Proving tree algorithms for succinct data structures

Reynald Affeldt [1] Jacques Garrigue [2]
Xuanrui Qi [3] Kazunari Tanaka [2]

[1]AIST

[2]Nagoya University

[3]Tufts University

October 19, 2018

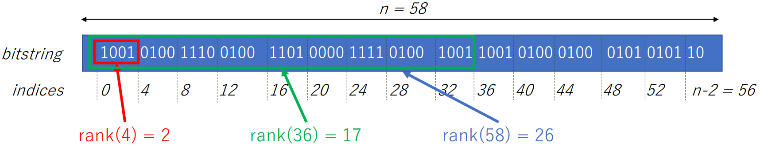
Succinct data structures

- Optimized for both time and space
- "compressed with no need to decompress"
- Many uses in big data
- Examples
  - Data compression for data mining
  - Dictionary of Google Japanese input method

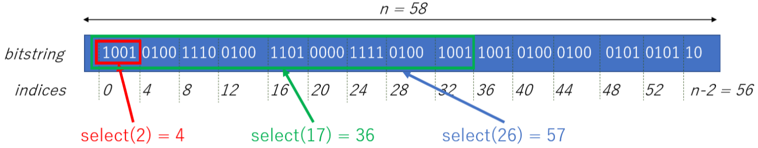
Rank and Select

To provide fast access and search in bit sequences, 2 specific primitives are optimized. Usually work in constant time.

- rank(i) = number of 1's up to the i^{th} *bit*



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



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

Today's story

Encoding and uses of trees in succinct data structures

Two viewpoints

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
- Can use the first on top of the second

Basic Coq definitions

rank can be defined easily. select is its inverse.

```
Variables (T : eqType) (b : T) (n : nat).
Definition rank i s := count_mem b (take i s).
Definition Rank (i : nat) (B : n.-tuple T) :=
  #|[set k : [1,n] | (k <= i) && (tacc B k == b)]|.
Lemma select_spec (i : nat) (B : n.-tuple T) :
  exists k, ((k <= n) && (Rank b k B == i)) ||
             (k == n.+1) && (count_mem b B < i).
Definition Select i (B : n.-tuple T) :=
  ex_minn (select_spec i B).
```

pred s y is the last b up to y. succ s y if the first b from y.

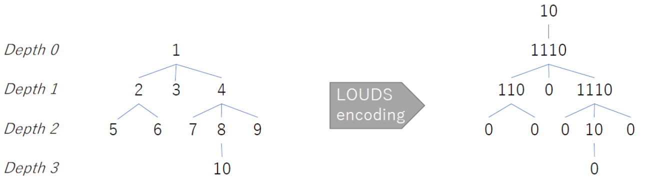
```
Definition pred s y := select (rank y s) s.
Definition succ s y := select (rank y.-1 s).+1 s.
```

Hard to set the indices correctly.
Here we use indices starting from 1, but it varies among books.

L.O.U.D.S.

Level-**O**rder **U**nary **D**egree **S**equence

[Navarro 2016, Chapter 8]



Depth 0	*Depth 1*	*Depth 2*	*Depth 3*
1	234	56789	10

	Depth 0	*Depth 1*	*Depth 2*	*Depth 3*
10	1110	11001110	000100	0

- Breadth first sequence of the unary representations of node arities
- Each node is represented by *a* 1's followed by a 0
- The structure of *n*-node tree is represented by exactly $2n + 2$ bits
- Applications to dictionaries (cf. Google IME)

Basic operations

We define a bijection between paths in the tree and positions in the LOUDS.

Required operations:

- Position of the root (2, just after the virtual root at 0)
- Position of i^{th} child
- Position of the parent node
- Number of children

```
Variable B : seq bool.
Definition LOUDS_child v i :=
  select false (rank true (v + i) B).+1 B.
Definition LOUDS_parent v :=
  pred false B (select true (rank false v B) B).
Definition LOUDS_children v := succ false B v.+1 - v.+1.
```

Basic operations

We define a bijection between paths in the tree and positions in the LOUDS.

