Proving tree algorithms for succinct data structures

Reynald Affeldt ¹  Jacques Garrigue ²
Xuanrui Qi ²  Kazunari Tanaka ²

¹National Institute of Advanced Industrial Science and Technology, Japan
²Graduate School of Mathematics, Nagoya University

September 9, 2019
https://github.com/affeldt-aist/succinct
Succinct Data Structures

- Representation optimized for both time and space
- “Compression without need to decompress”
- Much used for Big Data
- Application examples
  - Compression for Data Mining
  - Google’s Japanese IME
Rank and Select

To allow fast access, two primitive functions are heavily optimized. They can be computed in constant time.

- \(\text{rank}(i) = \) number of 1's up to position \(i\)

\[
\begin{array}{cccccccccccccccc}
\text{bitstring} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\text{indices} & 0 & 4 & 8 & 12 & 16 & 20 & 24 & 28 & 32 & 36 & 40 & 44 & 48 & 52 & \_ & \_ \\
\end{array}
\]

\(\text{rank}(4) = 2\) \hspace{1cm} \(\text{rank}(36) = 17\) \hspace{1cm} \(\text{rank}(58) = 26\)

- \(\text{select}(i) = \) position of the \(i^{th}\) 1: \(\text{rank}(\text{select}(i)) = i\)

\[
\begin{array}{cccccccccccccccc}
\text{bitstring} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\text{indices} & 0 & 4 & 8 & 12 & 16 & 20 & 24 & 28 & 32 & 36 & 40 & 44 & 48 & 52 & \_ & \_ \\
\end{array}
\]

\(\text{select}(2) = 4\) \hspace{1cm} \(\text{select}(17) = 36\) \hspace{1cm} \(\text{select}(26) = 57\)
Computing Rank in constant time

By using a two-level index, one can compute rank in constant time.

The size of the indexes is in $o(n)$.

Certified implementation [Tanaka A., Affeldt, Garrigue 2016]
CoQ specifications

rank counts occurrences of \((b : T)\).

\[
\text{Definition rank } i \ (s : \text{list } T) := \\
\quad \text{count_mem } b \ (\text{take } i \ s).
\]

select is its (minimal) inverse.

\[
\text{Definition select } i \ (s : \text{list } T) : \text{nat} := \\
\quad \text{index } i \ [\text{seq } \text{rank } k \ s \ | \ k \from \text{iota } 0 \ (\text{size } s).+1].
\]

\(\text{pred } s \ y\) is the last \(b\) before \(y\) (included).

\[
\text{Definition pred } s \ y := \text{select } (\text{rank } y \ s) \ s.
\]

\(\text{succ } s \ y\) is the first \(b\) after \(y\) (included).

\[
\text{Definition succ } s \ y := \text{select } (\text{rank } y.-1 \ s).+1 \ s.
\]

Getting the indexing right is challenging.
Here indices start from 1, but there is no fixed convention.
Today’s story

Trees in Succinct Data Structures

Featuring two views

Tree as sequence Encode the structure of a tree as a bit sequence, providing efficient navigation through rank and select

Sequence as tree Balanced trees (here red-black) can be used to encode dynamic bit sequences

• Both implemented and proved in Coq/SSReflect
• They can be combined together
L.O.U.D.S.

Level-Order Unary Degree Sequence
[Navarro 2016, Chapter 8]

- Unary coding of node arities, put in breadth-first order
- Each node of arity \( a \) is represented by \( a \) 1’s followed by 0
- The structure of a tree uses just \( 2n \) bits
- Useful for dictionaries (e.g. Google Japanese IME)
  - Allows to include a full Japanese dictionary in 50 MB
What is a Japanese IME?

