

# van Emde Boas Trees

Thomas Ammer and Peter Lammich

May 26, 2024

## Abstract

The *van Emde Boas tree* or *van Emde Boas priority queue* [1, 2] is a data structure supporting membership test, insertion, predecessor and successor search, minimum and maximum determination and deletion in  $\mathcal{O}(\log \log |\mathcal{U}|)$  time, where  $\mathcal{U} = \{0, \dots, 2^n - 1\}$  is the overall range to be considered. The presented formalization follows Chapter 20 of the popular *Introduction to Algorithms (3rd ed.)* [3] by Cormen, Leiserson, Rivest and Stein (CLRS), extending the list of formally verified CLRS algorithms [4]. Our current formalization is based on the first author's bachelor's thesis.

First, we prove correct a *functional* implementation, w.r.t. an abstract data type for sets. Apart from functional correctness, we show a resource bound, and runtime bounds w.r.t. manually defined timing functions [5] for the operations.

Next, we refine the operations to Imperative HOL [6, 7] with time [8], and show correctness and complexity. This yields a practically more efficient implementation, and eliminates the manually defined timing functions from the trusted base of the proof.

# Contents

<b>1 Preliminaries and Preparations</b>	<b>5</b>
1.1 Data Type Definition . . . . .	5
1.2 Functions for obtaining high and low bits of an input number. . . . .	5
1.3 Some auxiliary lemmata . . . . .	5
1.4 Auxiliary functions for defining valid Van Emde Boas Trees . . . . .	6
1.5 Inductive Definition of semantically valid Van Emde Boas Trees . . . . .	6
1.6 Function for generating an empty tree of arbitrary degree respectively order . .	9
<b>2 Member Function</b>	<b>10</b>
<b>3 Insert Function</b>	<b>13</b>
<b>4 Correctness of the Insert Operation</b>	<b>14</b>
4.1 Validness Preservation . . . . .	14
4.2 Correctness with Respect to Set Interpretation . . . . .	14
<b>5 The Minimum and Maximum Operation</b>	<b>15</b>
<b>6 The Successor Operation</b>	<b>18</b>
6.1 Auxiliary Lemmas on Sets and Successorship . . . . .	18
6.2 The actual Function . . . . .	19
6.3 Lemmas for Term Decomposition . . . . .	19
6.4 Correctness Proof . . . . .	20
<b>7 The Predecessor Operation</b>	<b>20</b>
7.1 Lemmas on Sets and Predecessorship . . . . .	21
7.2 The actual Function for Predecessor Search . . . . .	21
7.3 Auxiliary Lemmas . . . . .	22
7.4 Correctness Proof . . . . .	22
<b>8 Deletion</b>	<b>23</b>
8.1 Function Definition . . . . .	23
8.2 Auxiliary Lemmas . . . . .	24
8.3 Validness Preservation . . . . .	29
8.4 Correctness with Respect to Set Interpretation . . . . .	30
<b>9 Uniqueness Property of valid Trees</b>	<b>30</b>
<b>10 Heights of van Emde Boas Trees</b>	<b>31</b>
<b>11 Upper Bounds for canonical Functions: Relationships between Run Time and Tree Heights</b>	<b>32</b>
11.1 Membership test . . . . .	32
11.2 Minimum, Maximum, Emptiness Test . . . . .	33

11.3	Insertion . . . . .	34
11.4	Successor Function . . . . .	35
11.5	Predecessor Function . . . . .	36
11.6	Running Time Bounds for Deletion . . . . .	37
<b>12</b>	<b>Space Complexity and <i>buildup</i> Time Consumption</b>	<b>39</b>
12.1	Space Complexity of valid van Emde Boas Trees . . . . .	39
12.2	Auxiliary Lemmas for List Summation . . . . .	40
12.3	Actual Space Reasoning . . . . .	41
12.4	Complexity of Generation Time . . . . .	41
<b>13</b>	<b>Functional Interface</b>	<b>43</b>
13.1	Code Generation Setup . . . . .	43
13.1.1	Code Equations . . . . .	43
13.2	Correctness Lemmas . . . . .	46
13.2.1	Space Bound . . . . .	46
13.2.2	Buildup . . . . .	46
13.2.3	Equality . . . . .	46
13.2.4	Member . . . . .	46
13.2.5	Insert . . . . .	47
13.2.6	Maximum . . . . .	47
13.2.7	Minimum . . . . .	47
13.3	Emptiness determination . . . . .	48
13.3.1	Successor . . . . .	48
13.3.2	Predecessor . . . . .	48
13.3.3	Delete . . . . .	48
13.4	Interface Usage Example . . . . .	49
13.5	Lists . . . . .	50
<b>14</b>	<b>Imperative van Emde Boas Trees</b>	<b>54</b>
14.1	Assertions on van Emde Boas Trees . . . . .	54
14.2	High and low Bitsequences Definition . . . . .	56
<b>15</b>	<b>Imperative Implementation of <i>vebt</i> – <i>buildup</i></b>	<b>57</b>
<b>16</b>	<b>Minimum and Maximum Determination</b>	<b>60</b>
<b>17</b>	<b>Membership Test on imperative van Emde Boas Trees</b>	<b>61</b>
17.1	<i>minNulli</i> : empty tree? . . . . .	63
<b>18</b>	<b>Imperative <i>vebt</i> – <i>insert</i> to van Emde Boas Tree</b>	<b>64</b>
<b>19</b>	<b>Imperative Successor</b>	<b>66</b>
<b>20</b>	<b>Imperative Predecessor</b>	<b>69</b>

<b>21 Imperative Delete</b>	<b>71</b>
<b>22 Imperative Interface</b>	<b>75</b>
22.1 Code Export . . . . .	75
22.2 Interface . . . . .	76
22.2.1 Buildup . . . . .	76
22.2.2 Member . . . . .	76
22.2.3 Insert . . . . .	76
22.2.4 Maximum . . . . .	76
22.2.5 Minimum . . . . .	76
22.2.6 Successor . . . . .	76
22.2.7 Predecessor . . . . .	77
22.2.8 Delete . . . . .	77
22.3 Setup of VCG . . . . .	77
<b>23 Interface Usage Example</b>	<b>77</b>
23.1 Test Program . . . . .	77
23.2 Correctness without Time . . . . .	77
23.3 Time Bound Reasoning . . . . .	78
<b>24 Conclusion</b>	<b>79</b>

```

theory VEBT-Definitions imports
  Main
  HOL-Library.Extended-Nat
  HOL-Library.Code-Target-Numeral
  HOL-Library.Code-Target-Nat

```

```

begin

```

## 1 Preliminaries and Preparations

### 1.1 Data Type Definition

```

datatype VEBT = is-Node: Node (info:(nat*nat) option)(deg: nat)(treeList: VEBT list) (summary:VEBT)
|
  is-Leaf: Leaf bool bool

```

```

hide-const (open) info deg treeList summary

```

```

locale VEBT-internal begin

```

### 1.2 Functions for obtaining high and low bits of an input number.

```

definition high :: nat ⇒ nat ⇒ nat where
  high x n = (x div (2n))

```

```

definition low :: nat ⇒ nat ⇒ nat where
  low x n = (x mod (2n))

```

### 1.3 Some auxiliary lemmata

```

lemma inthall[termination-simp]: (∧ x. x ∈ set xs ⇒ P x) ⇒ n < length xs ⇒ P (xs ! n)
  ⟨proof⟩

```

```

lemma intind: i < n ⇒ P x ⇒ P (replicate n x ! i)
  ⟨proof⟩

```

```

lemma concat-inth:(xs @ [x] @ ys) ! (length xs) = x
  ⟨proof⟩

```

```

lemma pos-n-replace: n < length xs ⇒ length xs = length (take n xs @ [y] @ drop (Suc n) xs)
  ⟨proof⟩

```

```

lemma inthrepl: i < n ⇒ (replicate n x) ! i = x ⟨proof⟩

```

```

lemma nth-repl: m < length xs ⇒ n < length xs ⇒ m ≠ n ⇒ (take n xs @ [x] @ drop (n+1) xs) ! m
  = xs ! m
  ⟨proof⟩

```

**lemma** *[termination-simp]:assumes*  $high\ x\ deg < length\ treeList$   
**shows**  $size\ (treeList\ !\ high\ x\ deg) < Suc\ (size-list\ size\ treeList + size\ s)$   
*<proof>*

## 1.4 Auxiliary functions for defining valid Van Emde Boas Trees

This function checks whether an element occurs in a Leaf

**fun** *naive-member* ::  $VEBT \Rightarrow nat \Rightarrow bool$  **where**  
*naive-member* (Leaf  $a\ b$ )  $x = (if\ x = 0\ then\ a\ else\ if\ x = 1\ then\ b\ else\ False)$  |  
*naive-member* (Node  $-\ 0\ -\ -$ )  $- = False$  |  
*naive-member* (Node  $- deg\ treeList\ s$ )  $x = (let\ pos = high\ x\ (deg\ div\ 2)\ in$   
 $(if\ pos < length\ treeList\ then\ naive-member\ (treeList\ !\ pos)\ (low\ x\ (deg\ div\ 2))\ else\ False))$

Test for elements stored by using the provide min-max-fields

**fun** *membermima* ::  $VEBT \Rightarrow nat \Rightarrow bool$  **where**  
*membermima* (Leaf  $- -$ )  $- = False$  |  
*membermima* (Node  $None\ 0\ -\ -$ )  $- = False$  |  
*membermima* (Node (Some  $(mi, ma)$ )  $0\ -\ -$ )  $x = (x = mi \vee x = ma)$  |  
*membermima* (Node (Some  $(mi, ma)$ )  $deg\ treeList\ -$ )  $x = (x = mi \vee x = ma \vee$   
 $let\ pos = high\ x\ (deg\ div\ 2)\ in\ (if\ pos < length\ treeList$   
 $then\ membermima\ (treeList\ !\ pos)\ (low\ x\ (deg\ div\ 2))\ else\ False))$  |  
*membermima* (Node  $None\ (deg)\ treeList\ -$ )  $x = (let\ pos = high\ x\ (deg\ div\ 2)\ in$   
 $(if\ pos < length\ treeList\ then\ membermima\ (treeList\ !\ pos)\ (low\ x\ (deg\ div\ 2))\ else\ False))$

**lemma** *length-mul-elem*:  $(\forall\ x \in set\ xs.\ length\ x = n) \implies length\ (concat\ xs) = (length\ xs) * n$   
*<proof>*

We combine both auxiliary functions: The following test returns true if and only if an element occurs in the tree with respect to our interpretation no matter where it is stored.

**definition** *both-member-options* ::  $VEBT \Rightarrow nat \Rightarrow bool$  **where**  
*both-member-options*  $t\ x = (naive-member\ t\ x \vee membermima\ t\ x)$

**end**  
**context begin**  
**interpretation** *VEBT-internal* *<proof>*

**definition** *set-vebt* ::  $VEBT \Rightarrow nat\ set$  **where**  
*set-vebt*  $t = \{x.\ both-member-options\ t\ x\}$   
**end**

## 1.5 Inductive Definition of semantically valid Van Emde Boas Trees

Invariant for verification proofs

**context begin**  
**interpretation** *VEBT-internal* *<proof>*

**inductive** *invar-vebt*:  $VEBT \Rightarrow nat \Rightarrow bool$  **where**

```

invar-vebt (Leaf a b) (Suc 0) |
(∀ t ∈ set treeList. invar-vebt t n) ⇒ invar-vebt summary m ⇒ length treeList = 2m
⇒ m = n ⇒ deg = n + m ⇒ (∄ i. both-member-options summary i)
⇒ (∀ t ∈ set treeList. ∄ x. both-member-options t x)
⇒ invar-vebt (Node None deg treeList summary) deg |
(∀ t ∈ set treeList. invar-vebt t n) ⇒ invar-vebt summary m
⇒ length treeList = 2m ⇒ m = Suc n ⇒ deg = n + m ⇒ (∄ i. both-member-options summary
i)
⇒ (∀ t ∈ set treeList. ∄ x. both-member-options t x)
⇒ invar-vebt (Node None deg treeList summary) deg |
(∀ t ∈ set treeList. invar-vebt t n) ⇒ invar-vebt summary m ⇒ length treeList = 2m ⇒ m =
n
⇒ deg = n + m ⇒ (∀ i < 2m. (∃ x. both-member-options (treeList ! i) x) ↔ (both-member-options
summary i)) ⇒
    (mi = ma → (∀ t ∈ set treeList. ∄ x. both-member-options t x)) ⇒
      mi ≤ ma ⇒ ma < 2deg ⇒
        (mi ≠ ma →
          (∀ i < 2m.
            (high ma n = i → both-member-options (treeList ! i) (low ma n)) ∧
              (∀ x. (high x n = i ∧ both-member-options (treeList ! i) (low x n)
                ) → mi < x ∧ x ≤ ma) ) )
        ⇒ invar-vebt (Node (Some (mi, ma)) deg treeList summary) deg |
(∀ t ∈ set treeList. invar-vebt t n) ⇒ invar-vebt summary m ⇒ length treeList = 2m
⇒ m = Suc n ⇒ deg = n + m ⇒ (∀ i < 2m. (∃ x. both-member-options (treeList ! i) x) ↔ (
both-member-options summary i)) ⇒
    (mi = ma → (∀ t ∈ set treeList. ∄ x. both-member-options t x)) ⇒
      mi ≤ ma ⇒ ma < 2deg ⇒
        (mi ≠ ma →
          (∀ i < 2m.
            (high ma n = i → both-member-options (treeList ! i) (low ma n)) ∧
              (∀ x. (high x n = i ∧ both-member-options (treeList ! i) (low x n)
                ) → mi < x ∧ x ≤ ma)))
        ⇒ invar-vebt (Node (Some (mi, ma)) deg treeList summary) deg

```

**end**

**context** *VEBT-internal* **begin**

**definition** *in-children*  $n$   $treeList$   $x \equiv both\_member\_options (treeList ! high\ x\ n) (low\ x\ n)$

functional validness definition

```

fun valid' :: VEBT ⇒ nat ⇒ bool where
  valid' (Leaf - -) d ↔ d=1
| valid' (Node mima deg treeList summary) deg' ↔
  (
    deg=deg' ∧ (
      let n = deg div 2; m = deg - n in
        (∀ t ∈ set treeList. valid' t n)
      ∧ valid' summary m
    )
  )

```

```

    ∧ length treeList = 2^m
    ∧ (
      case mima of
      None ⇒ (∄ i. both-member-options summary i) ∧ (∀ t ∈ set treeList. ∄ x. both-member-options
t x)
      | Some (mi,ma) ⇒
        mi ≤ ma ∧ ma < 2^deg
        ∧ (∀ i < 2^m. (∃ x. both-member-options (treeList ! i) x) ↔ ( both-member-options summary
i))
        ∧ (if mi=ma then (∀ t ∈ set treeList. ∄ x. both-member-options t x)
        else
          in-children n treeList ma
          ∧ (∀ x < 2^deg. in-children n treeList x → mi < x ∧ x ≤ ma)
        )
      )
    )
  )
)
)
)

```

equivalence proofs

**lemma** *high-bound-aux*:  $ma < 2^{(n+m)} \implies \text{high } ma \ n < 2^m$   
 ⟨proof⟩

**lemma** *valid-eq1*:  
**assumes** *invar-vebt*  $t \ d$   
**shows** *valid'*  $t \ d$   
 ⟨proof⟩

**lemma** *even-odd-cases*:  
**fixes**  $x :: \text{nat}$   
**obtains**  $n$  **where**  $x = n + n \mid n$  **where**  $x = n + \text{Suc } n$   
 ⟨proof⟩

**lemma** *valid-eq2*: *valid'*  $t \ d \implies \text{invar-vebt } t \ d$   
 ⟨proof⟩

**lemma** *valid-eq*: *valid'*  $t \ d \longleftrightarrow \text{invar-vebt } t \ d$   
 ⟨proof⟩

**lemma** [*termination-simp*]: **assumes** *odd*  $(v :: \text{nat})$  **shows**  $v \ \text{div } 2 < v$   
 ⟨proof⟩

**lemma** [*termination-simp*]: **assumes**  $n > 1$  **and** *odd*  $n$  **shows**  $\text{Suc } (n \ \text{div } 2) < n$   
 ⟨proof⟩

**end**



## 1.6 Function for generating an empty tree of arbitrary degree respectively order

**context begin**

**interpretation** *VEBT-internal*  $\langle proof \rangle$

**fun** *vebt-buildup* ::  $nat \Rightarrow VEBT$  **where**

*vebt-buildup* 0 = *Leaf False False* |

*vebt-buildup* (*Suc* 0) = *Leaf False False* |

*vebt-buildup* n = (if even n then (let half = n div 2 in

*Node None* n (*replicate* ( $2^{\widehat{half}}$ ) (*vebt-buildup* half)) (*vebt-buildup* half))

else (let half = n div 2 in

*Node None* n (*replicate* ( $2^{\widehat{Suc\ half}}$ ) (*vebt-buildup* half)) (*vebt-buildup* (*Suc* half))))

**end**

**context** *VEBT-internal* **begin**

**lemma** *buildup-nothing-in-leaf*:  $\neg naive-member$  (*vebt-buildup* n) x  
 $\langle proof \rangle$

**lemma** *buildup-nothing-in-min-max*:  $\neg membermima$  (*vebt-buildup* n) x  
 $\langle proof \rangle$

The empty tree generated by *vebt<sub>o</sub>ildup* is indeed a valid tree.

**lemma** *buildup-gives-valid*:  $n > 0 \implies invar-vebt$  (*vebt-buildup* n) n  
 $\langle proof \rangle$

**lemma** *mi-ma-2-deg*: **assumes** *invar-vebt* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) n **shows**  $mi \leq ma \wedge ma < 2^{\widehat{deg}}$   
 $\langle proof \rangle$

**lemma** *deg-not-0*: *invar-vebt* t n  $\implies n > 0$   
 $\langle proof \rangle$

**lemma** *set-n-deg-not-0*: **assumes**  $\forall t \in set\ treeList. invar-vebt\ t$  **and** *length treeList* =  $2^{\widehat{m}}$  **shows**  $n \geq 1$   
 $\langle proof \rangle$

**lemma** *both-member-options-ding*: **assumes** *invar-vebt* (*Node info deg treeList summary*) n **and**  $x < 2^{\widehat{deg}}$  **and** *both-member-options* (*treeList* ! (*high* x (*deg* div 2))) (*low* x (*deg* div 2)) **shows** *both-member-options* (*Node info deg treeList summary*) x  
 $\langle proof \rangle$

**lemma** *exp-split-high-low*: **assumes**  $x < 2^{\widehat{(n+m)}}$  **and**  $n > 0$  **and**  $m > 0$  **shows** *high* x n  $< 2^{\widehat{m}}$  **and** *low* x n  $< 2^{\widehat{n}}$   
 $\langle proof \rangle$

**lemma** *low-inv*: **assumes**  $x < 2^{\widehat{n}}$  **shows** *low* ( $y * 2^{\widehat{n}} + x$ ) n = x  $\langle proof \rangle$

**lemma** *high-inv*: **assumes**  $x < 2^{\wedge}n$  **shows**  $high (y * 2^{\wedge}n + x) n = y$  *<proof>*

**lemma** *both-member-options-from-child-to-complete-tree*:

**assumes**  $high x (deg \text{ div } 2) < length \text{ treeList}$  **and**  $deg \geq 1$  **and** *both-member-options* ( $treeList ! (high x (deg \text{ div } 2))$ ) ( $low x (deg \text{ div } 2)$ )

**shows** *both-member-options* ( $Node (Some (mi, ma)) deg \text{ treeList summary}$ )  $x$   
*<proof>*

**lemma** *both-member-options-from-complete-tree-to-child*:

**assumes**  $deg \geq 1$  **and** *both-member-options* ( $Node (Some (mi, ma)) deg \text{ treeList summary}$ )  $x$

**shows** *both-member-options* ( $treeList ! (high x (deg \text{ div } 2))$ ) ( $low x (deg \text{ div } 2) \vee x = mi \vee x = ma$ )

*<proof>*

**lemma** *pow-sum*: ( $divide :: nat \Rightarrow nat \Rightarrow nat$ ) ( $((2 :: nat) \wedge ((a :: nat) + (b :: nat))) (2^{\wedge}a) = 2^{\wedge}b$ )

*<proof>*

**fun** *elim-dead* ::  $VEBT \Rightarrow enat \Rightarrow VEBT$  **where**

*elim-dead* ( $Leaf a b$ ) - =  $Leaf a b$  |

*elim-dead* ( $Node \text{ info } deg \text{ treeList summary}$ )  $\infty$  =  
 $(Node \text{ info } deg (map (\lambda t. \text{elim-dead } t (enat (2^{\wedge}(deg \text{ div } 2)))) \text{ treeList}$   
 $(\text{elim-dead summary } \infty))$  |

*elim-dead* ( $Node \text{ info } deg \text{ treeList summary}$ ) ( $enat l$ ) =  
 $(Node \text{ info } deg (take (l \text{ div } (2^{\wedge}(deg \text{ div } 2))) (map (\lambda t. \text{elim-dead } t (enat (2^{\wedge}(deg \text{ div } 2)))) \text{ treeList}))$   
 $(\text{elim-dead summary } ((enat (l \text{ div } (2^{\wedge}(deg \text{ div } 2))))))$ )

**lemma** *elimnum*:  $invar\text{-vebt } (Node \text{ info } deg \text{ treeList summary}) n \Longrightarrow$

$\text{elim-dead } (Node \text{ info } deg \text{ treeList summary}) (enat ((2 :: nat) \wedge n)) = (Node \text{ info } deg \text{ treeList summary})$   
*<proof>*

**lemma** *elimcomplete*:  $invar\text{-vebt } (Node \text{ info } deg \text{ treeList summary}) n \Longrightarrow$

$\text{elim-dead } (Node \text{ info } deg \text{ treeList summary}) \infty = (Node \text{ info } deg \text{ treeList summary})$   
*<proof>*

**end**

**end**

**theory** *VEBT-Member* **imports** *VEBT-Definitions*

**begin**

## 2 Member Function

**context** **begin**

**interpretation** *VEBT-internal* *<proof>*

**fun** *vebt-member* ::  $VEBT \Rightarrow nat \Rightarrow bool$  **where**

*vebt-member* ( $Leaf a b$ )  $x = (if x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } False)$  |

*vebt-member* ( $Node \text{ None } - -$ )  $x = False$  |

```

vebt-member (Node - 0 - -) x = False|
vebt-member (Node - (Suc 0) - -) x = False|
vebt-member (Node (Some (mi, ma)) deg treeList summary) x = (
  if x = mi then True else
  if x = ma then True else
  if x < mi then False else
  if x > ma then False else (
    let h = high x (deg div 2);
        l = low x (deg div 2) in(
      if h < length treeList
      then vebt-member (treeList ! h) l
      else False)))

```

**end**

**context** *VEBT-internal* **begin**

**lemma** *member-inv*:

**assumes** *vebt-member* (Node (Some (mi, ma)) deg treeList summary) x

**shows**  $\text{deg} \geq 2 \wedge$

$(x = mi \vee x = ma \vee (x < ma \wedge x > mi \wedge \text{high } x (\text{deg div } 2) < \text{length } \text{treeList} \wedge$   
 $\text{vebt-member } (\text{treeList} ! (\text{high } x (\text{deg div } 2))) (\text{low } x (\text{deg div } 2))))$

*<proof>*

**definition** *bit-concat*:: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **where**

*bit-concat* h l d =  $h * 2^d + l$

**lemma** *bit-split-inv*: *bit-concat* (high x d) (low x d) d = x

*<proof>*

**definition** *set-vebt'*::*VEBT*  $\Rightarrow$  *nat set* **where**

*set-vebt'* t = {x. *vebt-member* t x}

**lemma** *Leaf-0-not*: **assumes** *invar-vebt* (Leaf a b) 0 **shows** False

*<proof>*

**lemma** *valid-0-not*: *invar-vebt* t 0  $\implies$  False

*<proof>*

**theorem** *valid-tree-deg-neq-0*:  $(\neg \text{invar-vebt } t \ 0)$

*<proof>*

**lemma** *deg-1-Leafy*: *invar-vebt* t n  $\implies$  n = 1  $\implies$   $\exists$  a b. t = Leaf a b

*<proof>*

**lemma** *deg-1-Leaf*: *invar-vebt* t 1  $\implies$   $\exists$  a b. t = Leaf a b

*<proof>*

**corollary** *deg1Leaf*:  $\text{invar-vebt } t \ 1 \longleftrightarrow (\exists a \ b. t = \text{Leaf } a \ b)$   
(proof)

