
\texttt{bal}(\mathbf{B}, \mathbf{R}\langle \mathbf{R}\langle A, b, C\rangle, d, E\rangle, f, G) &= \mathbf{R}\langle \mathbf{B}\langle A, b, C\rangle, d, \mathbf{B}\langle E, f, G\rangle\rangle \\
\texttt{bal}(\mathbf{B}, \mathbf{R}\langle A, b, \mathbf{R}\langle C, d, E\rangle\rangle, f, G) &= \quad \text{"} \quad \text{"} \quad \text{"} \\
\texttt{bal}(\mathbf{B}, A, b, \mathbf{R}\langle \mathbf{R}\langle C, d, E\rangle, f, G\rangle) &= \quad \text{"} \quad \text{"} \quad \text{"} \\
\texttt{bal}(\mathbf{B}, A, b, \mathbf{R}\langle C, d, \mathbf{R}\langle E, f, G\rangle\rangle) &= \quad \text{"} \quad \text{"} \quad \text{"} \\
\texttt{bal}(c, A, b, C) &= c\langle A, b, C\rangle \qquad \text{in all other cases}
\end{aligned}
$$

Deletion in a red-black tree

Same algorithm as for AVL trees:

- Find the sub-tree $k\langle A, x, C\rangle$ carrying x, the element to be removed.
- Replace it with $k\langle A, \texttt{min}(C), \texttt{delmin}(C)\rangle$.
- Rebalance (restore invariants) on the way up.

However, rebalancing is much more complicated than for insertion (12 different cases or more).

Explaining deletion with “double black” nodes

(K. Germane et M. Might, *Deletion: the curse of the red-black tree*, JFP 24(4), 2014.)

Intermediate results of deletion can be not only red-black trees, but also

- a “double” leaf ●●, which counts as 1 black node;
- a “double black” node $\mathbf{BB}\langle A, b, C\rangle$ (drawn in white!), which counts as 2 black nodes.

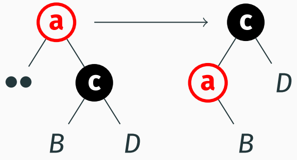
During rebalancing, these double nodes and leaves propagate upwards (while preserving red-black invariants) before being absorbed eventually.

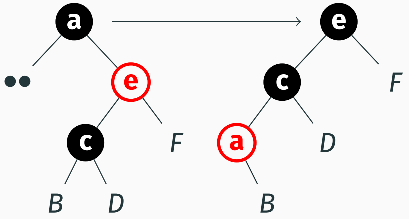
Deleting the smallest element

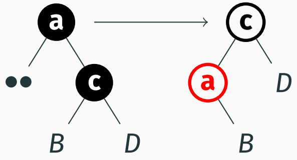
Base case: (preserving the black height)

$$\mathbf{R}\langle \bullet, x, \bullet\rangle \longrightarrow \bullet \qquad \mathbf{B}\langle \bullet, x, \bullet\rangle \longrightarrow \bullet\bullet$$

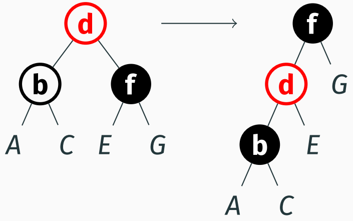
Propagating ●● upwards: (plus: left-right symmetric cases)

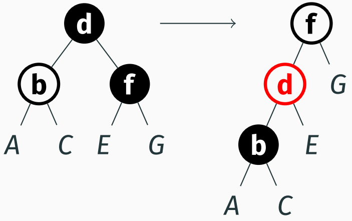


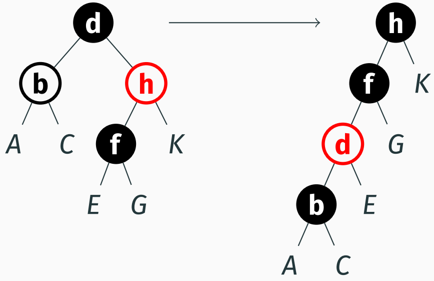




Propagating double black nodes upwards







Tries

Tries (also known as prefix trees)

(From *information reTRIEval*.)

A representation of finite sets or finite maps whose keys are words, that is, lists of symbols.

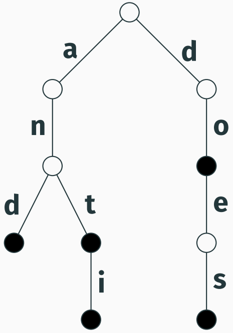
For instance:

- character strings
  (symbols = characters, or bytes, or bits)
- integer numbers
  (symbols = bits or groups of k bits).