- Incremental input
- Select a word in the dictionary according to a prefix
- Using LOUDS: each node contains one character; can collect them in a separate array
Implementation of primitives

Navigation primitives work by moving inside the LOUDS

The basic operations are

- Position of the $i^{th}$ child of a node
- Position of its parent
- Number of children

Variable $B : \text{list bool.} \quad (* \text{our LOUDS} *)$

Definition \text{LOUDS\_child $v \ i$ :=}
\begin{align*}
& \text{select false} \ (\text{rank true} \ (v \ + \ i) \ B).+1 \ B.
\end{align*}

Definition \text{LOUDS\_parent $v$ :=}
\begin{align*}
& \text{pred false} \ B \ (\text{select true} \ (\text{rank false} \ v \ B) \ B).
\end{align*}

Definition \text{LOUDS\_children $v$ :=}
\begin{align*}
& \text{succ false} \ B \ v.+1 \ - \ v.+1.
\end{align*}
LOUDS navigation

LOUDS\_parent \( v \) := \text{pred false} \ B \ (\text{select true} \ (\text{rank false} \ v \ B) \ B)

- \text{rank false} \ v \ B = 5 \text{ for } v = 14
  The number of nodes \( i \) before position \( v \).
- \text{select true} \ i \ B = 6 \text{ for } i = 5
  The position \( w \) of the branch leading to this node.
- \text{pred false} \ B \ w = 4 \text{ for } w = 6
  The position \( w' \) of the node containing this branch.
LOUDS navigation

\[
\begin{array}{cccc}
\text{level 0} & \text{level 1} & \text{level 2} & \text{level 3} \\
1110 & 11001110 & 000100 & 0 \\
\end{array}
\]

LOUDS\_parent \( v \) := \text{pred false} B (\text{select true} (\text{rank false} v B)

- \text{rank false} v B = 5 \text{ for } v = 14
  The number of nodes \( i \) before position \( v \).
- \text{select true} i B = 6 \text{ for } i = 5
  The position \( w \) of the branch leading to this node.
- \text{pred false} B w = 4 \text{ for } w = 6
  The position \( w' \) of the node containing this branch.
Proving tree algorithms for succinct data structures

Introduction
Rank&Select
Plan
LOUDS
Primitives
First attempt
Second try
Perspectives
Bonus

Dynamic data
Principle
Simply typed
Perspectives

LOUDS navigation

\[
\begin{array}{cccc}
\text{level 0} & \text{level 1} & \text{level 2} & \text{level 3} \\
1110 & 11001110 & 000100 & 0 \\
\end{array}
\]

LOUDS\_parent \( v := \text{pred false B (select true (rank false v B)} \)

- \( \text{rank false v B} = 5 \) for \( v = 14 \)
  The number of nodes \( i \) before position \( v \).
- \( \text{select true i B} = 6 \) for \( i = 5 \)
  The position \( w \) of the branch leading to this node.
- \( \text{pred false B w} = 4 \) for \( w = 6 \)
  The position \( w' \) of the node containing this branch.
LOUDS navigation

\[
\begin{array}{cccc}
\text{level 0} & \text{level 1} & \text{level 2} & \text{level 3} \\
1110 & 11001110 & 000100 & 0
\end{array}
\]

\[
\text{LOUDS\_parent } v := \text{pred false } B (\text{select true (rank false } v B)
\]

- \text{rank false } v B = 5 \text{ for } v = 14
  The number of nodes \( i \) before position \( v \).

- \text{select true } i B = 6 \text{ for } i = 5
  The position \( w \) of the branch leading to this node.

- \text{pred false } B w = 4 \text{ for } w = 6 \text{ (due to index shift)}
  The position \( w' \) of the node containing this branch.
Functional correctness

Assume an isomorphism `LOUDS_position` between valid paths in the tree, and valid positions in the LOUDS. Our 3 primitives shall satisfy the following invariants.

Definition `LOUDS_position (t : tree A) (p : list nat) : nat.
Variable `t : tree A.
Let `B := LOUDS t`.

Theorem `LOUDS_childE (p : list nat) (x : nat) : valid_position t (rcons p x) -> LOUDS_child B (LOUDS_position t p) x = LOUDS_position t (rcons p x).

Theorem `LOUDS_parentE (p : list nat) (x : nat) : valid_position t (rcons p x) -> LOUDS_parent B (LOUDS_position t (rcons p x)) = LOUDS_position t p.