**lemma** *deg-SUcn-Node*: **assumes** *invar-vebt tree (Suc (Suc n))* **shows**  
 $\exists \text{ info treeList } s. \text{tree} = \text{Node info (Suc (Suc n)) treeList } s$   
(proof)

**lemma** *invar-vebt (Node info deg treeList summary) deg*  $\implies \text{deg} > 1$   
(proof)

**lemma** *deg-deg-n*: **assumes** *invar-vebt (Node info deg treeList summary) n* **shows**  $\text{deg} = n$   
(proof)

**lemma** *member-valid-both-member-options*:  
 $\text{invar-vebt tree } n \implies \text{vebt-member tree } x \implies (\text{naive-member tree } x \vee \text{membermima tree } x)$   
(proof)

**lemma** *member-bound*:  $\text{vebt-member tree } x \implies \text{invar-vebt tree } n \implies x < 2^{\wedge}n$   
(proof)

**theorem** *inrange*: **assumes** *invar-vebt t n* **shows**  $\text{set-vebt}' t \subseteq \{0..2^{\wedge}n-1\}$   
(proof)

**theorem** *buildup-gives-empty*:  $\text{set-vebt}' (\text{vebt-buildup } n) = \{\}$   
(proof)

**fun** *minNull*::*VEBT*  $\Rightarrow$  *bool* **where**  
*minNull (Leaf False False) = True*  
*minNull (Leaf - -) = False*  
*minNull (Node None - - -) = True*  
*minNull (Node (Some -) - - -) = False*

**lemma** *min-Null-member*:  $\text{minNull } t \implies \neg \text{vebt-member } t \ x$   
(proof)

**lemma** *not-min-Null-member*:  $\neg \text{minNull } t \implies \exists y. \text{both-member-options } t \ y$   
(proof)

**lemma** *valid-member-both-member-options*:  $\text{invar-vebt } t \ n \implies \text{both-member-options } t \ x \implies \text{vebt-member } t \ x$   
(proof)

**corollary** *both-member-options-equiv-member*: **assumes** *invar-vebt t n*  
**shows**  $\text{both-member-options } t \ x \longleftrightarrow \text{vebt-member } t \ x$   
(proof)

**lemma** *member-correct*:  $\text{invar-vebt } t \ n \implies \text{vebt-member } t \ x \longleftrightarrow x \in \text{set-vebt } t$   
(proof)

**corollary** *set-vebt-set-vebt'-valid*: **assumes** *invar-vebt* *t n* **shows** *set-vebt* *t = set-vebt' t*  
 ⟨*proof*⟩

**lemma** *set-vebt-finite*: *invar-vebt* *t n*  $\implies$  *finite* (*set-vebt' t*)  
 ⟨*proof*⟩

**lemma** *mi-eq-ma-no-ch*: **assumes** *invar-vebt* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *deg* **and**  
*mi = ma*  
**shows**  $(\forall t \in \text{set } \text{treeList}. \nexists x. \text{both-member-options } t x) \wedge (\nexists x. \text{both-member-options } \text{summary } x)$   
 ⟨*proof*⟩

**end**  
**end**

**theory** *VEBT-Insert* **imports** *VEBT-Member*  
**begin**

### 3 Insert Function

**context** **begin**  
**interpretation** *VEBT-internal* ⟨*proof*⟩

**fun** *vebt-insert* :: *VEBT*  $\Rightarrow$  *nat*  $\Rightarrow$  *VEBT* **where**  
*vebt-insert* (*Leaf* *a b*) *x* = (*if* *x=0* *then Leaf True b* *else if* *x = 1* *then Leaf a True* *else Leaf a b*)|  
*vebt-insert* (*Node* *info 0 ts s*) *x* = (*Node* *info 0 ts s*)|  
*vebt-insert* (*Node* *info (Suc 0) ts s*) *x* = (*Node* *info (Suc 0) ts s*)|  
*vebt-insert* (*Node* *None (Suc deg) treeList summary*) *x* =  
 (*Node* (*Some* (*x,x*)) (*Suc deg*) *treeList summary*)|  
*vebt-insert* (*Node* (*Some* (*mi,ma*)) *deg treeList summary*) *x* = (  
*let* *xn* = (*if* *x < mi* *then mi* *else x*);  
*minn* = (*if* *x < mi* *then x* *else mi*);  
*l* = *low xn (deg div 2)*;  
*h* = *high xn (deg div 2)* *in* (  
*if* *h < length treeList*  $\wedge \neg (x = mi \vee x = ma)$  *then*  
   *Node* (*Some* (*minn*, *max xn ma*)) *deg* (*treeList*[*h:= vebt-insert* (*treeList ! h*) *l*])  
   (*if* *minNull* (*treeList ! h*) *then vebt-insert summary h* *else summary*)  
*else* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*))

**end**

**context** *VEBT-internal* **begin**

**lemma** *insert-simp-norm*:  
**assumes** *high x (deg div 2) < length treeList* **and** (*mi::nat*)  $< x$  **and** *deg*  $\geq 2$  **and** *x*  $\neq ma$   
**shows** *vebt-insert* (*Node* (*Some* (*mi,ma*)) *deg treeList summary*) *x* =  
 (*Node* (*Some* (*mi*, *max x ma*)) *deg* (*treeList* [(*high x (deg div 2)*):= *vebt-insert* (*treeList !*  
 (*high x (deg div 2)*)) (*low x (deg div 2)*)]))  
 (*if* *minNull* (*treeList ! (high x (deg div 2))*) *then vebt-insert summary* (*high x (deg*  
*div 2)*) *else summary*)

*<proof>*

**lemma** *insert-simp-excp*:

**assumes**  $high\ mi\ (deg\ div\ 2) < length\ treeList$  **and**  $(x::nat) < mi$  **and**  $deg \geq 2$  **and**  $x \neq ma$   
**shows**  $vebt-insert\ (Node\ (Some\ (mi,ma))\ deg\ treeList\ summary)\ x =$   
 $Node\ (Some\ (x,\ max\ mi\ ma))\ deg\ (treeList[(high\ mi\ (deg\ div\ 2)) := vebt-insert\ (treeList$   
 $!\ (high\ mi\ (deg\ div\ 2))\ (low\ mi\ (deg\ div\ 2))])$   
 $(if\ minNull\ (treeList\ !\ (high\ mi\ (deg\ div\ 2)))\ then\ vebt-insert\ summary\ (high\ mi\ (deg$   
 $div\ 2))\ else\ summary)$   
*<proof>*

**lemma** *insert-simp-mima*: **assumes**  $x = mi \vee x = ma$  **and**  $deg \geq 2$

**shows**  $vebt-insert\ (Node\ (Some\ (mi,ma))\ deg\ treeList\ summary)\ x = (Node\ (Some\ (mi,ma))\ deg$   
 $treeList\ summary)$   
*<proof>*

**lemma** *valid-insert-both-member-options-add*:  $invar-vebt\ t\ n \implies x < 2^{\widehat{n}} \implies both-member-options$   
 $(vebt-insert\ t\ x)\ x$   
*<proof>*

**lemma** *valid-insert-both-member-options-pres*:  $invar-vebt\ t\ n \implies x < 2^{\widehat{n}} \implies y < 2^{\widehat{n}} \implies both-member-options$   
 $t\ x$   
 $\implies both-member-options\ (vebt-insert\ t\ y)\ x$   
*<proof>*

**lemma** *post-member-pre-member*:  $invar-vebt\ t\ n \implies x < 2^{\widehat{n}} \implies y < 2^{\widehat{n}} \implies vebt-member\ (vebt-insert$   
 $t\ x)\ y \implies vebt-member\ t\ y \vee x = y$   
*<proof>*

**end**  
**end**

**theory** *VEBT-InsertCorrectness* **imports** *VEBT-Member* *VEBT-Insert*  
**begin**

**context** *VEBT-internal* **begin**

## 4 Correctness of the Insert Operation

### 4.1 Validness Preservation

**theorem** *valid-pres-insert*:  $invar-vebt\ t\ n \implies x < 2^{\widehat{n}} \implies invar-vebt\ (vebt-insert\ t\ x)\ n$   
*<proof>*

### 4.2 Correctness with Respect to Set Interpretation

**theorem** *insert-corr*:

**assumes** *invar-vebt* *t n* **and**  $x < 2^{\wedge}n$   
**shows**  $set-vebt' t \cup \{x\} = set-vebt' (vebt-insert t x)$   
 <proof>

**corollary** *insert-correct*: **assumes** *invar-vebt* *t n* **and**  $x < 2^{\wedge}n$  **shows**  
 $set-vebt t \cup \{x\} = set-vebt (vebt-insert t x)$   
 <proof>

**fun** *insert'*::*VEBT*  $\Rightarrow$  *nat*  $\Rightarrow$  *VEBT* **where**  
*insert'* (*Leaf a b*) *x* = *vebt-insert* (*Leaf a b*) *x* |  
*insert'* (*Node info deg treeList summary*) *x* =  
 (if  $x \geq 2^{\wedge}deg$  then (*Node info deg treeList summary*)  
   else *vebt-insert* (*Node info deg treeList summary*) *x*)

**theorem** *insert'-pres-valid*: **assumes** *invar-vebt* *t n* **shows** *invar-vebt* (*insert' t x*) *n*  
 <proof>

**theorem** *insert'-correct*: **assumes** *invar-vebt* *t n*  
**shows**  $set-vebt (insert' t x) = (set-vebt t \cup \{x\}) \cap \{0..2^{\wedge}n-1\}$   
 <proof>

**end**  
**end**

**theory** *VEBT-MinMax* **imports** *VEBT-Member*  
**begin**

## 5 The Minimum and Maximum Operation

**fun** *vebt-mint* :: *VEBT*  $\Rightarrow$  *nat option* **where**  
*vebt-mint* (*Leaf a b*) = (if *a* then *Some 0* else if *b* then *Some 1* else *None*) |  
*vebt-mint* (*Node None - - -*) = *None* |  
*vebt-mint* (*Node (Some (mi,ma)) - - -*) = *Some mi*

**fun** *vebt-maxt* :: *VEBT*  $\Rightarrow$  *nat option* **where**  
*vebt-maxt* (*Leaf a b*) = (if *b* then *Some 1* else if *a* then *Some 0* else *None*) |  
*vebt-maxt* (*Node None - - -*) = *None* |  
*vebt-maxt* (*Node (Some (mi,ma)) - - -*) = *Some ma*

**context** *VEBT-internal* **begin**

**fun** *option-shift*::(*'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a*)  $\Rightarrow$  *'a option*  $\Rightarrow$  *'a option*  $\Rightarrow$  *'a option* **where**  
*option-shift* - *None* - = *None* |  
*option-shift* - - *None* = *None* |  
*option-shift* *f* (*Some a*) (*Some b*) = *Some (f a b)*

**definition** *power*::*nat option*  $\Rightarrow$  *nat option*  $\Rightarrow$  *nat option* (**infixl**  $\hat{\ }_o$  81) **where**

$power = option-shift (\hat{\ })$

**definition**  $add :: nat\ option \Rightarrow nat\ option \Rightarrow nat\ option$  (**infixl** $_{+o}$  79) **where**  
 $add = option-shift (+)$

**definition**  $mul :: nat\ option \Rightarrow nat\ option \Rightarrow nat\ option$  (**infixl** $_{*o}$  80) **where**  
 $mul = option-shift (*)$

**fun**  $option-comp-shift :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ option \Rightarrow 'a\ option \Rightarrow bool$  **where**  
 $option-comp-shift - None - = False$   
 $option-comp-shift - - None = False$   
 $option-comp-shift f (Some x) (Some y) = f x y$

**fun**  $less :: nat\ option \Rightarrow nat\ option \Rightarrow bool$  (**infixl** $_{<o}$  80) **where**  
 $less x y = option-comp-shift (<) x y$

**fun**  $lesseq :: nat\ option \Rightarrow nat\ option \Rightarrow bool$  (**infixl** $_{\leq o}$  80) **where**  
 $lesseq x y = option-comp-shift (\leq) x y$

**fun**  $greater :: nat\ option \Rightarrow nat\ option \Rightarrow bool$  (**infixl** $_{>o}$  80) **where**  
 $greater x y = option-comp-shift (>) x y$

**lemma**  $add-shift : x + y = z \longleftrightarrow Some\ x +_o\ Some\ y = Some\ z$   
 $\langle proof \rangle$

**lemma**  $mul-shift : x * y = z \longleftrightarrow Some\ x *_o\ Some\ y = Some\ z \langle proof \rangle$

**lemma**  $power-shift : x \hat{\ } y = z \longleftrightarrow Some\ x \hat{\ }_o\ Some\ y = Some\ z \langle proof \rangle$

**lemma**  $less-shift : x < y \longleftrightarrow Some\ x <_o\ Some\ y \langle proof \rangle$

**lemma**  $lesseq-shift : x \leq y \longleftrightarrow Some\ x \leq_o\ Some\ y \langle proof \rangle$

**lemma**  $greater-shift : x > y \longleftrightarrow Some\ x >_o\ Some\ y \langle proof \rangle$

**definition**  $max-in-set :: nat\ set \Rightarrow nat \Rightarrow bool$  **where**  
 $max-in-set\ xs\ x \longleftrightarrow (x \in xs \wedge (\forall y \in xs. y \leq x))$

**lemma**  $maxt-member : invar-vebt\ t\ n \Longrightarrow vebt-maxt\ t = Some\ maxi \Longrightarrow vebt-member\ t\ maxi$   
 $\langle proof \rangle$

**lemma**  $maxt-corr-help : invar-vebt\ t\ n \Longrightarrow vebt-maxt\ t = Some\ maxi \Longrightarrow vebt-member\ t\ x \Longrightarrow maxi$   
 $\geq x$   
 $\langle proof \rangle$

**lemma**  $maxt-corr-help-empty : invar-vebt\ t\ n \Longrightarrow vebt-maxt\ t = None \Longrightarrow set-vebt'\ t = \{\}$   
 $\langle proof \rangle$



**theorem** *maxt-corr:assumes* *invar-vebt t n* **and** *vebt-maxt t = Some x* **shows** *max-in-set (set-vebt' t) x*  
 ⟨proof⟩

**theorem** *maxt-sound:assumes* *invar-vebt t n* **and** *max-in-set (set-vebt' t) x* **shows** *vebt-maxt t = Some x*  
 ⟨proof⟩

**definition** *min-in-set :: nat set ⇒ nat ⇒ bool* **where**  
*min-in-set xs x*  $\longleftrightarrow (x \in xs \wedge (\forall y \in xs. y \geq x))$

**lemma** *mint-member: invar-vebt t n ⇒ vebt-mint t = Some maxi ⇒ vebt-member t maxi*  
 ⟨proof⟩

**lemma** *mint-corr-help: invar-vebt t n ⇒ vebt-mint t = Some mini ⇒ vebt-member t x ⇒ mini ≤ x*  
 ⟨proof⟩

**lemma** *mint-corr-help-empty: invar-vebt t n ⇒ vebt-mint t = None ⇒ set-vebt' t = {}*  
 ⟨proof⟩

**theorem** *mint-corr:assumes* *invar-vebt t n* **and** *vebt-mint t = Some x* **shows** *min-in-set (set-vebt' t) x*  
 ⟨proof⟩

**theorem** *mint-sound:assumes* *invar-vebt t n* **and** *min-in-set (set-vebt' t) x* **shows** *vebt-mint t = Some x*  
 ⟨proof⟩

**lemma** *summaxma:assumes* *invar-vebt (Node (Some (mi, ma)) deg treeList summary) deg* **and** *mi ≠ ma*  
**shows** *the (vebt-maxt summary) = high ma (deg div 2)*  
 ⟨proof⟩

**lemma** *maxbmo: vebt-maxt t = Some x ⇒ both-member-options t x*  
 ⟨proof⟩

**lemma** *misiz: invar-vebt t n ⇒ Some m = vebt-mint t ⇒ m < 2<sup>n</sup>*  
 ⟨proof⟩

**lemma** *mintlistlength: assumes* *invar-vebt (Node (Some (mi, ma)) deg treeList summary) n*  
*mi ≠ ma* **shows** *ma > mi ∧ (∃ m. Some m = vebt-mint summary ∧ m < 2<sup>n - n div 2</sup>)*  
 ⟨proof⟩

**lemma** *power-minus-is-div:*  
*b ≤ a ⇒ (2 :: nat) <sup>a - b</sup> = 2<sup>a</sup> div 2<sup>b</sup>*

*<proof>*

**lemma** *nested-mint:assumes* *invar-vebt* (Node (Some (mi, ma)) deg treeList summary) n n = Suc (Suc va)

$\neg$  ma < mi ma  $\neq$  mi **shows**

high (the (vebt-mint summary) \* (2 \* 2 ^ (va div 2)) + the (vebt-mint (treeList ! the (vebt-mint summary)))) (Suc (va div 2))

< length treeList

*<proof>*

**lemma** *minminNull*: vebt-mint t = None  $\implies$  minNull t

*<proof>*

**lemma** *minNullmin*: minNull t  $\implies$  vebt-mint t = None

*<proof>*

**end**

**end**

**theory** *VEBT-Succ imports* *VEBT-Insert* *VEBT-MinMax*

**begin**

## 6 The Successor Operation

**definition** *is-succ-in-set* :: nat set  $\implies$  nat  $\implies$  nat  $\implies$  bool **where**

*is-succ-in-set* xs x y = (y  $\in$  xs  $\wedge$  y > x  $\wedge$  ( $\forall$  z  $\in$  xs. (z > x  $\longrightarrow$  z  $\geq$  y)))

**context** *VEBT-internal* **begin**

**corollary** *succ-member*: *is-succ-in-set* (set-vebt' t) x y = (vebt-member t y  $\wedge$  y > x  $\wedge$  ( $\forall$  z. vebt-member t z  $\wedge$  z > x  $\longrightarrow$  z  $\geq$  y))

*<proof>*

### 6.1 Auxiliary Lemmas on Sets and Successorship

**lemma** *finite* (A:: nat set)  $\implies$  A  $\neq$  {}  $\implies$  Min A  $\in$  A

*<proof>*

**lemma** *obtain-set-succ*: **assumes** (x::nat) < z **and** *max-in-set* A z **and** *finite* B **and** A=B **shows**  $\exists$  y. *is-succ-in-set* A x y

*<proof>*

**lemma** *succ-none-empty*: **assumes** ( $\nexists$  x. *is-succ-in-set* (xs) a x) **and** *finite* xs **shows**  $\neg$  ( $\exists$  x  $\in$  xs. *ord-class.greater* x a)

*<proof>*

**end**

## 6.2 The actual Function

context begin

interpretation *VEBT-internal* ⟨proof⟩

fun *vebt-succ* :: *VEBT* ⇒ *nat* ⇒ *nat option* where

*vebt-succ* (*Leaf* - *b*) 0 = (if *b* then *Some* 1 else *None*)|

*vebt-succ* (*Leaf* - -) (*Suc* *n*) = *None*|

*vebt-succ* (*Node* *None* - - -) - = *None*|

*vebt-succ* (*Node* - 0 - -) - = *None*|

*vebt-succ* (*Node* - (*Suc* 0) - -) - = *None*|

*vebt-succ* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (  
 if *x* < *mi* then (*Some* *mi*)  
 else (let *l* = *low* *x* (*deg div* 2); *h* = *high* *x* (*deg div* 2) in  
 if *h* < *length treeList* then  
 let *maxlow* = *vebt-maxt* (*treeList* ! *h*) in (  
 if *maxlow* ≠ *None* ∧ (*Some* *l* <<sub>o</sub> *maxlow*) then  
*Some* (2<sup>(*deg div* 2)</sup>) \*<sub>o</sub> *Some* *h* +<sub>o</sub> *vebt-succ* (*treeList* ! *h*) *l*  
 else let *sc* = *vebt-succ summary h* in  
 if *sc* = *None* then *None*  
 else *Some* (2<sup>(*deg div* 2)</sup>) \*<sub>o</sub> *sc* +<sub>o</sub> *vebt-mint* (*treeList* ! *the* *sc*) )  
 else *None*))

end

## 6.3 Lemmas for Term Decomposition

context *VEBT-internal* begin

lemma *succ-min*: assumes *deg* ≥ 2 and (*x::nat*) < *mi* shows

*vebt-succ* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = *Some* *mi*  
 ⟨proof⟩

lemma *succ-greatereq-min*: assumes *deg* ≥ 2 and (*x::nat*) ≥ *mi* shows

*vebt-succ* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (let *l* = *low* *x* (*deg div* 2); *h* = *high* *x*  
 (*deg div* 2) in

if *h* < *length treeList* then

let *maxlow* = *vebt-maxt* (*treeList* ! *h*) in

(if *maxlow* ≠ *None* ∧ (*Some* *l* <<sub>o</sub> *maxlow*) then

*Some* (2<sup>(*deg div* 2)</sup>) \*<sub>o</sub> *Some* *h* +<sub>o</sub> *vebt-succ* (*treeList* ! *h*) *l*

else let *sc* = *vebt-succ summary h* in

if *sc* = *None* then *None*

else *Some* (2<sup>(*deg div* 2)</sup>) \*<sub>o</sub> *sc* +<sub>o</sub> *vebt-mint* (*treeList* ! *the* *sc*) )

else *None*)

⟨proof⟩

lemma *succ-list-to-short*: assumes *deg* ≥ 2 and *x* ≥ *mi* and *high* *x* (*deg div* 2) ≥ *length treeList*  
 shows

*vebt-succ* (Node (Some (mi, ma)) deg treeList summary) x = None  
 ⟨proof⟩

**lemma succ-less-length-list:** assumes  $\text{deg} \geq 2$  and  $x \geq \text{mi}$  and  $\text{high } x (\text{deg div } 2) < \text{length treeList}$   
 shows

*vebt-succ* (Node (Some (mi, ma)) deg treeList summary) x =  
 (let l = low x (deg div 2); h = high x (deg div 2) ; maxlow = *vebt-maxt* (treeList ! h) in  
 (if maxlow  $\neq$  None  $\wedge$  (Some l  $<_o$  maxlow) then  
     Some ( $2^{\wedge}(\text{deg div } 2)$ ) \*<sub>o</sub> Some h +<sub>o</sub> *vebt-succ* (treeList ! h) l  
 else let sc = *vebt-succ summary* h in  
     if sc = None then None  
     else Some ( $2^{\wedge}(\text{deg div } 2)$ ) \*<sub>o</sub> sc +<sub>o</sub> *vebt-mint* (treeList !the sc)))

⟨proof⟩

## 6.4 Correctness Proof

**theorem succ-corr:** *invar-vebt* t n  $\implies$  *vebt-succ* t x = Some sx == *is-succ-in-set* (set-vebt' t) x sx  
 ⟨proof⟩

**corollary succ-empty:** assumes *invar-vebt* t n

shows (*vebt-succ* t x = None) = ({y. *vebt-member* t y  $\wedge$  y > x} = {})  
 ⟨proof⟩

**theorem succ-correct:** *invar-vebt* t n  $\implies$  *vebt-succ* t x = Some sx  $\longleftrightarrow$  *is-succ-in-set* (set-vebt t) x sx  
 ⟨proof⟩

**lemma is-succ-in-set** S x y  $\longleftrightarrow$  *min-in-set* {s . s  $\in$  S  $\wedge$  s > x} y  
 ⟨proof⟩

**lemma helpyd:** *invar-vebt* t n  $\implies$  *vebt-succ* t x = Some y  $\implies$  y <  $2^{\wedge}n$   
 ⟨proof⟩

**lemma geqmaxNone:**

assumes *invar-vebt* (Node (Some (mi, ma)) deg treeList summary) n x  $\geq$  ma  
 shows *vebt-succ* (Node (Some (mi, ma)) deg treeList summary) x = None  
 ⟨proof⟩

end  
 end

**theory** VEBT-Pred **imports** VEBT-MinMax VEBT-Insert  
**begin**

## 7 The Predecessor Operation

**definition** *is-pred-in-set* :: nat set  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool **where**  
*is-pred-in-set* xs x y = (y  $\in$  xs  $\wedge$  y < x  $\wedge$  ( $\forall$  z  $\in$  xs. (z < x  $\longrightarrow$  z  $\leq$  y)))

**context** VEBT-internal **begin**

## 7.1 Lemmas on Sets and Predecessorship

**corollary** *pred-member*:  $is\text{-}pred\text{-}in\text{-}set (set\text{-}vebt' t) x y = (vebt\text{-}member t y \wedge y < x \wedge (\forall z. vebt\text{-}member t z \wedge z < x \longrightarrow z \leq y))$