Each node of a trie has between 0 and K sub-tries, each labeled by a symbol. (K = number of different symbols)

An example of a trie

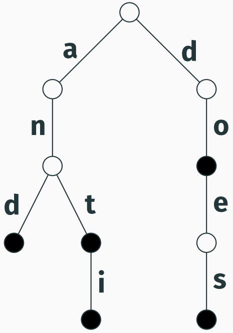
The set comprising the words **and**, **ant**, **anti**, **do**, **does**.



Black nodes (●) mark the end of a word.

An example of a trie

The set comprising the words **and**, **ant**, **anti**, **do**, **does**.

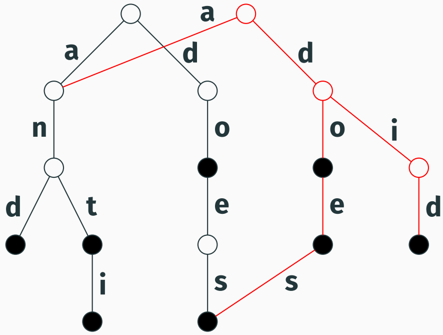


Searching: follow the edges that spell out the word.

The word belongs to the set iff we end on a black node.

An example of a trie

The set comprising the words **and, ant, anti, do, does**.



Persistent insertion: copy and complete the branch that spells out the word. Mark the final node black.
(Example: inserting **doe** and **did**.)

Purely functional implementation of tries

Sets and maps:

```
type set = Node of bool * (char * set) list
type 'a map = Node of 'a option * (char * 'a map) list
```

`char` is the type of symbols. Sub-trees are organized as association lists `(char * ...) list`.

Special case when symbols are bits:

```
type set = Empty | Node of set * bool * set
type 'a map = Empty | Node of 'a map * 'a option * 'a map
```

Always two sub-trees, one for bit `0`, the other for bit `1`.

Lookup and insertion (symbols are bits)

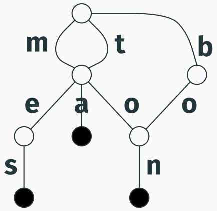
```
let rec mem s t =
  match t, s with
  | Empty, _ -> false
  | Node(_, acc, _), [] -> acc
  | Node(l, _ , r), b :: s’ -> mem s’ (if b then r else l)

let rec add s t =
  match t, s with
  | Empty -> add s (Node(Empty, false, Empty))
  | Node(l, acc, r), [] -> Node(l, true, r)
  | Node(l, acc, r), b :: s’ ->
      if b then Node(l, acc, add s’ r)
           else Node(add s’ l, acc, r)
```

Time proportional to $|s|$. If comparing two keys is expensive, tries are more efficient than BSTs (time $\mathcal{O}(|s| \log n)$).

Sharing sub-tries corresponding to common suffixes

In an immutable trie, sub-tries corresponding to suffixes common to several words can be shared, obtaining a directed acyclic word graph (DAWG), also known as deterministic acyclic finite-state automaton (DAFSA).

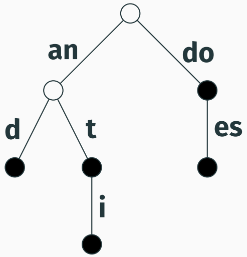
Example: **bon**, **mon**, **ma**, **mes**, **ton**, **ta**, **tes**.



Construction: using *hash consing* or automata minimization.

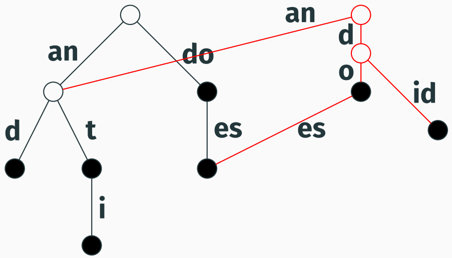
Compressed trie

To avoid trivial nodes, we can label each sub-trie not by a single symbol but by a sub-word (= a list of symbols).



Compressed trie

To avoid trivial nodes, we can label each sub-trie not by a single symbol but by a sub-word (= a list of symbols).

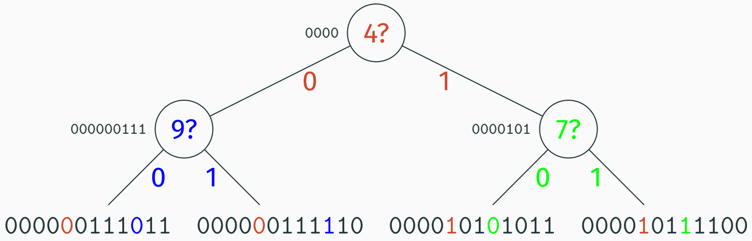


Insertion can require splitting an edge into an edge-node-edge in order to provide an anchor point for the new word (**did** in the example above). We split at the largest common prefix between the sub-word on the edge (**do**) and of the new word (**did**).