How do we prove it ?
First attempt

Define traversal by recursion on the height of the tree.

Fixpoint LOUDS' n (s : forest A) :=
  if n is n'.+1 then
    map children_description s ++ LOUDS' n' (children_of_forest s)
  else [].
Definition LOUDS (t : tree A) := flatten (LOUDS' (height t) :: t).

Definition LOUDS_position (t : tree A) (p : list nat) :=
  lo_index t p + (lo_index t (rcons p 0)).-1.
(* number of 0's number of 1's *)

Theorem LOUDS_positionE t (p : list nat) :
  let B := LOUDS t in valid_position t p ->
  LOUDS_position t p = foldl (LOUDS_child B) 0 p.

lo_index t p is the number of valid paths preceding p in breadth first order.
First attempt

**Success**! Could prove the correctness of all primitives.
First attempt

Success! Could prove the correctness of all primitives.

Various problems

- Breadth first traversal does not follow the tree structure
- Cannot use structural induction
- No natural correspondence to use in proofs
- Oh, the indices!

As a result

- LOUDS related proofs took more than 800 lines
- Many lemmas had proofs longer than 50 lines
- There should be a better approach...
Proving tree algorithms for succinct data structures

Introduction

Rank & Select
Plan

LOUDS
Primitives
First attempt
Second try
Perspectives
Bonus

Dynamic data
Principle
Simply typed
Perspectives

Second try

- Introduce traversal up to a path: `lo_traversal_lt`
  Generalization of `lo_index`, returning a list
- For easy induction, work on forests rather than trees
- A generating forest need not be on the same level!
Traversals and Remainder

Parameters of the traversal

Variables \((A \ B : Type) \ (f : tree A \rightarrow B)\).

Traversal of the nodes preceding path \(p\)

**Fixpoint** \(lo\_traversal\_lt (s : forest A) (p : list nat) : list B\).

Generating forest for nodes following path \(p\), aka fringe

**Fixpoint** \(lo\_fringe (s : forest A) (p : list nat) : forest A\).

Relation between traversal and fringe

**Lemma** \(lo\_traversal\_lt\_cat s p1 p2 :\)
\[lo\_traversal\_lt s (p1 ++ p2) =\]
\[lo\_traversal\_lt s p1 ++ lo\_traversal\_lt (lo\_fringe s p1) p2.\]

All paths lead to Rome, i.e. complete traversals are all equal

**Theorem** \(lo\_traversal\_lt\_max t p :\)
\[size p \geq height t \rightarrow\]
\[lo\_traversal\_lt [] : t] p = lo\_traversal\_lt [] : t] (nseq (height t) 0).\]
Path, index, and position in LOUDS

Index of a node in level-order, using the traversal

**Definition** `lo_index s p := size (lo_traversal_lt id s p)`.

LOUDS_lt generates the LOUDS as a path-indexed traversal

**Definition** `LOUDS_lt s p := flatten (lo_traversal_lt children_description s p)`.

Use it to define the position of a node in the LOUDS

**Definition** `LOUDS_position s p := size (LOUDS_lt s p)`.

Main lemmas: relate position in LOUDS and index in traversal. Suffix `p'` allows completion to the whole LOUDS `t`.

**Lemma** `LOUDS_position_select s p p'`:
\[
\text{valid_position (head dummy s) p} \rightarrow \text{LOUDS_position s p} = \text{select false (lo_index s p) (LOUDS_lt s (p ++ p'))}.
\]

**Lemma** `lo_index_rank s p p' n`:
\[
\text{valid_position (head dummy s) (rcons p n)} \rightarrow \text{lo_index s (rcons p n)} = \text{size s + rank true (LOUDS_position s p + n) (LOUDSLt s (p ++ n :: p'))}.
\]
Advantages of the new approach

- Could prove naturally all invariants
- All proofs are by induction on paths
- Common lemmas arise naturally
- Only about 500 lines in total, long proofs about 20 lines

