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 \( i \)

\[
\begin{align*}
\text{bitstring} & : 1001\ 0100\ 1110\ 0100\ 1101\ 0000\ 1111\ 0100\ 1001\ 1001\ 0100\ 0100\ 0101\ 0101\ 10 \\
\text{indices} & : 0\ \ 4\ \ 8\ \ 12\ \ 16\ \ 20\ \ 24\ \ 28\ \ 32\ \ 36\ \ 40\ \ 44\ \ 48\ \ 52\ \ n-2 = 56 \\
\text{rank}(4) & = 2 \quad \text{rank}(36) = 17 \quad \text{rank}(58) = 26
\end{align*}
\]

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

\[
\begin{align*}
\text{bitstring} & : 1001\ 0100\ 1110\ 0100\ 1101\ 0000\ 1111\ 0100\ 1001\ 1001\ 0100\ 0100\ 0101\ 0101\ 10 \\
\text{indices} & : 0\ \ 4\ \ 8\ \ 12\ \ 16\ \ 20\ \ 24\ \ 28\ \ 32\ \ 36\ \ 40\ \ 44\ \ 48\ \ 52\ \ n-2 = 56 \\
\text{select}(2) & = 4 \quad \text{select}(17) = 36 \quad \text{select}(26) = 57
\end{align*}
\]

Certified implementation of rank [Tanaka A., Affeldt, Garrigue 2016]
Coq definitions

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.

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

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

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

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.
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
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)
What is a Japanese IME?

- Incremental input
- Select a word in the dictionary according to a prefix
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 $B : \text{list bool}$. (* our LOUDS *)

Definition \text{LOUDS}_\text{child} v i :=

\[
\text{select false (rank true (v + i) B).+1 B.}
\]

Definition \text{LOUDS}_\text{parent} v :=

\[
\text{pred false B (select true (rank false v B) B).}
\]

Definition \text{LOUDS}_\text{children} v :=

\[
\text{succ false B v.+1 - v.+1.}
\]

Implementation of primitives
LOUDS navigation

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

- \( \text{rank false} \ v \ B = 5 \) 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**

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

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

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

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

- **pred false** \( B \ w = 4 \) for \( w = 6 \)
  The position \( w' \) of the node containing this branch.
LOUDS navigation

```
level 0 | level 1               | level 2 | level 3
--------|-----------------------|--------|--------
1110    | 11001110              | 000100 | 0
```

LOUDS_parent \( v \) := \text{pred false } B \left( \text{select true } (\text{rank false } v \ B) \right)

- \( \text{rank false } v \ B = 5 \) 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

![LOUDS navigation diagram]

<table>
<thead>
<tr>
<th>level 0</th>
<th>level 1</th>
<th>level 2</th>
<th>level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>11001110</td>
<td>000100</td>
<td>0</td>
</tr>
</tbody>
</table>

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.
Functional correctness

Assume an isomorphism \texttt{LOUDS\_position} between valid paths in the tree, and valid positions in the LOUDS.
Our 3 primitives shall satisfy the following invariants.

\begin{verbatim}
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.

Theorem LOUDS_childrenE (p : list nat) :
valid_position t p ->
children t p = LOUDS_children B (LOUDS_position t p).
\end{verbatim}

Proved using two approaches.
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...
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!
Proving tree algorithms for succinct data structures

Introduction

Rank&Select

Plan

LOUDS

Primitives

First attempt

Second try

Perspectives

Dynamic data

Principle

Simply typed

Perspectives

Traversals and Remainder

Parameters of the traversal

Variables \((A \ B : \text{Type}) (f : \text{tree } A \to B)\).

Traversal of the nodes preceding path \(p\)

Fixpoint \(\text{lo_traversal}_{-\text{lt}} (s : \text{forest } A) (p : \text{list } \text{nat}) : \text{list } B\).

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

Fixpoint \(\text{lo_fringe} (s : \text{forest } A) (p : \text{list } \text{nat}) : \text{forest } A\).

Relation between traversal and fringe

Lemma \(\text{lo_traversal}_{-\text{lt}}_{-\text{cat}} s p1 p2 :\)

\(\text{lo_traversal}_{-\text{lt}} s (p1 ++ p2) =\)

\(\text{lo_traversal}_{-\text{lt}} s p1 ++ \text{lo_traversal}_{-\text{lt}} (\text{lo_fringe } s p1) p2.\)

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

Theorem \(\text{lo_traversal}_{-\text{lt}}_{-\text{max}} t p :\)

\(\text{size } p >= \text{height } t \to\)

\(\text{lo_traversal}_{-\text{lt}} [:: t] p = \text{lo_traversal}_{-\text{lt}} [:: t] (\text{nseq } (\text{height } t \ 0)).\)
Path, index, and position in LOUDS

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

**Definition** \(\text{lo\_index } s\ p := \text{size } (\text{lo\_traversal\_lt } \text{id } s\ p)\).

LOUDS\_lt generates the LOUDS as a path-indexed traversal

**Definition** \(\text{LOUDS\_lt } s\ p := \text{flatten } (\text{lo\_traversal\_lt } \text{children\_description } s\ p)\).