Required operations:

- Position of the root (2, just after the virtual root at 0)
- Position of i^{th} child
- Position of the parent node
- Number of children

```
Variable B : seq bool.
Definition LOUDS_child v i :=
  select false (rank true (v + i) B).+1 B.
Definition LOUDS_parent v :=
  pred false B (select true (rank false v B) B).
Definition LOUDS_children v := succ false B v.+1 - v.+1.
```

Problem: This correspondence is not structural

Concrete bijection

count_smaller t p counts the number of nodes appearing before the one at path p in breadth first order

```
Definition LOUDS_position (t : tree) (p : seq nat) :=
  (count_smaller t p + (count_smaller t (rcons p 0)).-1).+2.
(*       0's                    1's                    virtual root *)

Definition LOUDS_subtree B (p : seq nat) :=
  foldl (LOUDS_child B) 2 p.

Theorem LOUDS_positionE t (p : seq nat) :
  let B := LOUDS t in valid_position t p ->
  LOUDS_position t p = LOUDS_subtree B p.

Theorem LOUDS_parentE t (p : seq nat) x :
  let B := LOUDS t in valid_position t (rcons p x) ->
  LOUDS_parent B (LOUDS_position t (rcons p x)) = LOUDS_position t p.

Theorem LOUDS_childrenE t (p : seq nat) :
  let B := LOUDS t in valid_position t p ->
  children t p = LOUDS_children B (LOUDS_position t p).
```

Difficulties with L.O.U.D.S

Many problems

- Breadth-first traversal is far from the structure of the tree
- One cannot use structural induction, only depth-induction on a forest
- The correspondance we defined is not "natural"
- Indices very hard to apprehend for a human brain

As a result

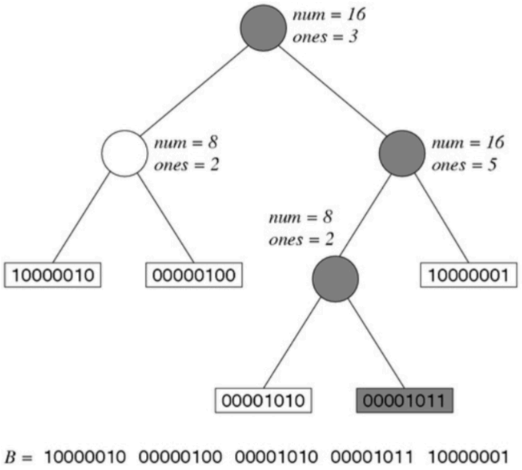
- The proof for LOUDS is about 800 lines
- Many lemmas require more than 50 lines
- Still looking for a better approach

Dynamic succinct data structures

- The optimal representation for static succinct data structures use arrays, which are not good for dynamic insertion/deletion
- There are concrete use case for dynamic succinct data structures
- 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 buts on the left, *ones* the number of 1's on the left

Implementation

- Use 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
- 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) (seq 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

```
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 => (w ^ 2)./2 <= size arr < (w ^ 2).*2
  end.
```

Simply typed 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 dtree_ind (P : dtree -> Prop) :
  (forall c l r num ones,
   num = size (dflatten l) ->
   ones = count_mem true (dflatten l) ->
   wf_dtree l /\ wf_dtree r ->
   P l -> P r -> P (Bnode c l (num, ones) r)) ->
  (forall s, (w ^ 2)./2 <= size s < (w ^ 2).*2 -> P (Bleaf _ s)) ->
  forall B, wf_dtree B -> P B.

Lemma drankE (B : dtree) i :
  wf_dtree B -> drank B i = rank true i (dflatten B).
```

All proofs are only a few line long

Simply typed basic operations

```
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.

Fixpoint dselect_0 (B : dtree) (i : nat) :=
  match B with
  | Bnode _ l (num, ones) r =>
    let zeroes := num - ones in
    if i <= zeroes then dselect_0 l i
                   else num + dselect_0 r (i - zeroes)
  | Bleaf s => select false i s
  end.