Remaining problems

- There are still longish lemmas (lo_index_rank, ...)
- Paths all over the place

Future work

- Can we apply that to other breadth-first traversals?
**Bonus: A Structural Traversal**

- `lo_traversal_lt` is nice, but still uses a path for induction
- How can we do a purely structural traversal?
Bonus: A Structural Traversal

- `lo_traversal_lt` is nice, but still uses a path for induction
- How can we do a purely structural traversal?
- The idea is to split the output in levels
- Then one can merge traversals by concatenating each level
- Gibbons and Jones gave a Squiggle algorithm in 1993, using the “long zip with plussle” `{\gamma_\oplus}`:

\[
\text{levels.} [x \triangleleft ts] = [x] :: \gamma_\oplus/.\text{levels.ts}
\]

where `{\gamma_M}` can be defined as `mzip` for any monoid `M`

```coq
Variable (A : Type) (e : A) (M : Monoid.law e).
Fixpoint mzip (l r : seq A) : seq A :=
match l, r with
| (l1::ls), (r1::rs) => (M l1 r1) :: mzip ls rs
| nil, s | s, nil => s
end.
```
mzip defines itself a new monoid, which we instantiate with the concatenation monoid

```
Lemma mzipA : associative mzip.
Lemma mzip1s s : mzip [::] s = s. Lemma mzips1 s : mzip s [::] = s.
Canonical mzip_monoid := Monoid.Law mzipA mzip1s mzips1.
```

```
Variables (A : eqType) (B : Type) (f : tree A -> B).
Definition mzip_cat := mzip_monoid (cat_monoid B).
```

```
Fixpoint level_traversal t := [:: f t] ::
    foldr (mzip_cat \o level_traversal) nil (children_of_node t).
Lemma level_traversalE t :
    level_traversal t = [:: f t] ::
    \big[mzip_cat/nil](i <- children_of_node t) level_traversal i.
```

```
Definition lo_traversal_st t := flatten (level_traversal t).
```

- To let Coq recognize the structural recursion, we have to use the recursor foldr in the definition of level_traversal
- The breadth-first traversal itself is lo_traversal_st
- Used morphism size o flatten o flatten \mapsto + to prove
  size (LOUDS t) = (number_of_nodes t) * 2 - 1
Dynamic succinct data structures

• Succinct data that can be updated (insertion/deletion)

• Concrete use cases: e.g. update in a dictionary

• Optimal static representation do not support updates. We cannot have both constant time rank/select and efficient insertion/deletion

• Using balanced trees, all operations are $O(\log n)$

[Navarro 2016, Chapter 12]
Dynamic bit sequence as tree

- `num` is the number of bits in the left subtree
- `ones` is the number of 1’s in the left subtree
Implementation

• Used red-black trees to implement
  • complexity is the same for all balanced trees
  • easy to represent in a functional style
  • already several implementations in Coq
  • however we need a different data layout with new invariants, so we had to reimplement

• Two implementations using types differently
  ① simply typed implementations, with invariants expressed as separate theorems
  ② dependent types, directly encoding all the required invariants (explained yesterday in Coq workshop)

• We implemented rank, select, insert and delete
Simply typed implementation

A red-black tree for bit sequences

Inductive color := Red | Black.

Inductive btree (D A : Type) : Type :=
| Bnode of color & btree D A & D & btree D A
| Bleaf of A.

Definition dtree := btree (nat * nat) (list bool).

The meaning of the tree is given by dflatten

Fixpoint dflatten (B : dtree) :=
    match B with
    | Bnode _ l _ r => dflatten l ++ dflatten r
    | Bleaf s => s
end.

Invariants on the internal representation

Variables low high : nat.
Fixpoint wf_dtree (B : dtree) :=
    match B with
    | Bnode _ l (num, ones) r => [&& num == size (dflatten l),
      ones == count_mem true (dflatten l), wf_dtree l & wf_dtree r]
    | Bleaf arr => low <= size arr < high
end.
Basic operations

Fixpoint drank (B : dtree) (i : nat) := match B with
  | Bnode _ l (num, ones) r =>
      if i < num then drank l i else ones + drank r (i - num)
  | Bleaf s => rank true i s
end.