PATRICIA trees

(Donald R. Morrison, *PATRICIA – Practical Algorithm to Retrieve Information Coded in Alphanumeric*, 1968.)

Tries indexed by bit strings (integers), optimized for sparse tries.



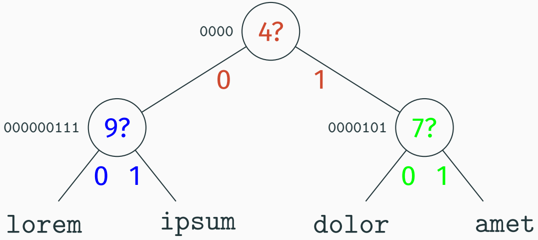
Each node carries the number of a bit to test (+ the prefix).

Keys are stored at the leaves.

Hash trees

(Phil Bagwell, *Ideal Hash Trees*, 2000.)

To represent sets $\{k_1, \ldots, k_n\}$ of keys of any type, we can transform these keys into bit strings (integers) using a hash function H, and store $k_1, \ldots, k_n$ at the leaves of a PATRICIA trie, with positions $H(k_1), \ldots, H(k_n)$.



Where $H(\texttt{"lorem"}) = 000000111011$, $H(\texttt{"dolor"}) = 000010101011$, etc.

Handling collisions

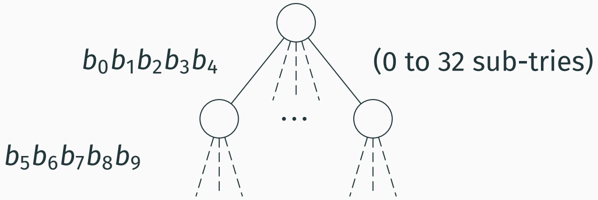
The hash function H takes finitely many values (typically, 32 or 64-bit integers). Hence there exists collisions: two different keys $k \neq k'$ that hash to the same bit string $H(k) = H(k')$.

Standard solution: at the leaves of the trie, put lists of keys $k, k', k'', \ldots$ that hash to the same value.

Bagwell's solution: "extend" the hash value by using several statistically-independent hash functions $H_0, H_1, \ldots$ In other words, we compute (on demand) a very long hash value $H_0(k).H_1(k) \ldots H_n(k) \ldots$ and we use the PATRICIA tree to distinguish between these hash values.

Hash Array Mapped Tries

Instead of a PATRICIA trie, HAMTs use a simple trie, but with a high branching degree, for instance $32 = 2^5$, hence hash values $H(k)$ are processed by groups of 5 bits.



(where $H(k) = b_0b_1b_2\dots$)

Hash Array Mapped Tries

Every node carries a partial function $f : [0, 31] \to \text{trie}$ implemented efficiently in space and in time as:

- A bit-vector B of size 32 (= a 32-bit machine integer).
  For each $i \in [0, 31]$, $B(i)$ says whether $f(i)$ is defined or not.
- An array T of n sub-tries,
  where n is the number of i's where $f(i)$ is defined.

Accessing the sub-trie labeled i is very efficient:

$$f(i) = \begin{cases} \text{undefined} & \text{if } B \ \& \ (1 \ \texttt{<<} \ i) = 0 \\ T[\mathtt{popcnt}(B \ \& \ ((1 \ \texttt{<<} \ i) - 1))] & \text{otherwise} \end{cases}$$

$\mathtt{popcnt}(n)$ is the Hamming weight of n (number of 1 bits).

Summary

Methodology

Using examples based on balanced trees, we saw two complementary approaches to developing persistent data structures:

1. Start from imperative, ephemeral data structures and introduce path copying to make them persistent.
2. Start from algebraic definitions of the data structure and its operations, and derive purely functional implementations.

Both approaches produce the same algorithms.

(2) facilitates specification and functional verification.
(1) is traditionally used for complexity analysis.

($\Rightarrow$ seminar by T. Nipkow)

Results

Persistent data structures with purely functional implementations for

- dictionaries, finite sets, finite maps;
- priority queues;
- indexed sequences.

All elementary operations run in time $\mathcal{O}(\log n)$.

A recurring question in later lectures:
how to improve on $\mathcal{O}(\log n)$?

References

References

Functional algorithms, with mechanized proofs of correctness and complexity:

- Tobias Nipkow *et al*, *Functional Algorithms, Verified!*, https://functional-algorithms-verified.org/

Imperative algorithms:

- D. E. Knuth, *The Art of Computer Programming*, volume 3, chapter 6, *Searching*.
- R. Sedgewick and K. Wayne, *Algorithms – 4th edition*, sections 3.2 and 3.3.