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

Lemma dselect_0E B i :
 wf_dtree B -> dselect_0 B i = select false i (dflatten B).
```

Simplify typed insertion

```
Fixpoint dins (B : dtree) b i w : dtree :=
  match B with
  | Bleaf s =>
    let s' := insert1 s b i in
    if size s + 1 == 2 * (w ^ 2)
    then let n  := (size s') %/ 2 in
         let sl := take n s' in
         let sr := drop n s' in
         Bnode Red (Bleaf _ sl)
               (size sl, rank true (size sl) sl)
               (Bleaf _ sr)
    else Bleaf _ s'
  | Bnode c l (num, ones) r =>
    if i < num then balanceL c (dins l b i w) r
               else balanceR c l (dins r b (i - num) w)
  end.

Definition dinsert (B : dtree) b i w : dtree :=
  match dins B b i w with
  | Bleaf s => Bleaf _ s
  | Bnode _ l d r => Bnode Black l d r
  end.
```

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.
```

Dependently typed definition

All the invariants are in the tree

- as a dynamic bit sequence
- as a red-black tree

```
Definition is_black c := if c is Black then true else false.
Definition color_ok parent child :=
    is_black parent || is_black child.

Inductive tree : nat -> nat -> nat -> color -> Type :=
  | Leaf : forall (arr : seq bool),
      (w ^ 2)./ 2 <= size arr < (w ^ 2).*2 ->
      tree (size arr) (count_one arr) 0 Black
  | Node : forall {s1 o1 s2 o2 d cl cr c},
      color_ok c cl -> color_ok c cr ->
      tree s1 o1 d cl -> tree s2 o2 d cr ->
      tree (s1 + s2) (o1 + o2) (d + is_black c) c.
```

Dependently typed operations

- Definition of basic operations almost unchanged
- No need for dtree_ind
- Could define dins using the Program environment

```
Program Fixpoint dinsert' {n m d c} (B : tree n m d c) (b : bool) i
    {measure (size_of_tree B)} : { B' : near_tree n.+1 (m + b) d c
                  | dflattenn B' = insert1 (dflatten B) b i } := ...
```

Generates 20 proof obligations, for a total of about 90 lines

- Defining balanceL and balanceR
  - The Program environment is almost unusable there
  - Could define (and prove) it in 17 lines of Ltac

```
Definition balanceL {nl ml d cl cr nr mr} (p : color)
    (l : near_tree nl ml d cl) (r : tree nr mr d cr) :
    color_ok p (fix_color l) -> color_ok p cr ->
    {tr : near_tree (nl + nr) (ml + mr) (inc_black d p) p
    | dflattenn tr = dflattenn l ++ dflatten r}.
  destruct r as [s1 o1 s2 o2 s3 o3 d' x y z | s o d' c' cc r'].
  + case: p => //= cpl cpr.
  (* 11 more lines of definition/proof *)
Defined.
```

Deletion

The mysterious side

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

Chose to rediscover it

- Start with the dependently typed version
  4 definitions: `merge_arrays`, `delete_leaves`, `balanceL2`, `balanceR2`, `ddelete`; all huge
- Use extraction to retrieve the computational part
- Rewrite and prove the simply typed version
  Proofs are small, except for the final one, 25 lines long

```
Lemma ddel_is_nearly_redblack' B i n c :
  0 < n -> is_redblack B c n ->
  is_nearly_redblack' (ddel B i) c n.
```

Dynamic bit sequences perspectives

- Simply typed approach
  - SSReflect style worked well, providing short and readable proofs
  - proof of balancing is more intuitive than in previous approaches
  - however many small lemmas are required
- Dependently typed version
  - all properties are in the types, no need for dispersed proofs
  - due to limitations in the Program environment, the definition must be rewritten
  - fixing the proofs after changes is painful
- Future work
  - We have not yet started working on complexity
  - First step: what would be a good definition?

Dynamic bit sequences perspectives

- Simply typed approach
  - SSReflect style worked well, providing short and readable proofs
  - proof of balancing is more intuitive than in previous approaches
  - however many small lemmas are required
- Dependently typed version
  - all properties are in the types, no need for dispersed proofs
  - due to limitations in the Program environment, the definition must be rewritten
  - fixing the proofs after changes is painful
- Future work
  - We have not yet started working on complexity
  - First step: what would be a good definition?

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