*<proof>*

**lemma** *finite* ( $A :: nat\ set$ )  $\implies A \neq \{\} \implies Max A \in A$

*<proof>*

**lemma** *obtain-set-pred*: **assumes** ( $x :: nat$ )  $> z$  **and** *min-in-set*  $A z$  **and** *finite*  $A$  **shows**  $\exists y. is\text{-}pred\text{-}in\text{-}set A x y$

*<proof>*

**lemma** *pred-none-empty*: **assumes** ( $\nexists x. is\text{-}pred\text{-}in\text{-}set (xs) a x$ ) **and** *finite*  $xs$  **shows**  $\neg (\exists x \in xs. ord\text{-}class.less x a)$

*<proof>*

**end**

## 7.2 The actual Function for Predecessor Search

**context** *begin*

**interpretation** *VEBT-internal* *<proof>*

**fun** *vebt-pred* ::  $VEBT \Rightarrow nat \Rightarrow nat\ option$  **where**

*vebt-pred* (*Leaf* - -) 0 = *None* |

*vebt-pred* (*Leaf* a -) (*Suc* 0) = (*if* a *then* *Some* 0 *else* *None*) |

*vebt-pred* (*Leaf* a b) - = (*if* b *then* *Some* 1 *else if* a *then* *Some* 0 *else* *None*) |

*vebt-pred* (*Node* *None* - - -) - = *None* |

*vebt-pred* (*Node* - 0 - -) - = *None* |

*vebt-pred* (*Node* - (*Suc* 0) - -) - = *None* |

*vebt-pred* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*)  $x =$  (

*if*  $x > ma$  *then* *Some* *ma*

*else* (*let*  $l = low\ x\ (deg\ div\ 2)$ ;  $h = high\ x\ (deg\ div\ 2)$  *in*

*if*  $h < length\ treeList$  *then*

*let*  $minlow = vebt\text{-}mint\ (treeList\ !\ h)$  *in* (

*if*  $minlow \neq None \wedge (Some\ l >_o\ minlow)$  *then*

*Some* ( $2^{\wedge}(deg\ div\ 2) *_o\ Some\ h +_o\ vebt\text{-}pred\ (treeList\ !\ h)\ l$ )

*else let*  $pr = vebt\text{-}pred\ summary\ h$  *in*

*if*  $pr = None$  *then* (

*if*  $x > mi$  *then* *Some* *mi*

*else* *None*)

*else* *Some* ( $2^{\wedge}(deg\ div\ 2) *_o\ pr +_o\ vebt\text{-}maxt\ (treeList\ !\ the\ pr)$ )

*else* *None*))

**end**

**context** *VEBT-internal* **begin**

### 7.3 Auxiliary Lemmas

**lemma** *pred-max*:

**assumes**  $\text{deg} \geq 2$  **and**  $(x::\text{nat}) > \text{ma}$

**shows**  $\text{vebt-pred} (\text{Node} (\text{Some} (\text{mi}, \text{ma})) \text{deg} \text{treeList} \text{summary}) x = \text{Some} \text{ma}$   
 ⟨proof⟩

**lemma** *pred-lesseq-max*:

**assumes**  $\text{deg} \geq 2$  **and**  $(x::\text{nat}) \leq \text{ma}$

**shows**  $\text{vebt-pred} (\text{Node} (\text{Some} (\text{mi}, \text{ma})) \text{deg} \text{treeList} \text{summary}) x = (\text{let } l = \text{low } x (\text{deg} \text{div } 2); h = \text{high } x (\text{deg} \text{div } 2) \text{ in}$   
     **if**  $h < \text{length } \text{treeList}$  **then**

$\text{let } \text{minlow} = \text{vebt-mint} (\text{treeList} ! h) \text{ in}$   
     **(if**  $\text{minlow} \neq \text{None} \wedge (\text{Some } l >_o \text{minlow})$  **then**  
          $\text{Some} (2^\wedge(\text{deg} \text{div } 2)) *_o \text{Some } h +_o \text{vebt-pred} (\text{treeList} ! h) l$   
     **else let**  $\text{pr} = \text{vebt-pred} \text{summary } h$  **in**  
     **if**  $\text{pr} = \text{None}$  **then (if**  $x > \text{mi}$  **then**  $\text{Some } \text{mi}$  **else**  $\text{None}$   
     **else**  $\text{Some} (2^\wedge(\text{deg} \text{div } 2)) *_o \text{pr} +_o \text{vebt-maxt} (\text{treeList} ! \text{the } \text{pr})$  **)**

**else**  $\text{None}$ )

⟨proof⟩

**lemma** *pred-list-to-short*:

**assumes**  $\text{deg} \geq 2$  **and**  $\text{ord-class.less-eq } x \text{ ma}$  **and**  $\text{high } x (\text{deg} \text{div } 2) \geq \text{length } \text{treeList}$

**shows**  $\text{vebt-pred} (\text{Node} (\text{Some} (\text{mi}, \text{ma})) \text{deg} \text{treeList} \text{summary}) x = \text{None}$   
 ⟨proof⟩

**lemma** *pred-less-length-list*:

**assumes**  $\text{deg} \geq 2$  **and**  $\text{ord-class.less-eq } x \text{ ma}$  **and**  $\text{high } x (\text{deg} \text{div } 2) < \text{length } \text{treeList}$

**shows**

$\text{vebt-pred} (\text{Node} (\text{Some} (\text{mi}, \text{ma})) \text{deg} \text{treeList} \text{summary}) x = (\text{let } l = \text{low } x (\text{deg} \text{div } 2); h = \text{high } x (\text{deg} \text{div } 2); \text{minlow} = \text{vebt-mint} (\text{treeList} ! h) \text{ in}$

**(if**  $\text{minlow} \neq \text{None} \wedge (\text{Some } l >_o \text{minlow})$  **then**  
          $\text{Some} (2^\wedge(\text{deg} \text{div } 2)) *_o \text{Some } h +_o \text{vebt-pred} (\text{treeList} ! h) l$   
     **else let**  $\text{pr} = \text{vebt-pred} \text{summary } h$  **in**  
     **if**  $\text{pr} = \text{None}$  **then (if**  $x > \text{mi}$  **then**  $\text{Some } \text{mi}$  **else**  $\text{None}$   
     **else**  $\text{Some} (2^\wedge(\text{deg} \text{div } 2)) *_o \text{pr} +_o \text{vebt-maxt} (\text{treeList} ! \text{the } \text{pr})$  **)**

⟨proof⟩

### 7.4 Correctness Proof

**theorem** *pred-corr*:  $\text{invar-vebt } t \ n \implies \text{vebt-pred } t \ x = \text{Some } \text{px} \implies \text{is-pred-in-set} (\text{set-vebt}' t) \ x \ \text{px}$   
 ⟨proof⟩

**corollary** *pred-empty*: **assumes**  $\text{invar-vebt } t \ n$

**shows**  $(\text{vebt-pred } t \ x = \text{None}) = (\{y. \text{vebt-member } t \ y \wedge y < x\} = \{\})$   
 ⟨proof⟩

**theorem** *pred-correct*:  $\text{invar-vebt } t \ n \implies \text{vebt-pred } t \ x = \text{Some } sx \longleftrightarrow \text{is-pred-in-set } (\text{set-vebt } t) \ x \ sx$   
 ⟨proof⟩

**lemma** *helpypredd*:  $\text{invar-vebt } t \ n \implies \text{vebt-pred } t \ x = \text{Some } y \implies y < 2^{\widehat{n}}$   
 ⟨proof⟩

**lemma** *invar-vebt*  $t \ n \implies \text{vebt-pred } t \ x = \text{Some } y \implies y < x$   
 ⟨proof⟩

**end**  
**end**

**theory** *VEBT-Delete* **imports** *VEBT-Pred VEBT-Succ*  
**begin**

## 8 Deletion

### 8.1 Function Definition

**context** **begin**

**interpretation** *VEBT-internal* ⟨proof⟩

**fun** *vebt-delete* :: *VEBT*  $\Rightarrow$  *nat*  $\Rightarrow$  *VEBT* **where**

```

vebt-delete (Leaf a b) 0 = Leaf False b |
vebt-delete (Leaf a b) (Suc 0) = Leaf a False |
vebt-delete (Leaf a b) (Suc (Suc n)) = Leaf a b |
vebt-delete (Node None deg treeList summary) - = (Node None deg treeList summary) |
vebt-delete (Node (Some (mi, ma)) 0 trLst smry) x = (Node (Some (mi, ma)) 0 trLst smry) |
vebt-delete (Node (Some (mi, ma)) (Suc 0) tr sm) x = (Node (Some (mi, ma)) (Suc 0) tr sm) |
vebt-delete (Node (Some (mi, ma)) deg treeList summary) x =
  if (x < mi  $\vee$  x > ma) then (Node (Some (mi, ma)) deg treeList summary)
  else if (x = mi  $\wedge$  x = ma) then (Node None deg treeList summary)
  else let xn = (if x = mi
    then the (vebt-mint summary) * 2(deg div 2)
    + the (vebt-mint (treeList ! the (vebt-mint summary)))
    else x);
    minn = (if x = mi then xn else mi);
    l = low xn (deg div 2);
    h = high xn (deg div 2) in
    if h < length treeList
    then(
      let newnode = vebt-delete (treeList ! h) l;
        newlist = treeList[h:= newnode]in
      if minNull newnode
      then( let sn = vebt-delete summary h in(
        Node (Some (minn, if xn = ma then
          (let maxs = vebt-maxt sn in (
            if maxs = None
            then minn

```

```

else 2(deg div 2) * the maxs
      + the (vebt-maxt (newlist ! the maxs)))
else ma)) deg newList sn))
else (Node (Some (minn, (if xn = ma
                    then h * 2(deg div 2) + the( vebt-maxt (newlist ! h))
                    else ma))) deg newList summary ))
else (Node (Some (mi, ma)) deg treeList summary))

```

end

## 8.2 Auxiliary Lemmas

context *VEBT-internal* begin

context begin

lemma *delt-out-of-range*:

assumes  $x < mi \vee x > ma$  and  $deg \geq 2$

shows

*vebt-delete* (Node (Some (mi, ma)) deg treeList summary)  $x =$  (Node (Some (mi, ma)) deg treeList summary)  
 (proof)

lemma *del-single-cont*:

assumes  $x = mi \wedge x = ma$  and  $deg \geq 2$

shows

*vebt-delete* (Node (Some (mi, ma)) deg treeList summary)  $x =$  (Node None deg treeList summary)  
 (proof)

lemma *del-in-range*:

assumes  $x \geq mi \wedge x \leq ma$  and  $mi \neq ma$  and  $deg \geq 2$

shows

*vebt-delete* (Node (Some (mi, ma)) deg treeList summary)  $x =$  ( let xn = (if x = mi  
                                   then the (vebt-mint summary) \* 2<sup>(deg div 2)</sup>  
                                   + the (vebt-mint (treeList ! the (vebt-mint summary)))  
                                   else x);  
 minn = (if x = mi then xn else mi);  
 l = low xn (deg div 2);  
 h = high xn (deg div 2) in  
 if h < length treeList  
 then(  
   let newnode = vebt-delete (treeList ! h) l;  
   newlist = treeList[h:= newnode] in  
   if minNull newnode  
   then(  
     let sn = vebt-delete summary h in  
     (Node (Some (minn, if xn = ma then (let maxs = vebt-maxt sn in  
   (if maxs = None  
   then minn



```

else 2deg div 2 * the maxs
      + the (vebt-maxt (newlist ! the maxs))
    )
  )
  else ma))
    deg newlist sn)
  )else
    (Node (Some (minn, (if xn = ma then
      h * 2deg div 2 + the( vebt-maxt (newlist ! h))
      else ma)))
      deg newlist summary )
    )else
      (Node (Some (mi, ma)) deg treeList summary))
  )
  <proof>

```

**lemma** *del-x-not-mia*:

**assumes**  $x > mi \wedge x \leq ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ x\ (deg\ div\ 2) = h$  **and**  
 $low\ x\ (deg\ div\ 2) = l$  **and**  $high\ x\ (deg\ div\ 2) < length\ treeList$

**shows**

```

vebt-delete (Node (Some (mi, ma)) deg treeList summary) x = (
  let newnode = vebt-delete (treeList ! h) l;
  newlist = treeList[h:= newnode] in
  if minNull newnode
  then(
    let sn = vebt-delete summary h in
    (Node (Some (mi, if x = ma then (let maxs = vebt-maxt sn in
      (if maxs = None
        then mi
        else 2deg div 2 * the maxs
        + the (vebt-maxt (newlist ! the maxs))
      )
    )
    else ma))
      deg newlist sn)
  )else
    (Node (Some (mi, (if x = ma then
      h * 2deg div 2 + the( vebt-maxt (newlist ! h))
      else ma)))
      deg newlist summary )
  )
  <proof>

```

**lemma** *del-x-not-mi*:

**assumes**  $x > mi \wedge x \leq ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ x\ (deg\ div\ 2) = h$  **and**  
 $low\ x\ (deg\ div\ 2) = l$  **and**  $newnode = vebt-delete\ (treeList\ !\ h)\ l$   
**and**  $newlist = treeList[h:= newnode]$  **and**  $high\ x\ (deg\ div\ 2) < length\ treeList$

**shows**

```

vebt-delete (Node (Some (mi, ma)) deg treeList summary) x = (
  if minNull newnode

```

```

then(
  let sn = vebt-delete summary h in
  (Node (Some (mi, if x = ma then (let maxs = vebt-maxt sn in
    (if maxs = None
      then mi
      else 2deg div 2 * the maxs
      + the (vebt-maxt (newlist ! the maxs))
    )
  )
  )
  else ma))
  deg newlist sn)
)else
(Node (Some (mi, (if x = ma then
  h * 2deg div 2 + the( vebt-maxt (newlist ! h))
  else ma)))
  deg newlist summary )
) <proof>

```

**lemma** *del-x-not-mi-new-node-nil*:

**assumes**  $x > mi \wedge x \leq ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ x\ (deg\ div\ 2) = h$  **and**  
 $low\ x\ (deg\ div\ 2) = l$  **and**  $newnode = vebt-delete\ (treeList\ !\ h)\ l$  **and**  $minNull\ newnode$  **and**  
 $sn = vebt-delete\ summary\ h$  **and**  $newlist = treeList[h := newnode]$  **and**  $high\ x\ (deg\ div\ 2) < length\ treeList$

**shows**

$vebt-delete\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ x = (Node\ (Some\ (mi,\ if\ x = ma\ then$   
 $(let\ maxs = vebt-maxt\ sn\ in$

```

    (if maxs = None
      then mi
      else 2deg div 2 * the maxs
      + the (vebt-maxt (newlist ! the maxs))
    )
  )
  else ma)) deg newlist sn)

```

<proof>

**lemma** *del-x-not-mi-newnode-not-nil*:

**assumes**  $x > mi \wedge x \leq ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ x\ (deg\ div\ 2) = h$  **and**  
 $low\ x\ (deg\ div\ 2) = l$  **and**  $newnode = vebt-delete\ (treeList\ !\ h)\ l$  **and**  $\neg\ minNull\ newnode$  **and**  
 $newlist = treeList[h := newnode]$  **and**  $high\ x\ (deg\ div\ 2) < length\ treeList$

**shows**

$vebt-delete\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ x =$   
 $(Node\ (Some\ (mi,\ (if\ x = ma\ then$   
 $h * 2<sup>deg div 2</sup> + the( vebt-maxt (newlist ! h))$   
 $else\ ma)))$   
 $deg\ newlist\ summary\ )$

<proof>

**lemma** *del-x-mia*: **assumes**  $x = mi \wedge x < ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$

**shows**  $vebt-delete\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ x =$

```

let xn = the (vebt-mint summary) * 2^(deg div 2)
           + the (vebt-mint (treeList ! the (vebt-mint summary)));
           minn = xn;
l = low xn (deg div 2);
h = high xn (deg div 2) in
if h < length treeList
then(
  let newnode = vebt-delete (treeList ! h) l;
  newlist = treeList[h:= newnode]in
  if minNull newnode
  then(
    let sn = vebt-delete summary h in
    (Node (Some (minn, if xn = ma then (let maxs = vebt-maxt sn in
                                      (if maxs = None
                                       then minn
                                       else 2^(deg div 2) * the maxs
                                       + the (vebt-maxt (newlist ! the maxs))
                                      )
                                      )
          else ma))
      deg newlist sn)
  )else
  (Node (Some (minn, (if xn = ma then
                    h * 2^(deg div 2) + the( vebt-maxt (newlist ! h))
                    else ma)))
      deg newlist summary )
  )else
(Node (Some (mi, ma)) deg treeList summary)
)
⟨proof⟩

```

**lemma** *del-x-mi*:

**assumes**  $x = mi \wedge x < ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ xn\ (deg\ div\ 2) = h$  **and**  
 $xn = the\ (vebt-mint\ summary) * 2^{(deg\ div\ 2)} + the\ (vebt-mint\ (treeList\ !\ the\ (vebt-mint\ summary)))$

$low\ xn\ (deg\ div\ 2) = land\ high\ xn\ (deg\ div\ 2) < length\ treeList$

**shows**

```

vebt-delete (Node (Some (mi, ma)) deg treeList summary) x = (
  let newnode = vebt-delete (treeList ! h) l;
  newlist = treeList[h:= newnode]in
  if minNull newnode
  then(
    let sn = vebt-delete summary h in
    (Node (Some (xn, if xn = ma then (let maxs = vebt-maxt sn in
                                      (if maxs = None
                                       then xn
                                       else 2^(deg div 2) * the maxs
                                       + the (vebt-maxt (newlist ! the maxs))
                                      )
                                      )
          )
  )
)

```

```

    )
    else ma))
  deg newList sn)
)else
(Node (Some (xn, (if xn = ma then
  h * 2(deg div 2) + the( vebt-maxt (newlist ! h))
  else ma)))
  deg newList summary ))

```

*<proof>*

**lemma** *del-x-mi-lets-in:*

**assumes**  $x = mi \wedge x < ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ xn\ (deg\ div\ 2) = h$  **and**  
 $xn = the\ (vebt-mint\ summary) * 2^{(deg\ div\ 2)} + the\ (vebt-mint\ (treeList\ !\ the\ (vebt-mint\ summary)))$

$low\ xn\ (deg\ div\ 2) = l$  **and**  $high\ xn\ (deg\ div\ 2) < length\ treeList$  **and**

$newnode = vebt-delete\ (treeList\ !\ h)\ l$  **and**  $newlist = treeList[h:= newnode]$

**shows**  $vebt-delete\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ x =$  *if minNull newnode*  
*then*

```

  let sn = vebt-delete summary h in
  (Node (Some (xn, if xn = ma then (let maxs = vebt-maxt sn in
    (if maxs = None
      then xn
      else 2(deg div 2) * the maxs
      + the (vebt-maxt (newlist ! the maxs))
    )
  )
  else ma))
  deg newList sn)
)else
(Node (Some (xn, (if xn = ma then
  h * 2(deg div 2) + the( vebt-maxt (newlist ! h))
  else ma)))
  deg newList summary ))

```

*<proof>*

**lemma** *del-x-mi-lets-in-minNull:*

**assumes**  $x = mi \wedge x < ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ xn\ (deg\ div\ 2) = h$  **and**  
 $xn = the\ (vebt-mint\ summary) * 2^{(deg\ div\ 2)} + the\ (vebt-mint\ (treeList\ !\ the\ (vebt-mint\ summary)))$

$low\ xn\ (deg\ div\ 2) = l$  **and**  $high\ xn\ (deg\ div\ 2) < length\ treeList$  **and**

$newnode = vebt-delete\ (treeList\ !\ h)\ l$  **and**  $newlist = treeList[h:= newnode]$  **and**

$minNull\ newnode$  **and**  $sn = vebt-delete\ summary\ h$

**shows**

$vebt-delete\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ x =$   
 $(Node\ (Some\ (xn,\ if\ xn = ma\ then\ (let\ maxs = vebt-maxt\ sn\ in$   
 $(if\ maxs = None$   
 $then\ xn$   
 $else\ 2^{(deg\ div\ 2)} * the\ maxs$

+ the (vebt-maxt (newlist ! the maxs))  
 )  
 )  
 else ma)) deg newlist sn)

⟨proof⟩

**lemma** *del-x-mi-lets-in-not-minNull*:

**assumes**  $x = mi \wedge x < ma$  **and**  $mi \neq ma$  **and**  $deg \geq 2$  **and**  $high\ xn\ (deg\ div\ 2) = h$  **and**  
 $xn = the\ (vebt-mint\ summary) * 2^{(deg\ div\ 2)} + the\ (vebt-mint\ (treeList\ !\ the\ (vebt-mint\ summary)))$   
 $low\ xn\ (deg\ div\ 2) =$  **land**  $high\ xn\ (deg\ div\ 2) < length\ treeList$  **and**  
 $newnode = vebt-delete\ (treeList\ !\ h)\ l$  **and**  $newlist = treeList[h := newnode]$  **and**  
 $\neg minNull\ newnode$

**shows**

$vebt-delete\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ x =$   
 $(Node\ (Some\ (xn,\ (if\ xn = ma\ then$   
 $h * 2^{(deg\ div\ 2)} + the\ (vebt-maxt\ (newlist\ !\ h))$   
 $else\ ma)))$   
 $deg\ newlist\ summary)$

⟨proof⟩

**theorem** *dele-bmo-cont-corr*:  $invar-vebt\ t\ n \implies (both-member-options\ (vebt-delete\ t\ x)\ y \longleftrightarrow x \neq y \wedge both-member-options\ t\ y)$

⟨proof⟩

**end**

**corollary**  $invar-vebt\ t\ n \implies both-member-options\ t\ x \implies x \neq y \implies both-member-options\ (vebt-delete\ t\ y)\ x$

⟨proof⟩

**corollary**  $invar-vebt\ t\ n \implies both-member-options\ t\ x \implies \neg both-member-options\ (vebt-delete\ t\ x)\ x$

⟨proof⟩

**corollary**  $invar-vebt\ t\ n \implies both-member-options\ (vebt-delete\ t\ y)\ x \implies both-member-options\ t\ x \wedge x \neq y$

⟨proof⟩

**end**

**end**

**theory** *VEBT-DeleteCorrectness* **imports** *VEBT-Delete*  
**begin**

**context** *VEBT-internal* **begin**

### 8.3 Validness Preservation

**theorem** *delete-pres-valid*:  $invar-vebt\ t\ n \implies invar-vebt\ (vebt-delete\ t\ x)\ n$

*<proof>*

**corollary** *dele-member-cont-corr:invar-vebt*  $t\ n \implies (\text{vebt-member } (\text{vebt-delete } t\ x)\ y \longleftrightarrow x \neq y \wedge \text{vebt-member } t\ y)$   
*<proof>*

## 8.4 Correctness with Respect to Set Interpretation

**theorem** *delete-correct'*: **assumes** *invar-vebt*  $t\ n$   
**shows**  $\text{set-vebt}' (\text{vebt-delete } t\ x) = \text{set-vebt}' t - \{x\}$   
*<proof>*

**corollary** *delete-correct*: **assumes** *invar-vebt*  $t\ n$   
**shows**  $\text{set-vebt}' (\text{vebt-delete } t\ x) = \text{set-vebt } t - \{x\}$   
*<proof>*

**end**  
**end**

**theory** *VEBT-Uniqueness* **imports** *VEBT-InsertCorrectness VEBT-Succ VEBT-Pred VEBT-DeleteCorrectness*  
**begin**

**context** *VEBT-internal* **begin**

## 9 Uniqueness Property of valid Trees

Two valid van Emde Boas trees having equal degree number and representing the same integer set are equal.