Lemma drankE (B : dtree) i :
  wf_dtree B -> drank B i = rank true i (dflatten B).
Proof. move=> wf; move: B wf i. apply: dtree_ind. (*) Qed.

Fixpoint dselect_1 (B : dtree) (i : nat) := match B with
  | Bnode _ l (num, ones) r =>
      if i <= ones then dselect_1 l i
      else num + dselect_1 r (i - ones)
  | Bleaf s => select true i s
end.

Lemma dselect_1E B i :
  wf_dtree B -> dselect_1 B i = select true i (dflatten B).

where dtree_ind is a custom induction principle.
All proofs are only a few lines long.
**Insertion**

**Definition** \(\text{dins}_\text{leaf} \ s \ b \ i :=\)

\[
\text{let } s' := \text{insert1} \ s \ b \ i \text{ in } (* \text{insert bit } b \text{ in } s \text{ at position } i *) \\
\text{if } \text{size} \ s + 1 = \text{high} \text{ then} \\
\text{let } n := \text{size} \ s \div 2 \text{ in} \\
\text{let } sl := \text{take} \ n \ s' \text{ in} \text{ let } sr := \text{drop} \ n \ s' \text{ in} \\
\text{Bnode Red (Bleaf } _ \text{ sl}) (n, \text{count_mem true} sl) (\text{Bleaf } _ \text{ sr}) \\
\text{else Bleaf } _ s'.
\]

**Fixpoint** \(\text{dins} (B : \text{dtree}) b i : \text{dtree} := \text{match } B \text{ with} \)

| \text{Bleaf } s \Rightarrow \text{dins}_\text{leaf} \ s \ b \ i |
| \text{Bnode } c \ l d r \Rightarrow |
| \text{if } i < d.1 \text{ then balanceL } c (\text{dins} \ l \ b \ i) r (d.1.+1, d.2 + b) |
| \text{else balanceR } c \ l (\text{dins} \ r \ b \ (i - d.1)) \ d |

end.

**Definition** \(\text{dinsert } B b i : \text{dtree} := \text{blacken} (\text{dins } B \ b \ i).\)

**The real work is in balanceL/balanceR**
Proving tree algorithms for succinct data structures

Introduction

Rank&Select Plan

LOUDS Primitives First attempt Second try Perspectives Bonus

Dynamic data Principle Simply typed Perspectives

Balancing

Variables  \( \text{addD \ subD : D \rightarrow D \rightarrow D.} \)

Definition  \( \text{balanceL \ col \ (l \ r : btree D A) \ dl : btree D A :=} \)

match  \( \text{col with} \)
  \( \text{Red} \Rightarrow \)  \( \text{Bnode Red l dl r} \)
  \( \text{Black} \Rightarrow \) match  \( \text{l with} \)
    \( \text{Bnode Red (Bnode Red a da b) dab c} \Rightarrow \)
    \( \text{Bnode Red (Bnode Black a da b) dab} \)
    \( \text{(Bnode Black c (subD dl dab) r)} \)
    \( \text{Bnode Red a da (Bnode Red b db c) \Rightarrow} \)
    \( \text{Bnode Red (Bnode Black a da b) (addD da db)} \)
    \( \text{(Bnode Black c (subD (subD dl da) db) r)} \)
    \( _\Rightarrow \text{Bnode Black l dl r} \)

end.

• Separated  \( \text{balanceL and balanceR} \)

• This avoids creating two many cases during the proof
Balancing

- Number of cases is the main difficulty for red-black trees
- Expanding balanceL generates 11 cases
- Following SSReflect style, we avoid opaque automation

Ltac decompose_rewrite :=
  let H := fresh "H" in
  case/andP || (move=>H; rewrite ?H ?(eqP H)).