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

**Definition** \(\text{LOUDS\_position } s\ p := \text{size } (\text{LOUDS\_lt } s\ p)\).

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

**Lemma** \(\text{LOUDS\_position\_select } s\ p\ p' : \text{valid\_position } (\text{head } \text{dummy } s)\ p \rightarrow \text{LOUDS\_position } s\ p\ =\ \text{select } \text{false } (\text{lo\_index } s\ p)\ (\text{LOUDS\_lt } s\ (p\ ++\ p')))\).

**Lemma** \(\text{lo\_index\_rank } s\ p\ p'\ n : \text{valid\_position } (\text{head } \text{dummy } s)\ (rcons\ p\ n) \rightarrow \text{lo\_index } s\ (rcons\ p\ n) = \text{size } s\ +\ \text{rank } \text{true } (\text{LOUDS\_position } s\ p\ +\ n)\ (\text{LOUDS\_lt } s\ (p\ ++\ n::p')))\).
LOUDS perspectives

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?
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

\textbf{Inductive} color := Red | Black.
\textbf{Inductive} btree (D A : Type) : Type :=
| Bnode of color & btree D A & D & btree D A
| Bleaf of A.
\textbf{Definition} dtree := btree (nat * nat) (list bool).

The meaning of the tree is given by \texttt{dflatten}

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

Invariants on the internal representation

\textbf{Variables} low high : nat.
\textbf{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

\begin{verbatim}
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).
\end{verbatim}

where \texttt{dtree\_ind} is a custom induction principle.
All proofs are only a few lines long.
Definition \texttt{dins	extunderscore leaf} \texttt{s b i} :=
\begin{align*}
\text{let } s' & := \text{insert1 } s \text{ b i in (* insert bit b in s at position i *)} \\
\text{if } \text{size } s + 1 & = \text{high then} \\
\text{let } n & := \text{size } s' \div 2 \text{ in} \\
\text{let } sl & := \text{take } n \text{ s'} \text{ in let } sr & := \text{drop } n \text{ s'} \text{ in} \\
\text{Bnode Red (Bleaf } \_ \text{ sl) } (n, \text{count	extunderscore mem true sl}) & \text{ (Bleaf } \_ \text{ sr) } \\
\text{else } & \text{Bleaf } \_ \text{ s'.}
\end{align*}

\textbf{Fixpoint} \texttt{dins} \texttt{(B : dtree) b i : dtree := match B with} \\
\begin{align*}
| \text{Bleaf s} & \Rightarrow \texttt{dins	extunderscore leaf } s \text{ b i} \\
| \text{Bnode c l d r} & \Rightarrow \\
\text{if } i \leq d.1 & \text{ then balanceL c } (\texttt{dins} l \text{ b i}) \text{ r } (d.1.+1, \text{d.2} + b) \\
\text{else } & \text{balanceR c l } (\texttt{dins} r \text{ b } (i - d.1)) \text{ d}
\end{align*}

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

\textbf{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

Dynamic data

Principle

Simply typed

Perspectives

Balancing

• Number of cases is the main difficulty for red-black trees
• Expanding balance\text{l} generates 11 cases
• Following \text{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 =>
        rewrite wfr; repeat decompose_rewrite;
        by rewrite ?(dsizeE,donesE,size_cat,count_cat,eqxx).
Qed.
Properties of insertion

Functional correctness

**Lemma** dinsertE (B : dtree) b i : wf_dtree' B ->

dflatten (dinsert B b i) = insert1 (dflatten B) b i.

Well-formedness and red-black invariants

**Lemma** dinsert_wf (B : dtree) b i :

wf_dtree' B -> wf_dtree' (dinsert B b i).

**Lemma** dinsert_is_redblack (B : dtree) b i n :

is_redblack B Red n ->

exists n', is_redblack (dinsert B b i) Red n'.

where

- **wf_dtree'** is needed for small sequences

**Definition** wf_dtree' t :=

if t is Bleaf s then size s < high else wf_dtree low high t.

- **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'.
Dynamic bit sequence perspectives

- Simply typed approach
  - \texttt{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