**theorem** *uniquetree*:  $\text{invar-vebt } t\ n \implies \text{invar-vebt } s\ n \implies \text{set-vebt}' t = \text{set-vebt}' s \implies s = t$   
*<proof>*

**corollary**  $\text{invar-vebt } t\ n \implies \text{set-vebt}' t = \{\} \implies t = \text{vebt-buildup } n$   
*<proof>*

**corollary** *unique-tree*:  $\text{invar-vebt } t\ n \implies \text{invar-vebt } s\ n \implies \text{set-vebt } t = \text{set-vebt } s \implies s = t$   
*<proof>*

**corollary**  $\text{invar-vebt } t\ n \implies \text{set-vebt } t = \{\} \implies t = \text{vebt-buildup } n$   
*<proof>*

All valid trees can be generated by *vebt – insertion* chains on an empty tree with same degree parameter:

**inductive** *perInsTrans::VEBT*  $\Rightarrow$  *VEBT*  $\Rightarrow$  *bool* **where**  
*perInsTrans*  $t\ t$  |  
 $(t = \text{vebt-insert } s\ x) \implies \text{perInsTrans } t\ u \implies \text{perInsTrans } s\ u$

**lemma** *perIT-concat*:  $\text{perInsTrans } s\ t \implies \text{perInsTrans } t\ u \implies \text{perInsTrans } s\ u$

*<proof>*

**lemma** *assumes* *invar-vebt t n* **shows**

*perInsTrans (vebt-buildup n) t*

*<proof>*

**end**

**end**

**theory** *VEBT-Height* **imports** *VEBT-Definitions Complex-Main*

**begin**

**context** *VEBT-internal* **begin**

## 10 Heights of van Emde Boas Trees

**fun** *height::VEBT*  $\Rightarrow$  *nat* **where**

*height (Leaf a b) = 0*

*height (Node - deg treeList summary) = (1+ Max (height ‘ (insert summary (set treeList))))*

**abbreviation** *lb x*  $\equiv$  *log 2 x*

**lemma** *setceilmax: invar-vebt s m*  $\implies \forall t \in$  *set listy. invar-vebt t n*

$\implies m = \text{Suc } n \implies (\forall t \in \text{set listy. } \text{height } t = \lceil \text{lb } n \rceil) \implies \text{height } s = \lceil \text{lb } m \rceil$

$\implies \text{Max (height ‘ (insert s (set listy)))} = \lceil \text{lb } m \rceil$

*<proof>*

**lemma** *log-ceil-idem:*

**assumes** *(x::real)  $\geq 1$*

**shows**  $\lceil \text{lb } x \rceil = \lceil \text{lb } \lceil x \rceil \rceil$

*<proof>*

**lemma** *height-uplog-rel: invar-vebt t n*  $\implies (\text{height } t) = \lceil \text{lb } n \rceil$

*<proof>*

**lemma** *two-powr-height-bound-deg:*

**assumes** *invar-vebt t n*

**shows**  $2^{\lceil \text{height } t \rceil} \leq 2 * (n::\text{nat})$

*<proof>*

Main Theorem

**theorem** *height-double-log-univ-size:*

**assumes**  $u = 2^{\text{deg}}$  **and** *invar-vebt t deg*

**shows**  $\text{height } t \leq 1 + \text{lb } (\text{lb } u)$

*<proof>*

**lemma** *height-compose-list: t*  $\in$  *set treeList*  $\implies$

$\text{Max (height ‘ (insert summary (set treeList)))} \geq \text{height } t$

*<proof>*

```

lemma height-compose-child:  $t \in \text{set treeList} \implies$ 
  height (Node info deg treeList summary)  $\geq 1 + \text{height } t$  <proof>

lemma height-compose-summary: height (Node info deg treeList summary)  $\geq 1 + \text{height summary}$ 
  <proof>

lemma height-i-max:  $i < \text{length } x13 \implies$ 
  height (x13 ! i)  $\leq \max \text{foo } (\text{Max } (\text{height ' set } x13))$ 
  <proof>

lemma max-ins-scaled:  $n * \text{height } x14 \leq m + n * \text{Max } (\text{insert } (\text{height } x14) (\text{height ' set } x13))$ 
  <proof>

lemma max-idx-list:
  assumes  $i < \text{length } x13$ 
  shows  $n * \text{height } (x13 ! i) \leq \text{Suc } (\text{Suc } (n * \max (\text{height } x14) (\text{Max } (\text{height ' set } x13))))$ 
  <proof>

end
end

theory VEBT-Bounds imports VEBT-Height VEBT-Member VEBT-Insert VEBT-Succ VEBT-Pred
begin

```

## 11 Upper Bounds for canonical Functions: Relationships between Run Time and Tree Heights

### 11.1 Membership test

```

context begin

```

```

  interpretation VEBT-internal <proof>

```

```

fun T_member::VEBT  $\Rightarrow$  nat  $\Rightarrow$  nat where

```

```

  T_member (Leaf a b) x = 2 + (if x = 0 then 1 else 1 + (if x=1 then 1 else 1))|

```

```

  T_member (Node None - -) x = 2|

```

```

  T_member (Node - 0 -) x = 2|

```

```

  T_member (Node - (Suc 0) -) x = 2|

```

```

  T_member (Node (Some (mi, ma)) deg treeList summary) x = 2 + (

```

```

    if x = mi then 1 else 1+ (

```

```

      if x = ma then 1 else 1+ (

```

```

        if x < mi then 1 else 1+ (

```

```

          if x > ma then 1 else 9 +

```

```

            (let

```

```

              h = high x (deg div 2);

```

```

              l = low x (deg div 2) in

```

```

              (if h < length treeList

```

```

                then 1 + T_member (treeList ! h) l

```



else 1))))))

**fun**  $T_{member}'::VEBT \Rightarrow nat \Rightarrow nat$  **where**  
 $T_{member}' (Leaf\ a\ b)\ x = 1$   
 $T_{member}' (Node\ None\ -\ -)\ x = 1$   
 $T_{member}' (Node\ 0\ -)\ x = 1$   
 $T_{member}' (Node\ (Suc\ 0)\ -)\ x = 1$   
 $T_{member}' (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ x = 1 +$   
 if  $x = mi$  then 0 else (  
 if  $x = ma$  then 0 else (  
 if  $x < mi$  then 0 else (  
 if  $x > ma$  then 0 else if  $(x > mi \wedge x < ma)$  then  
 (let  
    $h = high\ x\ (deg\ div\ 2)$ ;  
    $l = low\ x\ (deg\ div\ 2)$  in  
   (if  $h < length\ treeList$   
    then  $T_{member}' (treeList\ !\ h)\ l$   
    else 0))  
 else 0)))))

**lemma** *height-node*:  $invar\ vebt\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ n$   
 $\implies height\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary) \geq 1$   
 <proof>

**theorem** *member-bound-height*:  $invar\ vebt\ t\ n \implies T_{member}\ t\ x \leq (1 + height\ t) * 15$   
 <proof>

**theorem** *member-bound-height'*:  $invar\ vebt\ t\ n \implies T_{member}'\ t\ x \leq (1 + height\ t)$   
 <proof>

**theorem** *member-bound-size-univ*:  $invar\ vebt\ t\ n \implies u = 2^n \implies T_{member}\ t\ x \leq 30 + 15 * lb\ (lb\ u)$   
 <proof>

## 11.2 Minimum, Maximum, Emptiness Test

**fun**  $T_{mint}::VEBT \Rightarrow nat$  **where**  
 $T_{mint} (Leaf\ a\ b) = (1 + (if\ a\ then\ 0\ else\ 1 + (if\ b\ then\ 1\ else\ 1)))$   
 $T_{mint} (Node\ None\ -\ -) = 1$   
 $T_{mint} (Node\ (Some\ (mi,\ ma))\ -\ -) = 1$

**lemma** *mint-bound*:  $T_{mint}\ t \leq 3$  <proof>

**fun**  $T_{maxt}::VEBT \Rightarrow nat$  **where**  
 $T_{maxt} (Leaf\ a\ b) = (1 + (if\ b\ then\ 1\ else\ 1 + (if\ a\ then\ 1\ else\ 1)))$   
 $T_{maxt} (Node\ None\ -\ -) = 1$

$T_{maxt} (Node (Some (mi,ma)) - - -) = 1$

**lemma** *maxt-bound*:  $T_{maxt} t \leq 3$  *<proof>*

**fun**  $T_{minNull}::VEBT \Rightarrow nat$  **where**

$T_{minNull} (Leaf False False) = 1|$   
 $T_{minNull} (Leaf - -) = 1|$   
 $T_{minNull} (Node None - - -) = 1|$   
 $T_{minNull} (Node (Some -) - - -) = 1$

**lemma** *minNull-bound*:  $T_{minNull} t \leq 1$   
*<proof>*

### 11.3 Insertion

**fun**  $T_{insert}::VEBT \Rightarrow nat \Rightarrow nat$  **where**

$T_{insert} (Leaf a b) x = 1 + (if\ x=0\ then\ 1\ else\ 1 + (if\ x=1\ then\ 1\ else\ 1))|$   
 $T_{insert} (Node\ info\ 0\ ts\ s) x = 1|$   
 $T_{insert} (Node\ info\ (Suc\ 0)\ ts\ s) x = 1|$   
 $T_{insert} (Node\ None\ (Suc\ deg)\ treeList\ summary) x = 2|$   
 $T_{insert} (Node\ (Some\ (mi,ma))\ deg\ treeList\ summary) x = 19 +$   
 $(\ let\ xn = (if\ x < mi\ then\ mi\ else\ x);\ minn = (if\ x < mi\ then\ x\ else\ mi);$   
 $\ l = low\ xn\ (deg\ div\ 2); h = high\ xn\ (deg\ div\ 2)$   
 $\ in$   
 $\ (if\ h < length\ treeList \wedge \neg (x = mi \vee x = ma)\ then$   
 $\ T_{insert} (treeList ! h) l + T_{minNull} (treeList ! h) +$   
 $\ (if\ minNull (treeList ! h)\ then\ T_{insert}\ summary\ h\ else\ 1)$   
 $\ else\ 1))$

**fun**  $T_{insert}'::VEBT \Rightarrow nat \Rightarrow nat$  **where**

$T_{insert}' (Leaf a b) x = 1|$   
 $T_{insert}' (Node\ info\ 0\ ts\ s) x = 1|$   
 $T_{insert}' (Node\ info\ (Suc\ 0)\ ts\ s) x = 1|$   
 $T_{insert}' (Node\ None\ (Suc\ deg)\ treeList\ summary) x = 1|$   
 $T_{insert}' (Node\ (Some\ (mi,ma))\ deg\ treeList\ summary) x =$   
 $(let\ xn = (if\ x < mi\ then\ mi\ else\ x); minn = (if\ x < mi\ then\ x\ else\ mi);$   
 $\ l = low\ xn\ (deg\ div\ 2); h = high\ xn\ (deg\ div\ 2)$   
 $\ in\ (if\ h < length\ treeList \wedge \neg (x = mi \vee x = ma)\ then$   
 $\ T_{insert}' (treeList ! h) l +$   
 $\ (if\ minNull (treeList ! h)\ then\ T_{insert}'\ summary\ h\ else\ 1)\ else\ 1))$

**lemma** *insertsimp:assumes invar-vebt t n and*  $\nexists x.$  *both-member-options t x shows*  $T_{insert} t y \leq 3$   
*<proof>*

**lemma** *insertsimp*:  $invar-vebt\ t\ n \implies minNull\ t \implies T_{insert}\ t\ l \leq 3$   
*<proof>*

**lemma** *insersimp'*: *assumes* *invar-vebt* *t n* **and**  $\nexists x$ . *both-member-options* *t x* **shows**  $T_{insert}' t y \leq 1$

*<proof>*

**lemma** *insertsimp'*: *invar-vebt* *t n*  $\implies$  *minNull* *t*  $\implies$   $T_{insert}' t l \leq 1$

*<proof>*

**theorem** *insert-bound-height*: *invar-vebt* *t n*  $\implies$   $T_{insert} t x \leq (1+height t)*23$

*<proof>*

**theorem** *insert-bound-size-univ*: *invar-vebt* *t n*  $\implies$   $u = 2^{\widehat{n}} \implies T_{insert} t x \leq 46 + 23 * lb (lb u)$

*<proof>*

**theorem** *insert'-bound-height*: *invar-vebt* *t n*  $\implies$   $T_{insert}' t x \leq (1+height t)$

*<proof>*

## 11.4 Successor Function

**fun**  $T_{succ}::VEBT \Rightarrow nat \Rightarrow nat$  **where**

$T_{succ} (Leaf - b) 0 = 1 + (if\ b\ then\ 1\ else\ 1)|$   
 $T_{succ} (Leaf - -) (Suc\ n) = 1|$   
 $T_{succ} (Node\ None\ -\ -) - = 1|$   
 $T_{succ} (Node - 0 - -) - = 1|$   
 $T_{succ} (Node - (Suc\ 0) - -) - = 1|$   
 $T_{succ} (Node (Some (mi, ma)) deg\ treeList\ summary) x = 1 + ($   
 $\quad if\ x < mi\ then\ 1$   
 $\quad else\ (let\ l = low\ x\ (deg\ div\ 2); h = high\ x\ (deg\ div\ 2)\ in\ 10 +$   
 $\quad (if\ h < length\ treeList\ then\ 1 + T_{maxt} (treeList ! h) + ($   
 $\quad \quad let\ maxlow = vebt-maxt (treeList ! h)\ in\ 3 +$   
 $\quad \quad (if\ maxlow \neq None \wedge (Some\ l <_o\ maxlow)\ then$   
 $\quad \quad \quad 4 + T_{succ} (treeList ! h) l$   
 $\quad \quad else\ let\ sc = vebt-succ\ summary\ h\ in\ 1 + T_{succ}\ summary\ h + 1 + ($   
 $\quad \quad if\ sc = None\ then\ 1$   
 $\quad \quad else\ (4 + T_{mint} (treeList ! the\ sc) ))))$   
 $\quad else\ 1))$

**fun**  $T_{succ}'::VEBT \Rightarrow nat \Rightarrow nat$  **where**

$T_{succ}' (Leaf - b) 0 = 1|$   
 $T_{succ}' (Leaf - -) (Suc\ n) = 1|$   
 $T_{succ}' (Node\ None\ -\ -) - = 1|$   
 $T_{succ}' (Node - 0 - -) - = 1|$   
 $T_{succ}' (Node - (Suc\ 0) - -) - = 1|$   
 $T_{succ}' (Node (Some (mi, ma)) deg\ treeList\ summary) x = ($   
 $\quad if\ x < mi\ then\ 1$   
 $\quad else\ (let\ l = low\ x\ (deg\ div\ 2); h = high\ x\ (deg\ div\ 2)\ in$   
 $\quad (if\ h < length\ treeList\ then\ ($   
 $\quad \quad let\ maxlow = vebt-maxt (treeList ! h)\ in$

(if maxlow  $\neq$  None  $\wedge$  (Some l  $<_o$  maxlow) then  
     1+ T<sub>succ'</sub> (treeList ! h) l  
     else let sc = vebt-succ summary h in T<sub>succ'</sub> summary h + (  
       if sc = None then 1  
       else 1 )))  
 else 1)))

**theorem succ-bound-height:** invar-vebt t n  $\implies$  T<sub>succ</sub> t x  $\leq$  (1+height t)\*27  
 <proof>

**theorem succ-bound-size-univ:** invar-vebt t n  $\implies$  u = 2<sup>n</sup>  $\implies$  T<sub>succ</sub> t x  $\leq$  54 + 27 \* lb (lb u)  
 <proof>

**theorem succ'-bound-height:** invar-vebt t n  $\implies$  T<sub>succ'</sub> t x  $\leq$  (1+height t)  
 <proof>

**theorem succ-bound-size-univ':** invar-vebt t n  $\implies$  u = 2<sup>n</sup>  $\implies$  T<sub>succ'</sub> t x  $\leq$  2 + lb (lb u)  
 <proof>

## 11.5 Predecessor Function

**fun** T<sub>pred</sub>::VEBT  $\Rightarrow$  nat  $\Rightarrow$  nat **where**

T<sub>pred</sub> (Leaf - -) 0 = 1|  
 T<sub>pred</sub> (Leaf a -) (Suc 0) = 1+ (if a then 1 else 1)|  
 T<sub>pred</sub> (Leaf a b) - = 1+ (if b then 1 else 1+ (if a then 1 else 1))|  
  
 T<sub>pred</sub> (Node None - -) - = 1|  
 T<sub>pred</sub> (Node - 0 -) - = 1|  
 T<sub>pred</sub> (Node - (Suc 0) -) - = 1|  
 T<sub>pred</sub> (Node (Some (mi, ma)) deg treeList summary) x = 1+ (  
   if x > ma then 1  
   else (let l = low x (deg div 2); h = high x (deg div 2) in 10 + 1+  
       (if h < length treeList then  
  
       let minlow = vebt-mint (treeList ! h) in 2 + T<sub>mint</sub>(treeList ! h) + 3 +  
       (if minlow  $\neq$  None  $\wedge$  (Some l  $>_o$  minlow) then  
           4 + T<sub>pred</sub> (treeList ! h) l  
       else let pr = vebt-pred summary h in 1 + T<sub>pred</sub> summary h + 1 + (  
           if pr = None then 1 + (if x > mi then 1 else 1)  
           else 4 + T<sub>maxt</sub> (treeList ! the pr) ))  
       else 1)))

**theorem pred-bound-height:** invar-vebt t n  $\implies$  T<sub>pred</sub> t x  $\leq$  (1+height t)\*29  
 <proof>

**theorem pred-bound-size-univ:** invar-vebt t n  $\implies$  u = 2<sup>n</sup>  $\implies$  T<sub>pred</sub> t x  $\leq$  58 + 29 \* lb (lb u)  
 <proof>

**fun**  $T_{pred}'::VEBT \Rightarrow nat \Rightarrow nat$  **where**

$T_{pred}'(Leaf \ - \ -) \ 0 = 1$  |

$T_{pred}'(Leaf \ a \ -) \ (Suc \ 0) = 1$  |

$T_{pred}'(Leaf \ a \ b) \ - = 1$  |

$T_{pred}'(Node \ None \ - \ -) \ - = 1$  |

$T_{pred}'(Node \ - \ 0 \ -) \ - = 1$  |

$T_{pred}'(Node \ - \ (Suc \ 0) \ -) \ - = 1$  |

$T_{pred}'(Node \ (Some \ (mi, \ ma)) \ deg \ treeList \ summary) \ x =$  (

*if*  $x > ma$  *then* 1

*else* (*let*  $l = low \ x \ (deg \ div \ 2)$ ;  $h = high \ x \ (deg \ div \ 2)$  *in*

*(if*  $h < length \ treeList$  *then*

*let*  $minlow = vebt\text{-}mint \ (treeList \ ! \ h)$  *in*

*(if*  $minlow \neq None \wedge (Some \ l >_o \ minlow)$  *then*

$1 + T_{pred}'(treeList \ ! \ h) \ l$

*else let*  $pr = vebt\text{-}pred \ summary \ h$  *in*  $T_{pred}' \ summary \ h +$  (

*if*  $pr = None$  *then* 1

*else* 1))

*else* 1)))

**theorem** *pred-bound-height'*:  $invar\text{-}vebt \ t \ n \Longrightarrow T_{pred}' \ t \ x \leq (1 + height \ t)$

*<proof>*

**theorem** *pred-bound-size-univ'*:  $invar\text{-}vebt \ t \ n \Longrightarrow u = 2^{\widehat{n}} \Longrightarrow T_{pred}' \ t \ x \leq 2 + lb \ (lb \ u)$

*<proof>*

**end**

**end**

**theory** *VEBT-DeleteBounds* **imports** *VEBT-Bounds* *VEBT-Delete* *VEBT-DeleteCorrectness*

**begin**

## 11.6 Running Time Bounds for Deletion

**context** **begin**

**interpretation** *VEBT-internal* *<proof>*

**fun**  $T_{delete}'::VEBT \Rightarrow nat \Rightarrow nat$  **where**

$T_{delete}'(Leaf \ a \ b) \ 0 = 1$  |

$T_{delete}'(Leaf \ a \ b) \ (Suc \ 0) = 1$  |

$T_{delete}'(Leaf \ a \ b) \ (Suc \ (Suc \ n)) = 1$  |

$T_{delete}'(Node \ None \ deg \ treeList \ summary) \ - = 1$  |

$T_{delete}'(Node \ (Some \ (mi, \ ma)) \ 0 \ treeList \ summary) \ x = 1$  |

$T_{delete}'(Node \ (Some \ (mi, \ ma)) \ (Suc \ 0) \ treeList \ summary) \ x = 1$  |

$T_{delete}'(Node \ (Some \ (mi, \ ma)) \ deg \ treeList \ summary) \ x = 3 +$  (

*if*  $(x < mi \vee x > ma)$  *then* 1

*else*  $3 +$  (*if*  $(x = mi \wedge x = ma)$  *then* 3

*else*  $13 +$  (*if*  $x = mi$  *then*  $T_{mint} \ summary + T_{mint} \ (treeList \ ! \ the \ (vebt\text{-}mint \ summary))$ ) $+$

```

7 else 1 )+
      (if x = mi then 1 else 1) +
      ( let xn = (if x = mi
                  then the (vebt-mint summary) * 2^(deg div 2) + the (vebt-mint (treeList ! the
(vebt-mint summary)))
                  else x);
        minn = (if x = mi then xn else mi);
        l = low xn (deg div 2);
        h = high xn (deg div 2) in
        if h < length treeList
          then( 4 + T_delete (treeList ! h) l +(
                let newnode = vebt-delete (treeList ! h) l;
                newlist = treeList[h:= newnode]in 1 + T_minNull newnode + (
                if minNull newnode
                  then( 1 + T_delete summary h + (
                      let sn = vebt-delete summary h in
                      2+ (if xn = ma then 1 + T_maxt sn + (let maxs = vebt-maxt sn in
                                                                1 + (if maxs = None
                                                                    then 1
                                                                    else 8+ T_maxt (newlist ! the maxs)
                                                                ) )
                      else 1)
                  ))else
                2 + (if xn = ma then 6+ T_maxt (newlist ! h) else 1)
          )))else 1 )))

```

**end**

**context** *VEBT-internal* **begin**

**lemma** *tdeletemimi*:  $\text{deg} \geq 2 \implies T_{\text{delete}} (\text{Node} (\text{Some} (mi, mi)) \text{deg} \text{treeList} \text{summary}) x \leq 9$   
 ⟨proof⟩

**lemma** *minNull-delete-time-bound*:  $\text{invar-vebt } t \ n \implies \text{minNull} (\text{vebt-delete } t \ x) \implies T_{\text{delete}} t \ x \leq 9$   
 ⟨proof⟩

**lemma** *delete-bound-height*:  $\text{invar-vebt } t \ n \implies T_{\text{delete}} t \ x \leq (1 + \text{height } t) * 70$   
 ⟨proof⟩

**theorem** *delete-bound-size-univ*:  $\text{invar-vebt } t \ n \implies u = 2^{\wedge} n \implies T_{\text{delete}} t \ x \leq 140 + 70 * \text{lb} (\text{lb } u)$   
 ⟨proof⟩

**fun**  $T_{\text{delete}}' :: \text{VEBT} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **where**

```

T_delete' (Leaf a b) 0 = 1|
T_delete' (Leaf a b) (Suc 0) = 1|
T_delete' (Leaf a b) (Suc (Suc n)) = 1|
T_delete' (Node None deg treeList summary) - = 1|
T_delete' (Node (Some (mi, ma)) 0 treeList summary) x = 1|

```

```

T_delete' (Node (Some (mi, ma)) (Suc 0) treeList summary) x = 1 |
T_delete' (Node (Some (mi, ma)) deg treeList summary) x = (
  if (x < mi ∨ x > ma) then 1
  else if (x = mi ∧ x = ma) then 1
  else ( let xn = (if x = mi
    then the (vebt-mint summary) * 2^(deg div 2) + the (vebt-mint (treeList ! the
(vebt-mint summary)))
    else x);
  minn = (if x = mi then xn else mi);
  l = low xn (deg div 2);
  h = high xn (deg div 2) in
  if h < length treeList
  then( T_delete' (treeList ! h) l +(
    let newnode = vebt-delete (treeList ! h) l;
    newlist = treeList[h:= newnode]in
    if minNull newnode
    then T_delete' summary h
    else 1
  ))else 1 ))

```

**lemma** *tdeletemimi'*:  $\text{deg} \geq 2 \implies T_{\text{delete}'} (\text{Node } (\text{Some } (mi, mi)) \text{ deg treeList summary}) x \leq 1$   
 ⟨proof⟩

**lemma** *minNull-delete-time-bound'*:  $\text{invar-vebt } t \ n \implies \text{minNull } (\text{vebt-delete } t \ x) \implies T_{\text{delete}'} t \ x \leq 1$   
 ⟨proof⟩

**lemma** *delete-bound-height'*:  $\text{invar-vebt } t \ n \implies T_{\text{delete}'} t \ x \leq 1 + \text{height } t$   
 ⟨proof⟩

**theorem** *delete-bound-size-univ'*:  $\text{invar-vebt } t \ n \implies u = 2^n \implies T_{\text{delete}'} t \ x \leq 2 + \text{lb } (\text{lb } u)$   
 ⟨proof⟩