Lemma balanceL_wf c (l r : dtree) :
  wf_dtree l -> wf_dtree r -> wf_dtree (balanceL c l r).
Proof.
case: c => /= wfl wfr. by rewrite wfl wfr ?(dsizeE,donesE,eqxx).
case: l wfl =>
  [[][] lll [lln llo] llr|llA] [ln lo] [[] lrl [lrn lro] lrr|lrA]
  |ll [ln lo] lr|lA] /=;
  rewrite wfr; repeat decompose_rewrite;
  by rewrite ?(dsizeE,donesE,size_cat,count_cat,eqxx).
Qed.
Properties of insertion

Functional correctness

Lemma \( \text{dinsertE} \ (B : \text{dtree}) \ b \ i : \text{wf_dtree}' \ B \rightarrow \ d\text{flatten} \ (\text{dinsert} \ B \ b \ i) = \text{insert1} \ (d\text{flatten} \ B) \ b \ i. \)

Well-formedness and red-black invariants

Lemma \( \text{dinsert_wf} \ (B : \text{dtree}) \ b \ i : \)
\( \text{wf_dtree}' \ B \rightarrow \text{wf_dtree}' \ (\text{dinsert} \ B \ b \ i). \)
Lemma \( \text{dinsert_is_redblack} \ (B : \text{dtree}) \ b \ i \ n : \)
\( \text{is_redblack} \ B \ \text{Red} \ n \rightarrow \exists n', \text{is_redblack} \ (\text{dinsert} \ B \ b \ i) \ \text{Red} \ n'. \)

where

- \( \text{wf_dtree}' \) is needed for small sequences

Definition \( \text{wf_dtree}' \ t := \)
\( \text{if } t \text{ is } \text{Bleaf} \ s \text{ then } \text{size} \ s < \text{high} \ \text{else} \ \text{wf_dtree} \text{ low high t.} \)

- \( \text{is_redblack} \) checks the red-black tree invariants:
  - the child of a red node cannot be red
  - both children have the same black depth
Deletion

The mysterious side

- Omitted in Okasaki’s Book
- Enigmatic algorithm by Stefan Kahrs, with an invariant but no details

Chose to rediscover it

- Started with dependent types, guessing invariants
- Used extraction to retrieve the computational part
- Rewrote and proved the simply typed version
  Proofs are small, but use Ltac for repetitive cases.
- As case analysis generates hundreds of cases, performance can be a problem.

Lemma ddelete_is_redblack B i n :
  is_redblack B Red n -> exists n', is_redblack (ddel B i) Red n'.
Deletion main function

Fixpoint bdel B (i : nat) { struct B } : deleted_btree :=
  match B with
  | Bnode c (Bleaf l) d (Bleaf r) => delete_from_leaves c l r i
  | Bnode Black (Bnode Red (Bleaf ll) ld (Bleaf lr) as l) d (Bleaf r) =>
    if lt_index i d
    then balanceL' Black (bdel l i) d (Bleaf _ r)
    else balanceR' Black (Bleaf _ ll) ld
        (delete_from_leaves Red lr r (right_index i ld))
  | Bnode Black (Bleaf l) ld (Bnode Red (Bleaf rl) d (Bleaf rr) as r) =>
    if lt_index (right_index i ld) d
    then balanceL' Black (delete_from_leaves Red l rl i)
        (addD ld d) (Bleaf _ rr)
    else balanceR' Black (Bleaf _ l) ld (bdel r (right_index i ld))
  | Bnode c l d r =>
    if lt_index i d
    then balanceL' c (bdel l i) d r
    else balanceR' c l d (bdel r (right_index i d))
  | Bleaf x =>
    let (leaf, ret) := delete_leaf x i in
    MkD (Bleaf _ leaf) false ret
end.
Dynamic bit sequence perspectives

- Simply typed approach
  - SSReflect style worked well, providing short and maintainable proofs
  - could obtain proofs of balancing without complex machinery (just automatic case analysis)
  - however many small lemmas are required

- Dependently typed version
  - all properties are in the types, no need for dispersed proofs
  - Coq support not perfect yet

- Future work
  - We have not yet started working on complexity
  - We also need to extract efficient implementations

https://github.com/affeldt-aist/succinct