**end**  
**end**

**theory** *VEBT-Space* **imports** *VEBT-Definitions Complex-Main*  
**begin**

## 12 Space Complexity and *buildup* Time Consumption

### 12.1 Space Complexity of valid van Emde Boas Trees

Space Complexity is linear in relation to universe sizes

**context** *VEBT-internal* **begin**

**fun** *space*:: *VEBT*  $\Rightarrow$  *nat* **where**

$space (Leaf a b) = 3|$   
 $space (Node info deg treeList summary) = 5 + space summary + length treeList + foldr (\lambda a b. a+b)$   
 $(map space treeList) 0$

**fun**  $space'$ ::  $VEBT \Rightarrow nat$  **where**  
 $space' (Leaf a b) = 4|$   
 $space' (Node info deg treeList summary) = 6 + space' summary + foldr (\lambda a b. a+b) (map space'$   
 $treeList) 0$

Count in reals

**fun**  $cnt$ ::  $VEBT \Rightarrow real$  **where**  
 $cnt (Leaf a b) = 1|$   
 $cnt (Node info deg treeList summary) = 1 + cnt summary + foldr (\lambda a b. a+b) (map cnt treeList) 0$

## 12.2 Auxiliary Lemmas for List Summation

**lemma**  $list\text{-}every\text{-}elemnt\text{-}bound\text{-}sum\text{-}bound$ :  $\forall x \in set\ xs. f\ x \leq bound \implies foldr (\lambda a b. a+b) (map\ f\ xs) i \leq length\ xs * bound + i$   
 $\langle proof \rangle$

**lemma**  $list\text{-}every\text{-}elemnt\text{-}bound\text{-}sum\text{-}bound\text{-}real$ :  $\forall x \in set\ (xs::'a\ list). (f::'a \Rightarrow real)\ x \leq (bound::real)$   
 $\implies foldr (\lambda a b. a+b) (map\ f\ xs) i \leq real(length\ xs) * bound + i$   
 $\langle proof \rangle$

**lemma**  $foldr\text{-}one$ :  $d \leq foldr (+) ys (d::nat)$   
 $\langle proof \rangle$

**lemma**  $foldr\text{-}zero$ :  $\forall i < length\ xs. xs ! i > 0 \implies$   
 $foldr (\lambda a b. a+b) xs (d::nat) - d \geq length\ xs$   
 $\langle proof \rangle$

**lemma**  $foldr\text{-}mono$ :  $length\ xs = length\ ys \implies \forall i < length\ xs. xs ! i < ys ! i \implies c \leq d \implies$   
 $foldr (\lambda a b. a+b) xs\ c + length\ ys \leq foldr (\lambda a b. a+b) ys (d::nat)$   
 $\langle proof \rangle$

**lemma**  $two\text{-}realpow\text{-}ge\text{-}two$  :  $(n::real) \geq 1 \implies (2::real)^{\hat{n}} \geq 2$   
 $\langle proof \rangle$

**lemma**  $foldr0$ :  $foldr (+) xs (c+d) = foldr (+) xs (d::real) + c$   
 $\langle proof \rangle$

**lemma**  $f\text{-}g\text{-}map\text{-}foldr\text{-}bound$ :  $(\forall x \in set\ xs. f\ x \leq c * g\ x)$   
 $\implies foldr (\lambda a b. a+b) (map\ f\ xs) d \leq c * foldr (\lambda a b. a+b) (map\ g\ xs) (0::real) + d$   
 $\langle proof \rangle$

**lemma**  $real\text{-}nat\text{-}list$ :  $real (foldr (+) (map\ f\ xs) (c::nat))$   
 $= foldr (+) (map (\lambda x. real(f\ x)) xs) c$   
 $\langle proof \rangle$



### 12.3 Actual Space Reasoning

**lemma** *space-space'*:  $space' t > space t$   
 ⟨proof⟩

**lemma** *cnt-bound*:  
**defines**  $c \equiv 1.5$   
**shows**  $invar-vebt t n \implies cnt t \leq 2 * ((2^{\wedge} n - c)::real)$   
 ⟨proof⟩

**theorem** *cnt-bound'*:  $invar-vebt t n \implies cnt t \leq 2 * (2^{\wedge} n - 1)$   
 ⟨proof⟩

**lemma** *space-cnt*:  $space' t \leq 6 * cnt t$   
 ⟨proof⟩

**lemma** *space-2-pow-bound*: **assumes**  $invar-vebt t n$  **shows**  $real (space' t) \leq 12 * (2^{\wedge} n - 1)$   
 ⟨proof⟩

**lemma** *space'-bound*: **assumes**  $invar-vebt t n u = 2^{\wedge} n$   
**shows**  $space' t \leq 12 * u$   
 ⟨proof⟩

Main Theorem

**theorem** *space-bound*: **assumes**  $invar-vebt t n u = 2^{\wedge} n$   
**shows**  $space t \leq 12 * u$   
 ⟨proof⟩

### 12.4 Complexity of Generation Time

Space complexity is closely related to tree generation time complexity

Time approximation for replicate function.  $T_{replicate} n t x$  denotes running time of the  $n$ -times replication of  $x$  into a list.  $t$  models runtime for generation of a single  $x$ .

**fun**  $T_{buildup}::nat \Rightarrow nat$  **where**  
 $T_{buildup} 0 = 3$   
 $T_{buildup} (Suc 0) = 3$   
 $T_{buildup} n = (if\ even\ n\ then\ 1 + (let\ half = n\ div\ 2\ in$   
 $\quad 9 + T_{buildup}\ half + (2^{\wedge} half) * (T_{buildup}\ half + 1))$   
 $\quad else\ (let\ half = n\ div\ 2\ in$   
 $\quad 11 + T_{buildup}\ (Suc\ half) + (2^{\wedge}(Suc\ half))* (T_{buildup}\ half + 1)))$

**fun**  $T_{build}::nat \Rightarrow nat$  **where**  
 $T_{build} 0 = 4$   
 $T_{build} (Suc 0) = 4$   
 $T_{build} n = (if\ even\ n\ then\ 1 + (let\ half = n\ div\ 2\ in$   
 $\quad 10 + T_{build}\ half + (2^{\wedge} half) * (T_{build}\ half))$   
 $\quad else\ (let\ half = n\ div\ 2\ in$   
 $\quad 12 + T_{build}\ (Suc\ half) + (2^{\wedge}(Suc\ half))* (T_{build}\ half)))$

**lemma** *buildup-build-time*:  $T_{buildup} n < T_{build} n$   
 ⟨proof⟩

**lemma** *listsum-bound*:  $(\bigwedge x. x \in set\ xs \implies f\ x \geq (0::real)) \implies$   
 $foldr\ (+)\ (map\ f\ xs)\ y \geq y$   
 ⟨proof⟩

**lemma** *cnt-non-neg*:  $cnt\ t \geq 0$   
 ⟨proof⟩

**lemma** *foldr-same*:  $(\bigwedge x\ y. x \in set\ (xs::real\ list) \implies y \in set\ xs \implies x = y) \implies$   
 $(\bigwedge x. (x::real) \in set\ xs \implies x = (y::real)) \implies$   
 $foldr\ (\lambda\ (a::real)\ (b::real).\ a+b)\ xs\ 0 = real\ (length\ xs) * y$   
 ⟨proof⟩

**lemma** *foldr-same-int*:  $(\bigwedge x\ y. x \in set\ xs \implies y \in set\ xs \implies x = y) \implies$   
 $(\bigwedge x. x \in set\ xs \implies x = y) \implies$   
 $foldr\ (+)\ xs\ 0 = (length\ xs) * y$   
 ⟨proof⟩

**lemma** *t-build-cnt*:  $T_{build} n \leq cnt\ (vebt-buildup\ n) * 13$   
 ⟨proof⟩

**lemma** *t-buildup-cnt*:  $T_{buildup} n \leq cnt\ (vebt-buildup\ n) * 13$   
 ⟨proof⟩

**lemma** *count-buildup*:  $cnt\ (vebt-buildup\ n) \leq 2 * 2^{\wedge}n$   
 ⟨proof⟩

**lemma** *count-buildup'*:  $cnt\ (vebt-buildup\ n) \leq 2 * (2::nat)^{\wedge}n$   
 ⟨proof⟩

**theorem** *vebt-buildup-bound*:  $u = 2^{\wedge}n \implies T_{buildup} n \leq 26 * u$   
 ⟨proof⟩

Count in natural numbers

**fun** *cnt'*:: *VEBT*  $\Rightarrow$  *nat* **where**  
 $cnt'\ (Leaf\ a\ b) = 1$   
 $cnt'\ (Node\ info\ deg\ treeList\ summary) = 1 + cnt'\ summary + foldr\ (\lambda\ a\ b.\ a+b)\ (map\ cnt'\ treeList)$   
 $0$

**lemma** *cnt-cnt-eq*:  $cnt\ t = cnt'\ t$   
 ⟨proof⟩

**end**  
**end**

## 13 Functional Interface

```

theory VEBT-Intf-Functional
imports Main
         VEBT-Definitions VEBT-Space
         VEBT-Uniqueness
         VEBT-Member
         VEBT-Insert VEBT-InsertCorrectness
         VEBT-MinMax
         VEBT-Pred VEBT-Succ
         VEBT-Bounds
         VEBT-Delete VEBT-DeleteCorrectness VEBT-DeleteBounds

```

```
begin
```

### 13.1 Code Generation Setup

#### 13.1.1 Code Equations

Code generator seems to not support patterns and nat code target

```

context begin
  interpretation VEBT-internal ⟨proof⟩

```

**lemma** *vebt-member-code*[*code*]:

```

vebt-member (Leaf a b) x = (if x = 0 then a else if x=1 then b else False)
vebt-member (Node None t r e) x = False
vebt-member (Node (Some (mi, ma)) deg treeList summary) x =
(if deg = 0 ∨ deg = Suc 0 then False else (
  if x = mi then True else
  if x = ma then True else
  if x < mi then False else
  if x > ma then False else
  (let
    h = high x (deg div 2);
    l = low x (deg div 2) in
    (if h < length treeList
      then vebt-member (treeList ! h) l
      else False))))
  ⟨proof⟩

```

**lemma** *vebt-insert-code*[*code*]:

```

vebt-insert (Leaf a b) x = (if x=0 then Leaf True b else if x=1 then Leaf a True else Leaf a b)
vebt-insert (Node info deg treeList summary) x = (
  if deg ≤ 1 then
    (Node info deg treeList summary)
  else ( case info of
    None ⇒ (Node (Some (x,x)) deg treeList summary)
  | Some mima ⇒ ( case mima of (mi, ma) ⇒ (
    let

```

```

    xn = (if x < mi then mi else x);
    minn = (if x < mi then x else mi);
    l = low xn (deg div 2); h = high xn (deg div 2)
  in (
    if h < length treeList  $\wedge$   $\neg$  (x = mi  $\vee$  x = ma) then
      Node (Some (minn, max xn ma))
        deg
        (treeList[h:= vebt-insert (treeList ! h) l])
        (if minNull (treeList ! h) then vebt-insert summary h else summary)
    else Node (Some (mi, ma)) deg treeList summary)
  ))))
<proof>

```

**lemma** *vebt-succ-code*[code]:

```

vebt-succ (Leaf a b) x = (if b  $\wedge$  x = 0 then Some 1 else None)
vebt-succ (Node info deg treeList summary) x = (if deg  $\leq$  1 then None else
(case info of None  $\Rightarrow$  None |
(Some mima)  $\Rightarrow$  (case mima of (mi, ma)  $\Rightarrow$  (
  if x < mi then (Some mi)
  else (let l = low x (deg div 2); h = high x (deg div 2) in(
    if h < length treeList then
      let maxlow = vebt-maxt (treeList ! h) in
      (if maxlow  $\neq$  None  $\wedge$  (Some l  $<_o$  maxlow) then
        Some (2 $^{\wedge}$ (deg div 2)) * $_o$  Some h + $_o$  vebt-succ (treeList ! h) l
      else let sc = vebt-succ summary h in
        if sc = None then None
        else Some (2 $^{\wedge}$ (deg div 2)) * $_o$  sc + $_o$  vebt-mint (treeList ! the sc) )
    else None))))))

```

<proof>

**lemma** *vebt-pred-code*[code]:

```

vebt-pred (Leaf a b) x = (if x = 0 then None else if x = 1 then
  (if a then Some 0 else None) else
  (if b then Some 1 else if a then Some 0 else None)) and
vebt-pred (Node info deg treeList summary) x = (if deg  $\leq$  1 then None else (
case info of None  $\Rightarrow$  None |
(Some mima)  $\Rightarrow$  (case mima of (mi, ma)  $\Rightarrow$  (
  if x > ma then Some ma
  else (let l = low x (deg div 2); h = high x (deg div 2) in
    if h < length treeList then
      let minlow = vebt-mint (treeList ! h) in
      (if minlow  $\neq$  None  $\wedge$  (Some l  $>_o$  minlow) then
        Some (2 $^{\wedge}$ (deg div 2)) * $_o$  Some h + $_o$  vebt-pred (treeList ! h) l
      else let pr = vebt-pred summary h in
        if pr = None then (if x > mi then Some mi else None)
        else Some (2 $^{\wedge}$ (deg div 2)) * $_o$  pr + $_o$  vebt-maxt (treeList ! the pr) )
    else None))))))

```

<proof>

**lemma** *vebt-delete-code*[code]:

*vebt-delete* (Leaf a b) x = (if x = 0 then Leaf False b else if x = 1 then Leaf a False else Leaf a b)  
*vebt-delete* (Node info deg treeList summary) x = (

case info of

None  $\Rightarrow$  (Node info deg treeList summary)

| Some mima  $\Rightarrow$  (

if deg  $\leq$  1 then (Node info deg treeList summary)

else (case mima of (mi, ma)  $\Rightarrow$  (

if (x < mi  $\vee$  x > ma) then (Node (Some (mi, ma)) deg treeList summary)

else if (x = mi  $\wedge$  x = ma) then (Node None deg treeList summary)

else let

xn = (if x = mi then the (vebt-mint summary) \*  $2^{\wedge}(\text{deg div } 2)$

+ the (vebt-mint (treeList ! the (vebt-mint summary))))

else x);

minn = (if x = mi then xn else mi);

l = low xn (deg div 2);

h = high xn (deg div 2)

in

if h < length treeList then let

newnode = vebt-delete (treeList ! h) l;

newlist = treeList[h:= newnode]

in

if minNull newnode then let

sn = vebt-delete summary h;

maxn =

if xn = ma then let

maxs = vebt-maxt sn

in

if maxs = None then minn

else  $2^{\wedge}(\text{deg div } 2)$  \* the maxs + the (vebt-maxt (newlist ! the maxs))

else ma

in (Node (Some (minn, maxn)) deg newlist sn)

else let

maxn = (if xn = ma then h \*  $2^{\wedge}(\text{deg div } 2)$  + the (vebt-maxt (newlist ! h))

else ma)

in (Node (Some (minn, maxn)) deg newlist summary)

else (Node (Some (mi, ma)) deg treeList summary)

))))

<proof>

**end**

**lemmas** [code] =

*VEBT-internal.high-def VEBT-internal.low-def VEBT-internal.minNull.simps*

*VEBT-internal.less.simps VEBT-internal.mul-def VEBT-internal.add-def*

*VEBT-internal.option-comp-shift.simps VEBT-internal.option-shift.simps*

**export-code**

*vebt-buildup*  
*vebt-insert*  
*vebt-member*  
*vebt-maxt*  
*vebt-mint*  
*vebt-pred*  
*vebt-succ*  
*vebt-delete*  
**checking** *SML*

## 13.2 Correctness Lemmas

**named-theorems** *vebt-simps* *⟨Simplifier rules for VEBT functional interface⟩*

**locale** *vebt-inst* =  
**fixes** *n :: nat*  
**begin**

**interpretation** *VEBT-internal* *⟨proof⟩*

### 13.2.1 Space Bound

**theorem** *vebt-space-linear-bound*:

**fixes** *t*  
**defines**  $u \equiv 2^{\hat{n}}$   
**shows**  $\text{invar-vebt } t \ n \implies \text{space } t \leq 12 * u$   
*⟨proof⟩*

### 13.2.2 Buildup

**lemma** *invar-vebt-buildup[vebt-simps]*:  $\text{invar-vebt } (\text{vebt-buildup } n) \ n \longleftrightarrow n > 0$   
*⟨proof⟩*

**lemma** *set-vebt-buildup[vebt-simps]*:  $\text{set-vebt } (\text{vebt-buildup } i) = \{\}$   
*⟨proof⟩*

**lemma** *time-vebt-buildup*:  $u = 2^{\hat{n}} \implies T_{\text{buildup}} \ n \leq 26 * u$   
*⟨proof⟩*

### 13.2.3 Equality

**lemma** *set-vebt-equal[vebt-simps]*:  $\text{invar-vebt } t_1 \ n \implies \text{invar-vebt } t_2 \ n \implies t_1 = t_2 \longleftrightarrow \text{set-vebt } t_1 = \text{set-vebt } t_2$   
*⟨proof⟩*

### 13.2.4 Member

**lemma** *set-vebt-member[vebt-simps]*:  $\text{invar-vebt } t \ n \implies \text{vebt-member } t \ x \longleftrightarrow x \in \text{set-vebt } t$   
*⟨proof⟩*

**theorem** *time-vebt-member*:  $\text{invar-vebt } t \ n \implies u = 2^{\wedge}n \implies T_{\text{member}} \ t \ x \leq 30 + 15 * \text{lb} (\text{lb } u)$   
 ⟨proof⟩

### 13.2.5 Insert

**theorem** *invar-vebt-insert[vebt-simps]*:  $\text{invar-vebt } t \ n \implies x < 2^{\wedge}n \implies \text{invar-vebt} (\text{vebt-insert } t \ x) \ n$   
 ⟨proof⟩

**theorem** *set-vebt-insert[vebt-simps]*:  $\text{invar-vebt } t \ n \implies x < 2^{\wedge}n \implies \text{set-vebt} (\text{vebt-insert } t \ x) = \text{set-vebt } t \cup \{x\}$   
 ⟨proof⟩

**theorem** *time-vebt-insert*:  $\text{invar-vebt } t \ n \implies u = 2^{\wedge}n \implies T_{\text{insert}} \ t \ x \leq 46 + 23 * \text{lb} (\text{lb } u)$   
 ⟨proof⟩

### 13.2.6 Maximum

**theorem** *set-vebt-maxt*:  $\text{invar-vebt } t \ n \implies \text{vebt-maxt } t = \text{Some } x \longleftrightarrow \text{max-in-set} (\text{set-vebt } t) \ x$   
 ⟨proof⟩

**theorem** *set-vebt-maxt'*:  $\text{invar-vebt } t \ n \implies \text{vebt-maxt } t = \text{Some } x \longleftrightarrow (x \in \text{set-vebt } t \wedge (\forall y \in \text{set-vebt } t. x \geq y))$   
 ⟨proof⟩

**lemma** *set-vebt-maxt''[vebt-simps]*:  
 $\text{invar-vebt } t \ n \implies \text{vebt-maxt } t = (\text{if } \text{set-vebt } t = \{\} \text{ then None else Some } (\text{Max} (\text{set-vebt } t)))$   
 ⟨proof⟩

**lemma** *time-vebt-maxt*:  $T_{\text{maxt}} \ t \leq 3$   
 ⟨proof⟩

### 13.2.7 Minimum

**theorem** *set-vebt-mint[vebt-simps]*:  $\text{invar-vebt } t \ n \implies \text{vebt-mint } t = \text{Some } x \longleftrightarrow \text{min-in-set} (\text{set-vebt } t) \ x$   
 ⟨proof⟩

**theorem** *set-vebt-mint'*:  $\text{invar-vebt } t \ n \implies \text{vebt-mint } t = \text{Some } x \longleftrightarrow (x \in \text{set-vebt } t \wedge (\forall y \in \text{set-vebt } t. x \leq y))$   
 ⟨proof⟩

**lemma** *set-vebt-mint''[vebt-simps]*:  
 $\text{invar-vebt } t \ n \implies \text{vebt-mint } t = (\text{if } \text{set-vebt } t = \{\} \text{ then None else Some } (\text{Min} (\text{set-vebt } t)))$   
 ⟨proof⟩

**lemma** *time-vebt-mint*:  $T_{\text{mint}} \ t \leq 3$   
 ⟨proof⟩

### 13.3 Emptiness determination

A tree is empty if and only if its minimum is None

**lemma** *vebt-minNull-mint*:  $\text{minNull } t \longleftrightarrow \text{vebt-mint } t = \text{None}$   
(proof)

**lemma** *set-vebt-minNull*:  $\text{invar-vebt } t \ n \implies \text{minNull } t \longleftrightarrow \text{set-vebt } t = \{\}$   
(proof)

**lemma** *time-vebt-minNull*:  $T_{\text{minNull}} t \leq 1$   
(proof)

#### 13.3.1 Successor

**theorem** *set-vebt-succ*:  $\text{invar-vebt } t \ n \implies \text{vebt-succ } t \ x = \text{Some } sx \longleftrightarrow \text{is-succ-in-set } (\text{set-vebt } t) \ x \ sx$   
(proof)

**lemma** *set-vebt-succ'[vebt-simps]*:  $\text{invar-vebt } t \ n \implies \text{vebt-succ } t \ x = (\text{if } \exists \ y \in \text{set-vebt } t. \ y > x \ \text{then } \text{Some } (\text{LEAST } y \in \text{set-vebt } t. \ y > x) \ \text{else } \text{None})$   
(proof)

**theorem** *time-vebt-succ*:  
**fixes**  $t$  **defines**  $u \equiv \mathcal{2}^{\hat{n}}$   
**shows**  $\text{invar-vebt } t \ n \implies T_{\text{succ}} t \ x \leq 54 + 27 * \text{lb } (\text{lb } u)$   
(proof)

#### 13.3.2 Predecessor

**theorem** *set-vebt-pred*:  $\text{invar-vebt } t \ n \implies \text{vebt-pred } t \ x = \text{Some } px \longleftrightarrow \text{is-pred-in-set } (\text{set-vebt } t) \ x \ px$   
(proof)

**theorem** *set-vebt-pred'[vebt-simps]*:  $\text{invar-vebt } t \ n \implies \text{vebt-pred } t \ x = (\text{if } \exists \ y \in \text{set-vebt } t. \ y < x \ \text{then } \text{Some } (\text{GREATEST } y. \ y \in \text{set-vebt } t \wedge y < x) \ \text{else } \text{None})$   
(proof)

**theorem** *time-vebt-pred*: **fixes**  $t$  **defines**  $u \equiv \mathcal{2}^{\hat{n}}$   
**shows**  $\text{invar-vebt } t \ n \implies T_{\text{pred}} t \ x \leq 58 + 29 * \text{lb } (\text{lb } u)$   
(proof)

#### 13.3.3 Delete

**theorem** *invar-vebt-delete[vebt-simps]*:  $\text{invar-vebt } t \ n \implies \text{invar-vebt } (\text{vebt-delete } t \ x) \ n$   
(proof)

**theorem** *set-vebt-delete[vebt-simps]*:  $\text{invar-vebt } t \ n \implies \text{set-vebt } (\text{vebt-delete } t \ x) = \text{set-vebt } t - \{x\}$   
(proof)



**theorem** *time-vebt-delete*: **fixes**  $t$  **defines**  $u \equiv 2^{\wedge}n$   
**shows**  $\text{invar-vebt } t \ n \implies T_{\text{delete}} \ t \ x \leq 140 + 70 * \text{lb} \ (\text{lb } u)$   
 $\langle \text{proof} \rangle$

**end**

### 13.4 Interface Usage Example

**experiment**  
**begin**

**definition** *test*  $n \ xs \ ys \equiv \text{let}$   
 $t = \text{vebt-buildup } n;$   
 $t = \text{foldl } \text{vebt-insert } t \ (0\#xs);$

$f = (\lambda x. \text{if } \text{vebt-member } t \ x \ \text{then } x \ \text{else the } (\text{vebt-pred } t \ x))$   
*in*  
 $\text{map } f \ ys$

**context** **fixes**  $n :: \text{nat}$  **begin**  
**interpretation** *vebt-inst*  $n \ \langle \text{proof} \rangle$

**lemmas** [*simp*] = *vebt-simps*

**lemma** [*simp*]:  
**assumes**  $\text{invar-vebt } t \ n \ \forall x \in \text{set } xs. \ x < 2^{\wedge}n$   
**shows**  $\text{invar-vebt } (\text{foldl } \text{vebt-insert } t \ xs) \ n$   
 $\langle \text{proof} \rangle$

**lemma** [*simp*]:  
**assumes**  $\text{invar-vebt } t \ n \ \forall x \in \text{set } xs. \ x < 2^{\wedge}n$   
**shows**  $\text{set-vebt } (\text{foldl } \text{vebt-insert } t \ xs) = \text{set-vebt } t \ \cup \ \text{set } xs$   
 $\langle \text{proof} \rangle$

**lemma**  $\llbracket \forall x \in \text{set } xs. \ x < 2^{\wedge}n; \ n > 0 \rrbracket \implies \text{test } n \ xs \ ys = \text{map } (\lambda y. (\text{GREATEST } y'. \ y' \in \text{insert } 0 \ (\text{set } xs) \wedge y' \leq y)) \ ys$   
 $\langle \text{proof} \rangle$

**end**

**end**

**end**

**theory** *VEBT-List-Assn*

**imports**

*Separation-Logic-Imperative-HOL/Sep-Main*  
*HOL-Library.Rewrite*

**begin**

### 13.5 Lists

**fun** *list-assn* :: ('a  $\Rightarrow$  'c  $\Rightarrow$  *assn*)  $\Rightarrow$  'a *list*  $\Rightarrow$  'c *list*  $\Rightarrow$  *assn* **where**  
  *list-assn* P [] [] = *emp*  
| *list-assn* P (a#as) (c#cs) = P a c \* *list-assn* P as cs  
| *list-assn* - - - = *false*

**lemma** *list-assn-aux-simps*[*simp*]:

*list-assn* P [] l' = ( $\uparrow$ (l'=[]))  
*list-assn* P l [] = ( $\uparrow$ (l=[]))  
*<proof>*

**lemma** *list-assn-aux-append*[*simp*]:

*length* l1 = *length* l1'  $\Longrightarrow$   
  *list-assn* P (l1@l2) (l1'@l2')  
  = *list-assn* P l1 l1' \* *list-assn* P l2 l2'  
*<proof>*

**lemma** *list-assn-aux-ineq-len*: *length* l  $\neq$  *length* li  $\Longrightarrow$  *list-assn* A l li = *false*  
*<proof>*

**lemma** *list-assn-aux-append2*[*simp*]:

**assumes** *length* l2 = *length* l2'  
**shows** *list-assn* P (l1@l2) (l1'@l2')  
  = *list-assn* P l1 l1' \* *list-assn* P l2 l2'  
*<proof>*

**lemma** *list-assn-simps*[*simp*]:

(*list-assn* P) [] [] = *emp*  
(*list-assn* P) (a#as) (c#cs) = P a c \* (*list-assn* P) as cs  
(*list-assn* P) (a#as) [] = *false*  
(*list-assn* P) [] (c#cs) = *false*  
*<proof>*

**lemma** *list-assn-mono*:

$\llbracket \bigwedge x x'. P x x' \Longrightarrow_A P' x x' \rrbracket \Longrightarrow$  *list-assn* P l l'  $\Longrightarrow_A$  *list-assn* P' l l'  
*<proof>*

**lemma** *list-assn-cong*[*fundef-cong*]:

**assumes** *xs*=*xs'* *xi*=*xi'*  
**assumes**  $\bigwedge x xi. x \in \text{set } xs' \Longrightarrow xi \in \text{set } xi' \Longrightarrow A x xi = A' x xi$   
**shows** *list-assn* A *xs xi* = *list-assn* A' *xs' xi'*

*<proof>*

**term** *prod-list*

**definition** *listI-assn*  $I A xs xsi \equiv$   
 $\uparrow(\text{length } xsi = \text{length } xs \wedge I \subseteq \{0..<\text{length } xs\})$   
 $* \text{Finite-Set.fold } (\lambda i a. a * A (xs!i) (xsi!i)) I I$

**lemmas** *comp-fun-commute-fold-insert* =  
*comp-fun-commute-on.fold-insert*[**where**  $S = UNIV$ , *folded comp-fun-commute-def', simplified*]

**lemma** *aux*:  $\text{Finite-Set.fold } (\lambda i aa. aa * P ((a \# as) ! i) ((c \# cs) ! i)) \text{emp } \{0..<\text{Suc } (\text{length } as)\}$   
 $= P a c * \text{Finite-Set.fold } (\lambda i aa. aa * P (as ! i) (cs ! i)) \text{emp } \{0..<\text{length } as\}$   
*<proof>*

**lemma** *list-assn-conv-idx*:  $\text{list-assn } A xs xsi = \text{listI-assn } \{0..<\text{length } xs\} A xs xsi$   
*<proof>*

**lemma** *listI-assn-conv*:  $n = \text{length } xs \implies \text{listI-assn } \{0..<n\} A xs xsi = \text{list-assn } A xs xsi$   
*<proof>*

**lemma** *listI-assn-conv'*:  $n = \text{length } xs \implies \text{listI-assn } \{0..<n\} A xs xsi * F = \text{list-assn } A xs xsi * F$   
*<proof>*

**lemma** *listI-assn-finite[simp]*:  $\neg \text{finite } I \implies \text{listI-assn } I A xs xsi = \text{false}$   
*<proof>*

**find-theorems** *Finite-Set.fold name: cong*

**lemma** *mult-fun-commute*:  $\text{comp-fun-commute } (\lambda i (a::\text{assn}). a * f i)$   
*<proof>*

**lemma** *listI-assn-weak-cong*:  
**assumes**  $I: I = I' \ A = A' \ \text{length } xs = \text{length } xs' \ \text{length } xsi = \text{length } xsi'$   
**assumes**  $A: \bigwedge i. \llbracket i \in I; i < \text{length } xs; \text{length } xs = \text{length } xsi \rrbracket$   
 $\implies xsi = xs!i \wedge xsi!i = xsi!i$   
**shows**  $\text{listI-assn } I A xs xsi = \text{listI-assn } I' A' xs' xsi'$   
*<proof>*

**lemma** *listI-assn-cong*:  
**assumes**  $I: I = I' \ \text{length } xs = \text{length } xs' \ \text{length } xsi = \text{length } xsi'$   
**assumes**  $A: \bigwedge i. \llbracket i \in I; i < \text{length } xs; \text{length } xs = \text{length } xsi \rrbracket$   
 $\implies xsi = xs!i \wedge xsi!i = xsi!i$   
 $\wedge A (xs!i) (xsi!i) = A' (xs!i) (xsi!i)$

**shows**  $listI-assn\ I\ A\ xs\ xsi = listI-assn\ I'\ A'\ xs'\ xsi'$   
 ⟨proof⟩

**lemma** *listI-assn-insert*:  $i \notin I \implies i < length\ xs \implies$   
 $listI-assn\ (insert\ i\ I)\ A\ xs\ xsi = A\ (xs!i)\ (xsi!i) * listI-assn\ I\ A\ xs\ xsi$   
 ⟨proof⟩

**lemma** *listI-assn-extract*:  
**assumes**  $i \in I\ i < length\ xs$   
**shows**  $listI-assn\ I\ A\ xs\ xsi = A\ (xs!i)\ (xsi!i) * listI-assn\ (I - \{i\})\ A\ xs\ xsi$   
 ⟨proof⟩

**lemma** *listI-assn-reinsert*:  
**assumes**  $P \implies_A\ A\ (xs!i)\ (xsi!i) * listI-assn\ (I - \{i\})\ A\ xs\ xsi * F$   
**assumes**  $i < length\ xs\ i \in I$   
**assumes**  $listI-assn\ I\ A\ xs\ xsi * F \implies_A\ Q$   
**shows**  $P \implies_A\ Q$   
 ⟨proof⟩

**lemma** *listI-assn-reinsert-upd*:  
**fixes**  $xs\ xsi :: -\ list$   
**assumes**  $P \implies_A\ A\ x\ xi * listI-assn\ (I - \{i\})\ A\ xs\ xsi * F$   
**assumes**  $i < length\ xs\ i \in I$   
**assumes**  $listI-assn\ I\ A\ (xs[i:=x])\ (xsi[i:=xi]) * F \implies_A\ Q$   
**shows**  $P \implies_A\ Q$   
 ⟨proof⟩

**lemma** *listI-assn-reinsert'!*:  
**assumes**  $P \implies_A\ A\ (xs!i)\ (xsi!i) * listI-assn\ (I - \{i\})\ A\ xs\ xsi * F$   
**assumes**  $i < length\ xs\ i \in I$   
**assumes**  $\langle listI-assn\ I\ A\ xs\ xsi * F \rangle c \langle Q \rangle$   
**shows**  $\langle P \rangle c \langle Q \rangle$   
 ⟨proof⟩

**lemma** *listI-assn-reinsert-upd'!*:  
**fixes**  $xs\ xsi :: -\ list$   
**assumes**  $P \implies_A\ A\ x\ xi * listI-assn\ (I - \{i\})\ A\ xs\ xsi * F$   
**assumes**  $i < length\ xs\ i \in I$   
**assumes**  $\langle listI-assn\ I\ A\ (xs[i:=x])\ (xsi[i:=xi]) * F \rangle c \langle Q \rangle$   
**shows**  $\langle P \rangle c \langle Q \rangle$   
 ⟨proof⟩

**lemma** *subst-not-in*:  
**assumes**  $i \notin I\ i < length\ xs$   
**shows**  $listI-assn\ I\ A\ (xs[i:=xI])\ (xsi[i := x2]) = listI-assn\ I\ A\ xs\ xsi$

*<proof>*

**lemma** *listI-assn-subst*:

**assumes**  $i \notin I \quad i < \text{length } xs$

**shows**  $\text{listI-assn } (\text{insert } i \ I) \ A \ (xs[i:=x1]) \ (xsi[i := x2]) = A \ x1 \ x2 * \text{listI-assn } I \ A \ xs \ xsi$

*<proof>*

**lemma** *extract-pre-list-assn-lengthD*:  $h \models \text{list-assn } A \ xs \ xsi \implies \text{length } xsi = \text{length } xs$

*<proof>*

**method** *unwrap-idx* **for**  $i :: \text{nat} =$

(*rewrite in*  $\langle \sqsupset \rangle \text{-} \langle \text{-} \rangle$  *list-assn-conv-idx*),

(*rewrite in*  $\langle \sqsupset \rangle \text{-} \langle \text{-} \rangle$  *listI-assn-extract*[**where**  $i=i$ ]),

(*simp split*: *if-splits*; *fail*),

(*simp split*: *if-splits*; *fail*)

**method** *wrap-idx* **uses**  $R =$

(*rule*  $R$ ),

*frame-inference*,

(*simp split*: *if-splits*; *fail*),

(*simp split*: *if-splits*; *fail*),

(*subst listI-assn-conv*, (*simp*; *fail*))

**method** *extract-pre-pure* **uses** *dest* =

(*rule hoare-triple-preI* | *drule asm-rl*[*of*  $\text{-} \models \text{-}$ ]),

(*determ*  $\langle \text{elim mod-starE dest}[\text{elim-format}] \rangle$ ),

((*determ*  $\langle \text{thin-tac } \text{-} \models \text{-} \rangle$ )<sup>?</sup>),

(*simp* (*no-asm*) *only*: *triv-forall-equality*)<sup>?</sup>

**lemma** *rule-at-index*:

**assumes**

1:  $P \implies_A \text{list-assn } A \ xs \ xsi * F$  **and**

2[*simp*]:  $i < \text{length } xs$  **and**

3:  $\langle A \ (xs \ ! \ i) \ (xsi \ ! \ i) \ * \$

*listI-assn*  $(\{0..<\text{length } xs\} - \{i\}) \ A \ xs \ xsi * F \rangle \ c \ \langle Q' \rangle$  **and**

4:  $\bigwedge r. Q' \ r \implies_A \ A \ (xs \ ! \ i) \ (xsi \ ! \ i) \ * \$

*listI-assn*  $(\{0..<\text{length } xs\} - \{i\}) \ A \ xs \ xsi * F' \ r$

**shows**

$\langle P \rangle \ c \ \langle \lambda r. \text{list-assn } A \ xs \ xsi * F' \ r \rangle$

*<proof>*

**end**

**theory** *VEBT-BuildupMemImp*

**imports**

*VEBT-List-Assn*

*VEBT-Space*

*Deriving.Derive*

*VEBT-Member VEBT-Insert*  
*HOL-Library.Countable*  
*Time-Reasoning/Time-Reasoning VEBT-DeleteBounds*  
**begin**

## 14 Imperative van Emde Boas Trees

**datatype** *VEBTi* = *Nodei* (nat\*nat) option nat *VEBTi* array *VEBTi* | *Leafi* bool bool

**derive** countable *VEBTi*  
**instance** *VEBTi* :: heap ⟨proof⟩

### 14.1 Assertions on van Emde Boas Trees

**fun** *vebt-assn-raw* :: *VEBT* ⇒ *VEBTi* ⇒ *assn* **where**  
*vebt-assn-raw* (*Leaf* *a b*) (*Leafi* *ai bi*) = ↑(*ai=a* ∧ *bi=b*)  
| *vebt-assn-raw* (*Node* *mmoi deg tree-list summary*) (*Nodei* *mmoi degi tree-array summaryi*) = (  
  ↑(*mmoi=mmo* ∧ *degi=deg*)  
  \* *vebt-assn-raw* *summary summaryi*  
  \* (∃<sub>*A*</sub> *tree-is*. *tree-array* ↦<sub>*a*</sub> *tree-is* \* *list-assn* *vebt-assn-raw* *tree-list* *tree-is*)  
)
| *vebt-assn-raw* - - = false

**lemmas** [*simp del*] = *vebt-assn-raw.simps*

**context** *VEBT-internal* **begin**

**lemmas** [*simp*] = *vebt-assn-raw.simps*

**lemma** *TBOUND-VEBT-case[TBOUND]*: **assumes** ∧ *a b*. *ti = Leafi a b* ⇒ *TBOUND* (*f a b*) (*bnd* *a b*)

  ∧ *info deg treeArray summary* . *ti = Nodei info deg treeArray summary* ⇒  
  *TBOUND* (*f' info deg treeArray summary*) (*bnd' info deg treeArray summary*)

**shows** *TBOUND* (*case ti of Leafi a b* ⇒ *f a b* |  
  *Nodei info deg treeArray summary* ⇒ *f' info deg treeArray summary*)  
  (*case ti of Leafi a b* ⇒ *bnd a b* |  
  *Nodei info deg treeArray summary* ⇒ *bnd' info deg treeArray summary*)  
  ⟨proof⟩

Some Lemmas

**lemma** *length-corresp*:(∃<sub>*A*</sub> *tree-is*. *tree-array* ↦<sub>*a*</sub> *tree-is*) = true ⇒ return (length *tree-is*) = *Array-Time.len* *tree-array*  
  ⟨proof⟩

**lemma** *heaphelp*:**assumes** *h* ⊨  
  *xa* ↦<sub>*a*</sub> *tree-is* \* *list-assn* *vebt-assn-raw* *treeList* *tree-is* \*

$vebt\text{-}assn\text{-}raw\ summary\ xb\ * \uparrow (None = None \wedge n = n) *$   
 $\uparrow (xc = Node\ i\ None\ n\ xa\ xb)$   
**shows**  $h \models vebt\text{-}assn\text{-}raw\ (Node\ None\ n\ treeList\ summary)\ xc$   
 $\langle proof \rangle$

**lemma** *assnle*:  $list\text{-}assn\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is\ * (x13 \mapsto_a\ tree\text{-}is\ * vebt\text{-}assn\text{-}raw\ summary\ x14)$   
 $\implies_A$   
 $vebt\text{-}assn\text{-}raw\ summary\ x14\ * x13 \mapsto_a\ tree\text{-}is\ * list\text{-}assn\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is$   
 $\langle proof \rangle$

**lemma** *ext*:  $y < length\ treeList \implies x13 \mapsto_a\ tree\text{-}is\ * (vebt\text{-}assn\text{-}raw\ summary\ x14\ * (vebt\text{-}assn\text{-}raw\ (treeList\ !\ y)\ (tree\text{-}is\ !\ y)\ * listI\text{-}assn\ (\{0..<length\ treeList\} - \{y\})\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is))$   
 $\implies_A (x13 \mapsto_a\ tree\text{-}is\ * vebt\text{-}assn\text{-}raw\ summary\ x14\ * (listI\text{-}assn\ (\{0..<length\ treeList\} - \{y\})\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is)) * vebt\text{-}assn\text{-}raw\ (treeList\ !\ y)\ (tree\text{-}is\ !\ y)$   
 $\langle proof \rangle$

**lemma** *txe*:  $y < length\ treeList \implies vebt\text{-}assn\text{-}raw\ (treeList\ !\ y)\ (tree\text{-}is\ !\ y)\ * x13 \mapsto_a\ tree\text{-}is\ * vebt\text{-}assn\text{-}raw\ summary\ x14\ * listI\text{-}assn\ (\{0..<length\ treeList\} - \{y\})\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is \implies_A vebt\text{-}assn\text{-}raw\ summary\ x14\ * x13 \mapsto_a\ tree\text{-}is\ * list\text{-}assn\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is$   
 $\langle proof \rangle$

**lemma** *recomp*:  $i < length\ treeList \implies vebt\text{-}assn\text{-}raw\ (treeList\ !\ i)\ (tree\text{-}is\ !\ i)\ * listI\text{-}assn\ (\{0..<length\ treeList\} - \{i\})\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is\ * x13 \mapsto_a\ tree\text{-}is\ * vebt\text{-}assn\text{-}raw\ summary\ x14 \implies_A vebt\text{-}assn\text{-}raw\ summary\ x14\ * x13 \mapsto_a\ tree\text{-}is\ * list\text{-}assn\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is$   
 $\langle proof \rangle$

**lemma** *repack*:  $i < length\ treeList \implies vebt\text{-}assn\text{-}raw\ (treeList\ !\ i)\ (tree\text{-}is\ !\ i)\ * Rest\ * (x13 \mapsto_a\ tree\text{-}is\ * vebt\text{-}assn\text{-}raw\ summary\ x14\ * listI\text{-}assn\ (\{0..<length\ treeList\} - \{i\})\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is) \implies_A Rest\ * vebt\text{-}assn\text{-}raw\ summary\ x14\ * x13 \mapsto_a\ tree\text{-}is\ * list\text{-}assn\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is$   
 $\langle proof \rangle$

**lemma** *big-assn-simp*:  $h < length\ treeList \implies vebt\text{-}assn\text{-}raw\ (vebt\text{-}delete\ (treeList\ !\ h)\ l)\ x\ * \uparrow (xaa = vebt\text{-}mint\ (vebt\text{-}delete\ (treeList\ !\ h)\ l))\ * (x13 \mapsto_a\ (tree\text{-}is\ [h := x])\ * vebt\text{-}assn\text{-}raw\ summary\ x14\ * listI\text{-}assn\ (\{0..<length\ treeList\} - \{h\})\ vebt\text{-}assn\text{-}raw\ treeList\ tree\text{-}is) \implies_A x13 \mapsto_a\ tree\text{-}is\ [h := x]\ * vebt\text{-}assn\text{-}raw\ summary\ x14\ * \uparrow (xaa = vebt\text{-}mint\ (vebt\text{-}delete\ (treeList\ !\ h)\ l))\ * list\text{-}assn\ vebt\text{-}assn\text{-}raw\ (treeList\ [h := (vebt\text{-}delete\ (treeList\ !\ h)\ l)])\ (tree\text{-}is\ [h := x])$

*<proof>*

**lemma** *tcd*:  $i < \text{length } \text{treeList} \implies \text{length } \text{treeList} = \text{length } \text{treeList}' \implies$   
 $\text{vebt-assn-raw } y \ x \ * \ x13 \mapsto_a \ \text{tree-is}[i:=x] \ * \ \text{vebt-assn-raw } \text{summary } x14 \ * \ \text{listI-assn } (\{0..<\text{length } \text{treeList}\} - \{i\}) \ \text{vebt-assn-raw } (\text{treeList}[i:=y]) \ (\text{tree-is}[i:=x])$   
 $\implies_A \ x13 \mapsto_a \ \text{tree-is}[i:=x] \ * \ \text{vebt-assn-raw } \text{summary } x14 \ * \ \text{list-assn } \text{vebt-assn-raw } (\text{treeList}[i:=y])$   
 $(\text{tree-is}[i:=x])$   
*<proof>*

**lemma** *big-assn-simp'*:  $h < \text{length } \text{treeList} \implies \text{xaa} = \text{vebt-delete } (\text{treeList} ! h) l \implies$   
 $\text{vebt-assn-raw } \text{xaa} \ x \ * \ \uparrow (xb = \text{vebt-mint } \text{xaa}) \ *$   
 $(x13 \mapsto_a \ \text{tree-is}[h:=x] \ * \ \text{vebt-assn-raw } \text{summary } x14 \ *$   
 $\text{listI-assn } (\{0..<\text{length } \text{treeList}\} - \{h\}) \ \text{vebt-assn-raw } \text{treeList } \text{tree-is}) \implies_A$   
 $(x13 \mapsto_a \ \text{tree-is}[h:=x] \ * \ \text{vebt-assn-raw } \text{summary } x14 \ * \ \uparrow (xb = \text{vebt-mint } \text{xaa}) \ *$   
 $\text{list-assn } \text{vebt-assn-raw } (\text{treeList}[h:=\text{xaa}]) \ (\text{tree-is}[h:=x]))$   
*<proof>*

**lemma** *refines-case-VEBTi[refines-rule]*: **assumes**  $ti = ti' \wedge a \ b$ . *refines*  $(f1 \ a \ b) (f1' \ a \ b)$   
 $\wedge \ \text{info } \text{deg } \text{treeArray } \text{summary} . \ \text{refines } (f2 \ \text{info } \text{deg } \text{treeArray } \text{summary}) (f2' \ \text{info } \text{deg } \text{treeArray } \text{summary})$   
**shows** *refines*  $(\text{case } ti \ \text{of } \text{Leafi } a \ b \Rightarrow f1 \ a \ b \mid$   
 $\text{Nodei } \text{info } \text{deg } \text{treeArray } \text{summary} \Rightarrow f2 \ \text{info } \text{deg } \text{treeArray } \text{summary})$   
 $(\text{case } ti' \ \text{of } \text{Leafi } a \ b \Rightarrow f1' \ a \ b \mid$   
 $\text{Nodei } \text{info } \text{deg } \text{treeArray } \text{summary} \Rightarrow f2' \ \text{info } \text{deg } \text{treeArray } \text{summary})$   
*<proof>*

## 14.2 High and low Bitsequences Definition

**definition** *highi*:: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **Heap where**  
 $\text{highi } x \ n == \text{return } (x \ \text{div } (2^n))$

**definition** *lowi*:: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **Heap where**  
 $\text{lowi } x \ n == \text{return } (x \ \text{mod } (2^n))$

**lemma** *highi-h*:  $\langle \text{emp} \rangle \ \text{highi } x \ n \ < \lambda \ r. \ \uparrow (r = \text{high } x \ n) \rangle$   
*<proof>*

**lemma** *highi-hT*:  $\langle \text{emp} \rangle \ \text{highi } x \ n \ < \lambda \ r. \ \uparrow (r = \text{high } x \ n) \rangle T[1]$   
*<proof>*

**lemma** *lowi-h*:  $\langle \text{emp} \rangle \ \text{lowi } x \ n \ < \lambda \ r. \ \uparrow (r = \text{low } x \ n) \rangle$   
*<proof>*

**lemma** *lowi-hT*:  $\langle \text{emp} \rangle \ \text{lowi } x \ n \ < \lambda \ r. \ \uparrow (r = \text{low } x \ n) \rangle T[1]$   
*<proof>*



## 15 Imperative Implementation of *vebt* – *buildup*

```
fun replicatei::nat ⇒ 'a Heap ⇒ ('a list) Heap where
  replicatei 0 x = return []
  replicatei (Suc n) x = do{ y <- x;
                           ys <- replicatei n x;
                           return (y#ys) }
```

**lemma** *time-replicate*:  $\llbracket \wedge h. \text{time } x \ h \leq c \rrbracket \implies \text{time } (\text{replicatei } n \ x) \ h \leq (1+(1+c)*n)$   
 ⟨proof⟩

**lemma** *TBOUND-replicate*:  $\llbracket TBOUND \ x \ c \rrbracket \implies TBOUND (\text{replicatei } n \ x) (1+(1+c)*n)$   
 ⟨proof⟩

**lemma** *refines-replicate*[*refines-rule*]:  
*refines* f f'  $\implies \text{refines } (\text{replicatei } n \ f) (\text{replicatei } n \ f')$   
 ⟨proof⟩

```
fun vebt-buildupi'::nat ⇒ VEBTi Heap where
  vebt-buildupi' 0 = return (Leafi False False)|
  vebt-buildupi' (Suc 0) = return (Leafi False False)|
  vebt-buildupi' n = (if even n then (let half = n div 2 in do{
    treeList <- replicatei (2half) (vebt-buildupi' half);
    assert' (length treeList = 2half);
    trees <- Array-Time.of-list treeList;
    summary <- (vebt-buildupi' half);
    return (Nodei None n trees summary)})
  else (let half = n div 2 in do{
    treeList <- replicatei (2(Suc half)) (vebt-buildupi' half);
    assert' (length treeList = 2Suc half);
    trees <- Array-Time.of-list treeList;
    summary <- (vebt-buildupi' (Suc half));
    return (Nodei None n trees summary)} ))
```

**end**

**context begin**  
**interpretation** *VEBT-internal* ⟨proof⟩

```
fun vebt-buildupi::nat ⇒ VEBTi Heap where
  vebt-buildupi 0 = return (Leafi False False)|
  vebt-buildupi (Suc 0) = return (Leafi False False)|
  vebt-buildupi n = (if even n then (let half = n div 2 in do{
    treeList <- replicatei (2half) (vebt-buildupi half);
    trees <- Array-Time.of-list treeList;
    summary <- (vebt-buildupi half);
    return (Nodei None n trees summary)})
  else (let half = n div 2 in do{
    treeList <- replicatei (2(Suc half)) (vebt-buildupi half);
```

```

trees <- Array-Time.of-list treeList;
summary <- (vebt-buildupi (Suc half));
return (Nodei None n trees summary) } )

```

**end**

**context** *VEBT-internal* **begin**

**lemma** *vebt-buildupi-refines*: *refines (vebt-buildupi n) (vebt-buildupi' n)*  
 <proof>

**fun** *T-vebt-buildupi* **where**

```

  T-vebt-buildupi 0 = Suc 0
| T-vebt-buildupi (Suc 0) = Suc 0
| T-vebt-buildupi (Suc (Suc n)) = (
  if even n then
    Suc (Suc (Suc (T-vebt-buildupi (Suc (n div 2)) +
      (4 * 2 ^ (n div 2) + 2 * (T-vebt-buildupi (Suc (n div 2)) * 2 ^ (n div 2))))))
  else
    Suc (Suc (Suc (T-vebt-buildupi (Suc (Suc (n div 2))) +
      (8 * 2 ^ (n div 2) + 4 * (T-vebt-buildupi (Suc (n div 2)) * 2 ^ (n div 2))))))

```

**lemma** *TBOUND-vebt-buildupi*:

**defines** *foo*  $\equiv$  *T-vebt-buildupi*

**shows** *TBOUND (vebt-buildupi' n) (foo n)*

<proof>

**lemma** *T-vebt-buildupi: time (vebt-buildupi' n) h*  $\leq$  *T-vebt-buildupi n*

<proof>

**lemma** *repli-cons-repl*:  $\langle Q \rangle x \langle \lambda r. Q * A y r \rangle \implies \langle Q \rangle \text{replicatei } n x \langle \lambda r. Q * \text{list-assn } A$   
*(replicate n y) r >*

<proof>

**corollary** *repli-emp*:  $\langle \text{emp} \rangle x \langle \lambda r. A y r \rangle \implies \langle \text{emp} \rangle \text{replicatei } n x \langle \lambda r. \text{list-assn } A$   
*(replicate n y) r >*

<proof>

**lemma** *builupi'corr*:  $\langle \text{emp} \rangle \text{vebt-buildupi' } n \langle \lambda r. \text{vebt-assn-raw (vebt-buildup } n) r \rangle$

<proof>

**lemma** *htt-vebt-buildupi'*:  $\langle \text{emp} \rangle (\text{vebt-buildupi' } n) \langle \lambda r. \text{vebt-assn-raw (vebt-buildup } n) r \rangle T$   
 [*T-vebt-buildupi n*]

<proof>

**lemma** *builupicorr*:  $\langle \text{emp} \rangle \text{vebt-buildupi } n \langle \lambda r. \text{vebt-assn-raw (vebt-buildup } n) r \rangle$

<proof>

**lemma** *htt-vebt-buildupi*:  $\langle emp \rangle (vebt-buildupi\ n) \langle \lambda r. vebt-assn-raw (vebt-buildup\ n)\ r \rangle T [T-vebt-buildupi\ n]$

*<proof>*

Closed bound for  $T - vebt - buildupi$

Amortization

**lemma** *T-vebt-buildupi-gg-0*:  $T-vebt-buildupi\ n > 0$

*<proof>*

**fun** *T-vebt-buildupi'*:: $nat \Rightarrow int$  **where**

*T-vebt-buildupi'* 0 = 1

| *T-vebt-buildupi'* (Suc 0) = 1

| *T-vebt-buildupi'* (Suc (Suc n)) = (

  if even n then

    3 + (*T-vebt-buildupi'* (Suc (n div 2)) +  
      (4 \* 2<sup>(n div 2)</sup> + 2 \* (*T-vebt-buildupi'* (Suc (n div 2)) \* 2<sup>(n div 2))))</sup>

  else

    3 + (*T-vebt-buildupi'* (Suc (Suc (n div 2))) +  
      (8 \* 2<sup>(n div 2)</sup> + 4 \* (*T-vebt-buildupi'* (Suc (n div 2)) \* 2<sup>(n div 2))))</sup>

**lemma** *Tbuildupi-buildupi'*:  $T-vebt-buildupi\ n = T-vebt-buildupi'\ n$

*<proof>*

**fun** *Tb*:: $nat \Rightarrow int$  **where**

*Tb* 0 = 3

| *Tb* (Suc 0) = 3

| *Tb* (Suc (Suc n)) = (

  if even n then

    5 + *Tb* (Suc (n div 2)) + (*Tb* (Suc (n div 2))) \* 2<sup>(Suc (n div 2))</sup>

  else

    5 + *Tb* (Suc (Suc (n div 2))) + (*Tb* (Suc (n div 2))) \* 2<sup>(Suc (Suc (n div 2)))</sup>

**lemma** *Tb-T-vebt-buildupi'*:  $T-vebt-buildupi'\ n \leq Tb\ n - 2$

*<proof>*

**fun** *Tb'*:: $nat \Rightarrow nat$  **where**

*Tb'* 0 = 3

| *Tb'* (Suc 0) = 3

| *Tb'* (Suc (Suc n)) = (

  if even n then

    5 + *Tb'* (Suc (n div 2)) + (*Tb'* (Suc (n div 2))) \* 2<sup>(Suc (n div 2))</sup>

  else

    5 + *Tb'* (Suc (Suc (n div 2))) + (*Tb'* (Suc (n div 2))) \* 2<sup>(Suc (Suc (n div 2)))</sup>

**lemma** *Tb-Tb'*:  $Tb\ t = Tb'\ t$

*<proof>*

**lemma** *Tb-T-vebt-buildupi*:  $T-vebt-buildupi\ n \leq Tb\ n - 2$

*<proof>*

**lemma** *Tb-T-vebt-buildupi''*:  $T\text{-vebt-buildupi } n \leq Tb' n - 2$   
 ⟨proof⟩

**lemma** *Tb'-cnt*:  $Tb' n \leq 5 * cnt' (vebt\text{-buildup } n)$   
 ⟨proof⟩

**lemma** *T-vebt-buildupi-cnt'*:  $T\text{-vebt-buildupi } n \leq 5 * cnt (vebt\text{-buildup } n)$   
 ⟨proof⟩

**lemma** *T-vebt-buildupi-univ*:  
**assumes**  $u = 2^{\wedge}n$   
**shows**  $T\text{-vebt-buildupi } n \leq 10 * u$   
 ⟨proof⟩

**lemma** *htt-vebt-buildupi'-univ*:  
**assumes**  $u = 2^{\wedge}n$   
**shows**  
 $\langle emp \rangle (vebt\text{-buildupi}' n) \langle \lambda r. vebt\text{-assn-raw } (vebt\text{-buildup } n) r \rangle T [10 * u]$   
 ⟨proof⟩

We obtain the main theorem for *buildupi*

**lemma** *htt-vebt-buildupi-univ*:  
**assumes**  $u = 2^{\wedge}n$   
**shows**  
 $\langle emp \rangle (vebt\text{-buildupi } n) \langle \lambda r. vebt\text{-assn-raw } (vebt\text{-buildup } n) r \rangle T [10 * u]$   
 ⟨proof⟩

**lemma** *vebt-buildupi-rule*:  $\langle \uparrow (n > 0) \rangle vebt\text{-buildupi } n \langle \lambda r. vebt\text{-assn-raw } (vebt\text{-buildup } n) r \rangle T[10 * 2^{\wedge}n]$   
 ⟨proof⟩

**lemma** *TBOUND-buildupi*: **assumes**  $n > 0$  **shows**  $TBOUND (vebt\text{-buildupi } n) (10 * 2^{\wedge}n)$   
 ⟨proof⟩

## 16 Minimum and Maximum Determination

end

context begin

interpretation *VEBT-internal* ⟨proof⟩

**fun** *vebt-minti*::  $VEBTi \Rightarrow nat\ option\ Heap$  **where**  
 $vebt\text{-minti } (Leafi\ a\ b) = (if\ a\ then\ return\ (Some\ 0)\ else\ if\ b\ then\ return\ (Some\ 1)\ else\ return\ None)$   
 $vebt\text{-minti } (Nodei\ None\ -\ -) = return\ None$   
 $vebt\text{-minti } (Nodei\ (Some\ (mi,ma))\ -\ -) = return\ (Some\ mi)$

**fun** *vebt-maxti*::  $VEBTi \Rightarrow nat\ option\ Heap$  **where**

$vebt-maxti$  (*Leaf*  $a$   $b$ ) = (if  $b$  then return (*Some* 1) else if  $a$  then return (*Some* 0) else return *None*)|  
 $vebt-maxti$  (*Node*  $i$  *None* - - -) = return *None*|  
 $vebt-maxti$  (*Node*  $i$  (*Some* ( $mi, ma$ )) - - -) = return (*Some*  $ma$ )

**end**

**context** *VEBT-internal* **begin**

**lemma** *vebt-minti-h*: $\langle vebt-assn-raw$   $t$   $ti \rangle$  *vebt-minti*  $ti$   $\langle \lambda r. vebt-assn-raw$   $t$   $ti$   $*$   $\uparrow(r = vebt-mint$   $t) \rangle$

$\langle proof \rangle$

**lemma** *vebt-maxti-h*: $\langle vebt-assn-raw$   $t$   $ti \rangle$  *vebt-maxti*  $ti$   $\langle \lambda r. vebt-assn-raw$   $t$   $ti$   $*$   $\uparrow(r = vebt-maxt$   $t) \rangle$

$\langle proof \rangle$

**lemma** *TBOUND-vebt-maxti*[*TBOUND*]: *TBOUND* (*vebt-maxti*  $t$ ) 1

$\langle proof \rangle$

**lemma** *TBOUND-vebt-minti*[*TBOUND*]: *TBOUND* (*vebt-minti*  $t$ ) 1

$\langle proof \rangle$

**lemma** *vebt-minti-hT*: $\langle vebt-assn-raw$   $t$   $ti \rangle$  *vebt-minti*  $ti$   $\langle \lambda r. vebt-assn-raw$   $t$   $ti$   $*$   $\uparrow(r = vebt-mint$   $t) \rangle T[1]$

$\langle proof \rangle$

**lemma** *vebt-maxti-hT*: $\langle vebt-assn-raw$   $t$   $ti \rangle$  *vebt-maxti*  $ti$   $\langle \lambda r. vebt-assn-raw$   $t$   $ti$   $*$   $\uparrow(r = vebt-maxt$   $t) \rangle T[1]$

$\langle proof \rangle$

**lemma** *vebt-maxtilist*: $i < length$   $ts \implies$

$\langle list-assn$  *vebt-assn-raw*  $ts$   $tsi \rangle$  *vebt-maxti* ( $tsi$  !  $i$ )  
 $\langle \lambda r. \uparrow(r = vebt-maxt$  ( $ts$  !  $i$ ))  $*$  *list-assn* *vebt-assn-raw*  $ts$   $tsi \rangle$

$\langle proof \rangle$

**lemma** *vebt-mintilist*: $i < length$   $ts \implies$

$\langle list-assn$  *vebt-assn-raw*  $ts$   $tsi \rangle$  *vebt-minti* ( $tsi$  !  $i$ )  
 $\langle \lambda r. \uparrow(r = vebt-mint$  ( $ts$  !  $i$ ))  $*$  *list-assn* *vebt-assn-raw*  $ts$   $tsi \rangle$

$\langle proof \rangle$

## 17 Membership Test on imperative van Emde Boas Trees

**end**

**context** **begin**

**interpretation** *VEBT-internal*  $\langle proof \rangle$

**partial-function** (*heap-time*) *vebt-memberi*::*VEBT*  $i \Rightarrow nat \Rightarrow bool$  *Heap* **where**

*vebt-memberi*  $t$   $x =$

(*case*  $t$  *of*

```

(Leafi a b ) ⇒ return (if x = 0 then a else if x=1 then b else False) |
(Nodei info deg treeList summary ) ⇒ (
  case info of None ⇒ return False |
  (Some (mi, ma)) ⇒ ( if deg ≤ 1 then return False else (
    if x = mi then return True else
    if x = ma then return True else
    if x < mi then return False else
    if x > ma then return False else
    (do {
      h ← highi x (deg div 2);
      l ← lowi x (deg div 2);
      len ← Array-Time.len treeList;
      if h < len then do {
        th ← Array-Time.nth treeList h;
        vebt-memberi th l
      } else return False
    }))))))

```

**end**

**context** *VEBT-internal* **begin**

**partial-function** (*heap-time*) *vebt-memberi'*::*VEBT* ⇒ *VEBT**i* ⇒ *nat* ⇒ *bool* **Heap** **where**

*vebt-memberi'* *t* *ti* *x* =

```

(case ti of
  (Leafi a b ) ⇒ return (if x = 0 then a else if x=1 then b else False) |
  (Nodei info deg treeArray summary ) ⇒ ( do {assert' (is-Node t);
    case info of None ⇒ return False |
    (Some (mi, ma)) ⇒ ( if deg ≤ 1 then return False else (
      if x = mi then return True else
      if x = ma then return True else
      if x < mi then return False else
      if x > ma then return False else
      (do {
        let (info',deg',treeList,summary') =
          (case t of (Node info' deg' treeList summary') ⇒
            (info', deg', treeList, summary'));
        assert'(info= info' ∧ deg = deg');
        h ← highi x (deg div 2);
        l ← lowi x (deg div 2);
        assert'(l = low x (deg div 2) ∧ h = high x (deg div 2));
        len ← Array-Time.len treeArray;
        assert'(len = length treeList);
        if h < len then do {
          assert'(h = high x (deg div 2) ∧ h < length treeList);

          th ← Array-Time.nth treeArray h;
          vebt-memberi' (treeList ! h) th l }
        else return False
      }
    )
  }
)

```

}})))))

**lemma** *highsimp*: return (high x n) = highi x n  
(proof)

**lemma** *lowsimp*: return (low x n) = lowi x n  
(proof)

**lemma** *TBOUND-highi*[TBOUND]: TBOUND (highi x n) 1  
(proof)

**lemma** *TBOUND-lowi*[TBOUND]: TBOUND (lowi x n) 1  
(proof)

Correctness of *vebt* – *memberi*

**lemma** *vebt-memberi'-rf-abstr*: <vebt-assn-raw t ti> vebt-memberi' t ti x < $\lambda r$ . vebt-assn-raw t ti \*  
 $\uparrow(r = \text{vebt-member } t x)$ >  
(proof)

**lemma** *TBOUND-vebt-memberi*:  
**defines** *foo-def*:  $\bigwedge t x. \text{foo } t x \equiv 4 * (1 + \text{height } t)$   
**shows** TBOUND (vebt-memberi' t ti x) (foo t x)  
(proof)

**lemma** *vebt-memberi-refines*: refines (vebt-memberi ti x) (vebt-memberi' t ti x)  
(proof)

**lemma** *htt-vebt-memberi*:  
<vebt-assn-raw t ti>vebt-memberi ti x < $\lambda r$ . vebt-assn-raw t ti \*  $\uparrow(r = \text{vebt-member } t x)$ >T[ 5 +  
5 \* height t ]  
(proof)

**lemma** *htt-vebt-memberi-invar-vebt*: **assumes** *invar-vebt* t n **shows**  
<vebt-assn-raw t ti> vebt-memberi ti x < $\lambda r$ . vebt-assn-raw t ti \*  $\uparrow(r = \text{vebt-member } t x)$ >T[5 +  
5 \* (nat [lb n ])]  
(proof)

## 17.1 minNulli: empty tree?

**fun** *minNulli*:: VEBTi  $\Rightarrow$  bool Heap **where**  
minNulli (Leafi False False) = return True|  
minNulli (Leafi - -) = return False|  
minNulli (Nodei None - -) = return True|  
minNulli (Nodei (Some -) - -) = return False

**lemma** *minNulli-rule*[sep-heap-rules]: <vebt-assn-raw t ti> minNulli ti < $\lambda r$ . vebt-assn-raw t ti \*  $\uparrow(r = \text{minNull } t)$ >  
(proof)

**lemma** *TBOUND-minNulli*[TBOUND]: TBOUND (minNulli t) 1

*<proof>*

**lemma** *minNulli-ruleT*:

*<vebt-assn-raw t ti> minNulli ti < $\lambda r. vebt-assn-raw t ti * \uparrow(r = minNull t)$ > T[I]*  
*<proof>*

## 18 Imperative *vebt – insert* to van Emde Boas Tree

**end**

**context begin**

**interpretation** *VEBT-internal* *<proof>*

**partial-function** (*heap-time*) *vebt-inserti::VEBTi  $\Rightarrow$  nat  $\Rightarrow$  VEBTi Heap* **where**

*vebt-inserti t x = (case t of*

*(Leafi a b)  $\Rightarrow$  (if x=0 then return (Leafi True b) else if x=1*  
*then return (Leafi a True) else return (Leafi a b)) |*

*(Nodei info deg treeArray summary)  $\Rightarrow$  ( case info of None  $\Rightarrow$*

*if deg  $\leq$  1 then*

*return (Nodei info deg treeArray summary)*

*else*

*return (Nodei (Some (x,x)) deg treeArray summary)|*

*(Some minma)  $\Rightarrow$*

*( if deg  $\leq$  1*

*then return (Nodei info deg treeArray summary)*

*else ( do{*

*mi  $\leftarrow$  return (fst minma);*

*ma  $\leftarrow$  return (snd minma);*

*xn  $\leftarrow$  (if x < mi then return mi else return x);*

*minn  $\leftarrow$  (if x < mi then return x else return mi);*

*l  $\leftarrow$  lowi xn (deg div 2);*

*h  $\leftarrow$  highi xn (deg div 2);*

*len  $\leftarrow$  Array-Time.len treeArray;*

*if h < len  $\wedge$   $\neg$  (x = mi  $\vee$  x = ma) then do {*

*node  $\leftarrow$  Array-Time.nth treeArray h;*

*empt  $\leftarrow$  minNulli node;*

*newnode  $\leftarrow$  vebt-inserti node l;*

*newarray  $\leftarrow$  Array-Time.upd h newnode treeArray;*

*newsummary  $\leftarrow$  (if empt then*

*vebt-inserti summary h*

*else return summary);*

*man  $\leftarrow$  (if xn > ma then return xn else return ma);*

*return (Nodei (Some (minn, man)) deg newarray*

*newsummary))}*

*else return (Nodei (Some (mi,ma)) deg treeArray summary)*

*}}))*

**end**



**context** *VEBT-internal* **begin**

**partial-function** (*heap-time*) *vebt-inserti'*::*VEBT*  $\Rightarrow$  *VEBTi*  $\Rightarrow$  *nat*  $\Rightarrow$  *VEBTi Heap* **where**

```

vebt-inserti' t ti x = (case ti of
  (Leafi a b)  $\Rightarrow$  (if x=0 then return (Leafi True b) else if x=1
    then return (Leafi a True) else return (Leafi a b)) |
  (Nodei info deg treeArray summary)  $\Rightarrow$  ( case info of None  $\Rightarrow$ 
    if deg  $\leq$  1 then
      return (Nodei info deg treeArray summary)
    else
      return (Nodei (Some (x,x)) deg treeArray summary)|
  (Some minma)  $\Rightarrow$ 
    ( if deg  $\leq$  1
      then return (Nodei info deg treeArray summary)
      else (
do{
  assert' (is-Node t);
  let (info',deg',treeList,summary') =
    (case t of (Node info' deg' treeList summary')  $\Rightarrow$ 
      (info', deg', treeList, summary'));
  assert'(info= info'  $\wedge$  deg = deg');
  let (mi', ma') = (the info');
  mi  $\leftarrow$  return (fst minma);
  ma  $\leftarrow$  return (snd minma);
  xn  $\leftarrow$  (if x < mi then return mi else return x);
  let xn' = (if x < mi' then mi' else x);
  minn  $\leftarrow$  (if x < mi then return x else return mi);
  let minn' = (if x < mi' then x else mi');
  l  $\leftarrow$  lowi xn (deg div 2);
  assert' (l = low xn' (deg' div 2));
  h  $\leftarrow$  highi xn (deg div 2);
  len  $\leftarrow$  Array-Time.len treeArray;
  if h < len  $\wedge$   $\neg$  (x = mi  $\vee$  x = ma) then do {
    assert' (h = high xn' (deg' div 2));
    assert' ( h < length treeList);
    node  $\leftarrow$  Array-Time.nth treeArray h;
    empt  $\leftarrow$  minNulli node;
    assert' (empt = minNull (treeList ! h));
    newnode  $\leftarrow$  vebt-inserti' (treeList ! h) node l;
    newarray  $\leftarrow$  Array-Time.upd h newnode treeArray;
    newsummary  $\leftarrow$  (if empt then
      vebt-inserti' summary' summary h
      else return summary);
    man  $\leftarrow$  (if xn > ma then return xn else return ma);
    return (Nodei (Some (minn, man)) deg newarray
  newsummary)}}
else return (Nodei (Some (mi,ma)) deg treeArray summary)
}}))

```

```

lemmas listI-assn-wrap-insert = listI-assn-reinsert-upd'[
  where x=VEBT-Insert.vebt-insert - - and A=vebt-assn-raw ]

lemma vebt-inserti'-rf-abstr: <vebt-assn-raw t ti> vebt-inserti' t ti x <λr. vebt-assn-raw ( vebt-insert
t x) r >
<proof>

lemma TBOUND-minNull: minNull t ⇒ TBOUND (vebt-inserti' t ti x) 1
<proof>

lemma TBOUND-vebt-inserti:
  defines foo-def: ∧ t x. foo t x ≡ if minNull t then 1 else 13 * (1+height t)
  shows TBOUND (vebt-inserti' t ti x) (foo t x)
<proof>

lemma vebt-inserti-refines: refines (vebt-inserti ti x) (vebt-inserti' t ti x)
<proof>

lemma htt-vebt-inserti:
  <vebt-assn-raw t ti> vebt-inserti ti x <λ r. vebt-assn-raw (vebt-insert t x) r> T[ 13 + 13 * height t]
<proof>

lemma htt-vebt-inserti-invar-vebt: assumes invar-vebt t n shows
  <vebt-assn-raw t ti> vebt-inserti ti x <λ r. vebt-assn-raw (vebt-insert t x) r> T[13 + 13 * (nat [lb
n ])]
<proof>

end
end

theory VEBT-SuccPredImperative
  imports VEBT-BuildupMemImp VEBT-Succ VEBT-Pred
begin

context begin
interpretation VEBT-internal <proof>

```

## 19 Imperative Successor

```

partial-function (heap-time) vebt-succi:: VEBTi ⇒ nat ⇒ (nat option) Heap where
  vebt-succi t x = (case t of (Leafi a b) ⇒ (if x = 0 then (if b then return (Some 1) else return None)
    else return None) |
    (Nodei info deg treeArray summary) ⇒ (
      case info of None ⇒ return None |
      (Some mima) ⇒ ( if deg ≤ 1 then return None else
        (if x < fst mima then return (Some (fst mima)) else
          if x ≥ snd mima then return None else
          do {

```

```

    l ← lowi x (deg div 2);
    h ← highi x (deg div 2);
    aktnode ← Array-Time.nth treeArray h;
    maxlow ← vebt-maxti aktnode;
    if (maxlow ≠ None ∧ (Some l <_o maxlow))
    then do {
        succy ← vebt-succi aktnode l;
        return (Some (2deg div 2) *_o Some h +_o succy)
    }
    else do {
        succsum ← vebt-succi summary h;
        if succsum = None then
            return None
        else
            do{
                nextnode ← Array-Time.nth treeArray (the succsum);
                minnext ← vebt-minti nextnode;
                return (Some (2deg div 2) *_o succsum +_o minnext)
            }
    }
})
)))

```

**end**

**context** *VEBT-internal* **begin**

**partial-function** (*heap-time*) *vebt-succi'*::*VEBT* ⇒ *VEBT**i* ⇒ *nat* ⇒ (*nat option*) *Heap* **where**  
*vebt-succi' t ti x* = (case *ti* of (*Leaf* *a b*) ⇒ (if *x* = 0 then (if *b* then return (Some 1) else return  
None)

```

    else return None)|
    (Node i info deg treeArray summary) ⇒ do { assert'( is-Node t);
    let (info', deg', treeList, summary') =
    (case t of Node info' deg' treeList summary' ⇒ (info', deg', treeList, summary'));
    assert'(info'=info ∧ deg'=deg ∧ is-Node t);
    case info of None ⇒ return None |
    (Some mima) ⇒ (if deg ≤ 1 then return None else
    (if x < fst mima then return (Some (fst mima)) else
    if x ≥ snd mima then return None else
    do {
        l ← lowi x (deg div 2);
        h ← highi x (deg div 2);

        assert'(l = low x (deg div 2));
        assert'(h = high x (deg div 2));
        assert'(h < length treeList);

        aktnode ← Array-Time.nth treeArray h;

```

```

let aknode' = treeList!h;

maxlow <- vebt-maxti aknode;
assert' (maxlow = vebt-maxt aknode');
if (maxlow ≠ None ∧ (Some l <_o maxlow))
then do {
  succy <- vebt-succi' aknode' aknode l;
  return (Some (2^(deg div 2)) *_o Some h +_o succy)
}
else do {
  succsum <- vebt-succi' summary' summary h;
  assert'(succsum = None ↔ vebt-succ summary' h = None);
  if succsum = None then do{
    return None}
  else
  do{
    nextnode <- Array-Time.nth treeArray (the succsum);
    minnext <- vebt-minti nextnode;
    return (Some (2^(deg div 2)) *_o succsum +_o minnext)
  }
}
})
))

```

**theorem** *vebt-succi'-rf-abstr:invar-vebt*  $t\ n \implies \langle \text{vebt-assign-raw } t\ ti \rangle \text{vebt-succi}'\ t\ ti\ x \langle \lambda r. \text{vebt-assign-raw } t\ ti\ * \uparrow(r = \text{vebt-succ } t\ x) \rangle$   
*<proof>*

**lemma** *TBOUND-vebt-succi*:  
**defines** *foo-def*:  $\bigwedge t\ x. \text{foo } t\ x \equiv 7 * (1 + \text{height } t)$   
**shows** *TBOUND* (*vebt-succi'*  $t\ ti\ x$ ) (*foo*  $t\ x$ )  
*<proof>*

**lemma** *vebt-succi-refines*: *refines* (*vebt-succi*  $ti\ x$ ) (*vebt-succi'*  $t\ ti\ x$ )  
*<proof>*

**lemma** *htt-vebt-succi*: **assumes** *invar-vebt*  $t\ n$   
**shows**  $\langle \text{vebt-assign-raw } t\ ti \rangle \text{vebt-succi } ti\ x \langle \lambda r. \text{vebt-assign-raw } t\ ti\ * \uparrow(r = \text{vebt-succ } t\ x) \rangle T[7 + 7 * (\text{nat } \lceil \text{lb } n \rceil)]$   
*<proof>*

**end**

**context begin**  
**interpretation** *VEBT-internal* *<proof>*

**partial-function** (*heap-time*) *vebt-predi*::  $VEBTi \Rightarrow \text{nat} \Rightarrow (\text{nat option}) \text{Heap}$  **where**



```

(Some mima) ⇒ ( if deg ≤ 1 then return None else
  (if x > snd mima then return (Some (snd mima)) else
    do {
      l ← lowi x (deg div 2);
      h ← highi x (deg div 2);

      assert'(l = low x (deg div 2));
      assert'(h = high x (deg div 2));
      assert'(h < length treeList);

      aknode ← Array-Time.nth treeArray h;
      let aknode' = treeList!h;
      minlow ← vebt-minti aknode;
      assert'(minlow = vebt-mint aknode');

      if (minlow ≠ None ∧ (Some l >ₒ minlow))
      then do {
        predy ← vebt-predi' aknode' aknode l;
        return ( Some (2^(deg div 2)) *ₒ Some h +ₒ predy)
      }
      else do {
        predsum ← vebt-predi' summary' summary h;
        assert'(predsum = None ↔ vebt-pred summary' h = None);
        if predsum = None then
          if x > fst mima then
            return (Some (fst mima))
          else
            return None
        else
          do{
            nextnode ← Array-Time.nth treeArray (the predsum);
            maxnext ← vebt-maxti nextnode;
            return (Some (2^(deg div 2)) *ₒ predsum +ₒ maxnext)
          }
      }
    }
  )
)

```

**theorem** *vebt-pred'-rf-abstr*:  $\text{invar-vebt } t \ n \implies \langle \text{vebt-assn-raw } t \ ti \rangle \text{vebt-predi}' \ t \ ti \ x \ \langle \lambda r. \text{vebt-assn-raw } t \ ti \ * \ \uparrow(r = \text{vebt-pred } t \ x) \rangle$   
*<proof>*

**lemma** *TBOUND-vebt-predi*:

**defines** *foo-def*:  $\bigwedge t \ x. \text{foo } t \ x \equiv 7 * (1 + \text{height } t)$

**shows** *TBOUND* (*vebt-predi'* *t ti x*) (*foo t x*)

*<proof>*

**lemma** *vebt-predi-refines*: *refines* (*vebt-predi ti x*) (*vebt-predi' t ti x*)

*<proof>*

**lemma** *htt-vebt-predi*: **assumes** *invar-vebt t n*

**shows**  $\langle \text{vebt-assn-raw } t \text{ ti} \rangle \text{vebt-predi } ti \ x \ \langle \lambda \ r. \ \text{vebt-assn-raw } t \ ti \ * \ \uparrow(r = \text{vebt-pred } t \ x) \rangle T[7 + 7 * (\text{nat } \lceil lb \ n \rceil)]$   
*<proof>*

**end**  
**end**

**theory** *VEBT-DelImperative* **imports** *VEBT-DeleteCorrectness VEBT-SuccPredImperative*  
**begin**

**context** **begin**  
**interpretation** *VEBT-internal* *<proof>*

## 21 Imperative Delete

**partial-function** (*heap-time*) *vebt-deletei*:: *VEBTi*  $\Rightarrow$  *nat*  $\Rightarrow$  *VEBTi Heap* **where**

*vebt-deletei t x* = (case *t* of (Leafi *a b*)  $\Rightarrow$  (if *x* = 0 then return (Leafi False *b*) else  
if *x* = 1 then return (Leafi *a* False) else  
return (Leafi *a b*)) |  
(Nodei *info deg treeArray summary*)  $\Rightarrow$  (  
if *deg*  $\leq$  1 then return (Nodei *info deg treeArray summary*) else  
case *info* of None  $\Rightarrow$  return (Nodei *info deg treeArray summary*) |  
(Some *mima*)  $\Rightarrow$  ( if *x* < fst *mima*  $\vee$  *x* > snd *mima* then return  
(Nodei *info deg treeArray summary*)  
else if fst *mima* = *x*  $\wedge$  snd *mima* = *x* then return (Nodei  
None *deg treeArray summary*)  
else do{ *xminew*  $\leftarrow$  (if *x* = fst *mima* then do {  
*firstcluster*  $\leftarrow$  *vebt-minti summary*;  
*firsttree*  $\leftarrow$  *Array-Time.nth treeArray (the*  
*firstcluster)*;  
  
*mintft*  $\leftarrow$  *vebt-minti firsttree*;  
let *xn* = ( $2^{\wedge}(\text{deg div } 2) * (\text{the } \text{firstcluster}) +$   
(*the mintft*));  
return (*xn, xn*)  
}  
else return (*x, fst mima*));  
let *xnew* = fst *xminew*;  
let *minew* = snd *xminew*;  
*h*  $\leftarrow$  *highi xnew (deg div 2)*;  
*l*  $\leftarrow$  *lowi xnew (deg div 2)*;  
*aknode*  $\leftarrow$  *Array-Time.nth treeArray h*;  
*aknode'*  $\leftarrow$  *vebt-deletei aknode l*;  
*treeArray'*  $\leftarrow$  *Array-Time.upd h aknode' treeArray*;  
*miny*  $\leftarrow$  *vebt-minti aknode'*;  
(if (*miny* = None) then  
do{  
*summary'*  $\leftarrow$  *vebt-deletei summary h*;  
*ma*  $\leftarrow$  (if *xnew* = snd *mima* then

```

do{
  summarax <- vebt-maxti summary';
  if summarax = None then
    return minew
  else do{
    maxtree <- Array-Time.nth treeArray' (the
summarax);
    mofmtree <- vebt-maxti maxtree;
    return (the summarax * 2^(deg div 2) +
the mofmtree )
  }
} else return (snd mima));
return (Nodei (Some (minew, ma)) deg treeArray'
summary')
} else if xnew = snd mima then
do{
  nextree <- Array-Time.nth treeArray' h;
  maxnext <- vebt-maxti nextree;
  let ma = h * 2^(deg div 2) +
(the maxnext);
  return (Nodei (Some ( minew, ma)) deg treeArray'
summary)
}
else return (Nodei (Some (minew, snd mima)) deg
treeArray' summary) )
))))

```

**end**

**context** *VEBT-internal* **begin**

Some general lemmas

**lemma** *midextr*:  $(P * Q * Q' * R \implies_A X) \implies (P * R * Q * Q' \implies_A X)$   
*<proof>*

**lemma** *groupy*:  $A * B * (C * D) \implies_A X \implies A * B * C * D \implies_A X$   
*<proof>*

**lemma** *swappa*:  $B * A * C \implies_A X \implies A * B * C \implies_A X$   
*<proof>*

**lemma** *mulcomm*:  $(i :: \text{nat}) * (2 * 2^{(va \text{ div } 2)}) = (2 * 2^{(va \text{ div } 2)}) * i$   
*<proof>*

Modified function with ghost variable

**partial-function** (*heap-time*) *vebt-deletei'*:  $VEBT \Rightarrow VEBT_i \Rightarrow \text{nat} \Rightarrow VEBT_i \text{ Heap}$  **where**  
*vebt-deletei' t ti x = (case ti of (Leafi a b)  $\Rightarrow$  (if x = 0 then return (Leafi False b) else  
if x = 1 then return (Leafi a False) else*



```

return (Leafi a b) |
(Nodei info deg treeArray summary) ⇒ (
do { assert'( is-Node t);
let (info',deg',treeList,summary') =
(case t of Node info' deg' treeList summary'
⇒ (info',deg',treeList,summary'));
assert'(info'=info ∧ deg'=deg ∧ is-Node t);
if deg ≤ 1 then return (Nodei info deg treeArray summary) else
case info of None ⇒ return (Nodei info deg treeArray summary)|
(Some mima) ⇒ (
if x < fst mima ∨ x > snd mima then return (Nodei info deg
treeArray summary)
else if fst mima = x ∧ snd mima = x then return (Nodei
None deg treeArray summary)
else do{ xminew <- (if x = fst mima then do {
firstcluster <- vebt-minti summary;
firsttree <- Array-Time.nth treeArray (the
firstcluster);
mintft <- vebt-minti firsttree;
let xn = (2^(deg div 2) * (the firstcluster) +
(the mintft) );
return (xn, xn)
}
else return (x, fst mima));
let xnew = fst xminew;
let xn' =
(if x = fst (the info')
then the (vebt-mint summary') * 2^(deg div 2)
+ the (vebt-mint (treeList ! the (vebt-mint summary'))))
else x);
assert'(xnew = xn');
let minew = snd xminew;
assert'(minew = (if x = fst (the info') then xn' else fst
(the info')));
h <- highi xnew (deg div 2);
assert'(h = high xnew (deg div 2));
assert'(h < length treeList);
l <- lowi xnew (deg div 2);
assert'(l = low xnew (deg div 2));
aknode <- Array-Time.nth treeArray h;
aknode' <- vebt-deletei' (treeList ! h) aknode l;
treeArray' <- Array-Time.upd h aknode' treeArray;
let funnode = vebt-delete (treeList ! h) l;
let treeList' = treeList[h:= funnode];
miny <- vebt-minti aknode';
assert'(miny = vebt-mint funnode);
(if (miny = None) then
do{
summaryi' <- vebt-deletei' summary' summary h;

```

```

    ma <- (if xnew = snd mima then
do{
    summax <- vebt-maxxi summaryi';
assert' (summax = vebt-maxt (vebt-delete summary'
h));

    if summax = None then
    return minew
    else do{
    maxtree <- Array-Time.nth treeArray' (the
summax);

    mofmtree <- vebt-maxxi maxtree;
    return (the summax * 2^(deg div 2) +
the mofmtree )
    }
    } else return (snd mima));
return (Nodei (Some (minew, ma)) deg treeArray'
summaryi')
} else if xnew = snd mima then
do{
    nextree <- Array-Time.nth treeArray' h;
    maxnext <- vebt-maxxi nextree;
    assert' (maxnext = vebt-maxt (treeList' ! h));
    let ma = h * 2^(deg div 2) +
(the maxnext);
    return (Nodei (Some ( minew, ma)) deg treeArray'
summary)
    }
else return (Nodei (Some (minew, snd mima)) deg
treeArray' summary) )
}}))

```

**theorem** *deleti'-rf-abstr*:  $\text{invar-vebt } t \ n \implies \langle \text{vebt-assn-raw } t \ ti \rangle \text{vebt-deletei}' \ t \ ti \ x \langle \text{vebt-assn-raw } (\text{vebt-delete } t \ x) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *TBOUND-vebt-deletei*:

**defines** *foo-def*:  $\bigwedge t \ x. \text{foo } t \ x \equiv \text{if } \text{minNull } (\text{vebt-delete } t \ x) \text{ then } 1 \text{ else } 20 * (1 + \text{height } t)$

**shows** *TBOUND* ( $\text{vebt-deletei}' \ t \ ti \ x$ ) ( $\text{foo } t \ x$ )

$\langle \text{proof} \rangle$

**lemma** *vebt-deletei-refines*:  $\text{refines } (\text{vebt-deletei } ti \ x) \ (\text{vebt-deletei}' \ t \ ti \ x)$

$\langle \text{proof} \rangle$

**lemma** *htt-vebt-deletei*: **assumes**  $\text{invar-vebt } t \ n$

**shows**  $\langle \text{vebt-assn-raw } t \ ti \rangle \text{vebt-deletei } ti \ x \langle \lambda \ r. \text{vebt-assn-raw } (\text{vebt-delete } t \ x) \ r \rangle T[20 + 20 * (\text{nat } \lceil \text{lb } n \rceil)]$

$\langle \text{proof} \rangle$

end  
end

## 22 Imperative Interface

```
theory VEBT-Intf-Imperative
  imports
    VEBT-Definitions
    VEBT-Uniqueness
    VEBT-Member
    VEBT-Insert VEBT-InsertCorrectness
    VEBT-MinMax
    VEBT-Pred VEBT-Succ
    VEBT-Delete VEBT-DeleteCorrectness
    VEBT-Bounds
    VEBT-DeleteBounds
    VEBT-Space
    VEBT-Intf-Functional
    VEBT-List-Assn
    VEBT-BuildupMemImp
    VEBT-SuccPredImperative
    VEBT-DelImperative
begin
```

### 22.1 Code Export

```
context begin
  interpretation VEBT-internal <proof>

  lemmas [code] = replicatei.simps vebt-memberi.simps highi-def lowi-def vebt-inserti.simps
    minNulli.simps vebt-succi.simps vebt-predi.simps vebt-deletei.simps

    greater.simps
end
```

```
export-code
  vebt-buildupi
  vebt-memberi
  vebt-inserti
  vebt-maxti vebt-minti
  vebt-predi vebt-succi
  vebt-deletei

checking SML-imp
```

## 22.2 Interface

**definition**  $vebt-assn::nat \Rightarrow nat \ set \Rightarrow VEBTi \Rightarrow assn$  **where**

$vebt-assn\ n\ s\ ti \equiv \exists_A\ t.\ vebt-assn-raw\ t\ ti * \uparrow(s = set-vebt\ t \wedge invar-vebt\ t\ n)$

### 22.2.1 Buildup

**context begin**

**interpretation**  $VEBT$ -internal  $\langle proof \rangle$

**interpretation**  $vebt-inst$  **for**  $n$   $\langle proof \rangle$

**lemma**  $vebt-buildupi-rule-basic[sep-heap-rules]: n > 0 \implies \langle emp \rangle\ vebt-buildupi\ n\ <\lambda\ r.\ vebt-assn\ n\ \{\} r >$   
 $\langle proof \rangle$

**lemma**  $vebt-buildupi-rule: \langle \uparrow (n > 0) \rangle\ vebt-buildupi\ n\ <\lambda\ r.\ vebt-assn\ n\ \{\} r >\ T[10 * 2^{\wedge}n]$   
 $\langle proof \rangle$

### 22.2.2 Member

**lemma**  $vebt-memberi-rule: \langle vebt-assn\ n\ s\ ti \rangle\ vebt-memberi\ ti\ x\ <\lambda\ r.\ vebt-assn\ n\ s\ ti * \uparrow(r = (x \in s)) \rangle\ T[5 + 5 * (nat\ \lceil lb\ n \rceil)]$   
 $\langle proof \rangle$

### 22.2.3 Insert

**lemma**  $vebt-inserti-rule: x < 2^{\wedge}n \implies \langle vebt-assn\ n\ s\ ti \rangle\ vebt-inserti\ ti\ x\ <\lambda\ r.\ vebt-assn\ n\ (s \cup \{x\})\ r >\ T[13 + 13 * (nat\ \lceil lb\ n \rceil)]$   
 $\langle proof \rangle$

### 22.2.4 Maximum

**lemma**  $vebt-maxti-rule: \langle vebt-assn\ n\ s\ ti \rangle\ vebt-maxti\ ti\ <\lambda\ r.\ vebt-assn\ n\ s\ ti * \uparrow(r = Some\ y \longleftrightarrow max-in-set\ s\ y) \rangle\ T[1]$   
 $\langle proof \rangle$

### 22.2.5 Minimum

**lemma**  $vebt-minti-rule: \langle vebt-assn\ n\ s\ ti \rangle\ vebt-minti\ ti\ <\lambda\ r.\ vebt-assn\ n\ s\ ti * \uparrow(r = Some\ y \longleftrightarrow min-in-set\ s\ y) \rangle\ T[1]$   
 $\langle proof \rangle$

### 22.2.6 Successor

**lemma**  $vebt-succ-i-rule: \langle vebt-assn\ n\ s\ ti \rangle\ vebt-succ-i\ ti\ x\ <\lambda\ r.\ vebt-assn\ n\ s\ ti * \uparrow(r = Some\ y \longleftrightarrow is-succ-in-set\ s\ x\ y) \rangle\ T[7 + 7 * (nat\ \lceil lb\ n \rceil)]$   
 $\langle proof \rangle$

### 22.2.7 Predecessor

**lemma** *vebt-predi-rule*:  $\langle \text{vebt-assn } n \ s \ ti \rangle \text{vebt-predi } ti \ x \ \langle \lambda \ r. \ \text{vebt-assn } n \ s \ ti \ * \ \uparrow (r = \text{Some } y \ \longleftrightarrow \text{is-pred-in-set } s \ x \ y) \rangle T[7 + 7 * (\text{nat } \lceil lb \ n \ \rceil)]$   
*<proof>*

### 22.2.8 Delete

**lemma** *vebt-deletei-rule*:  $\langle \text{vebt-assn } n \ s \ ti \rangle \text{vebt-deletei } ti \ x \ \langle \lambda \ r. \ \text{vebt-assn } n \ (s - \{x\}) \ r \rangle T[20 + 20 * (\text{nat } \lceil lb \ n \ \rceil)]$   
*<proof>*

## 22.3 Setup of VCG

**lemmas** *vebt-heap-rules*[*THEN* *htt-htD*,*sep-heap-rules*] =  
*vebt-buildupi-rule*  
*vebt-memberi-rule*  
*vebt-inserti-rule*  
*vebt-maxti-rule*  
*vebt-minti-rule*  
*vebt-succi-rule*  
*vebt-predi-rule*  
*vebt-deletei-rule*

**end**  
**end**

## 23 Interface Usage Example

**theory** *VEBT-Example*  
**imports** *VEBT-Intf-Imperative* *VEBT-Example-Setup*  
**begin**

### 23.1 Test Program

**definition** *test*  $n \ xs \ ys \equiv \text{do } \{$   
   $t \leftarrow \text{vebt-buildupi } n;$   
   $t \leftarrow \text{mfold } (\lambda x \ s. \ \text{vebt-inserti } s \ x) \ (0\#xs) \ t;$   
  
   $\text{let } f = (\lambda x. \ \text{if}_m \ \text{vebt-memberi } t \ x \ \text{then return } x \ \text{else the } \$_m \ (\text{vebt-predi } t \ x));$   
  
   $\text{mmap } f \ ys$   
   $\}$

### 23.2 Correctness without Time

The non-time part of our datastructure is fully integrated into *sep-auto*

**lemma** *fold-list-rl*[*sep-heap-rules*]:  $\forall x \in \text{set } xs. \ x < 2^n \implies \text{hoare-triple}$   
 $(\text{vebt-assn } n \ s \ t)$

$(\text{mfold } (\lambda x s. \text{vebt-inserti } s x) xs t)$   
 $(\lambda t'. \text{vebt-assn } n (s \cup \text{set } xs) t')$   
 <proof>

**lemma** *test-hoare*:  $\llbracket \forall x \in \text{set } xs. x < 2^{\wedge} n; n > 0 \rrbracket \implies$   
 $\langle \text{emp} \rangle (\text{test } n xs ys) < \lambda r. \uparrow (r = \text{map } (\lambda y. (\text{GREATEST } y'. y' \in \text{insert } 0 (\text{set } xs) \wedge y' \leq y)) ys)$   
 $>_t$   
 <proof>

### 23.3 Time Bound Reasoning

We use some ad-hoc reasoning to also show the time-bound of our test program. A generalization of such methods, or the integration of this entry into existing reasoning frameworks with time is left to future work.

**lemma** *insert-time-pure*[*cond-TBOUND*]:  $a < 2^{\wedge} n \implies$   
 $\S \text{vebt-assn } n S ti \S \text{TBOUND } (\text{vebt-inserti } ti a) (13 + 13 * \text{nat } \lceil \log 2 (\text{real } n) \rceil)$   
 <proof>

**lemma** *member-time-pure*[*cond-TBOUND*]:  $\S \text{vebt-assn } n S ti \S \text{TBOUND } (\text{vebt-memberi } ti a) (5 + 5 * \text{nat } \lceil \log 2 (\text{real } n) \rceil)$   
 <proof>

**lemma** *pred-time-pure*[*cond-TBOUND*]:  $\S \text{vebt-assn } n S ti \S \text{TBOUND } (\text{vebt-predi } ti a) (7 + 7 * \text{nat } \lceil \log 2 (\text{real } n) \rceil)$   
 <proof>

**lemma** *TBOUND-mfold*[*cond-TBOUND*]:  
 $(\bigwedge x. x \in \text{set } xs \implies x < 2^{\wedge} n) \implies$   
 $\S \text{vebt-assn } n S ti \S \text{TBOUND } (\text{mfold } (\lambda x s. \text{vebt-inserti } s x) xs ti) (\text{length } xs * (13 + 13 * \text{nat } \lceil \log 2 n \rceil) + 1)$   
 <proof>

**lemma** *TBOUND-mmap*[*cond-TBOUND*]:  
**defines** *b-def*:  $b \text{ ys } n \equiv 1 + \text{length } ys * (5 + 5 * \text{nat } \lceil \log 2 (\text{real } n) \rceil) + 9 + 7 * \text{nat } \lceil \log 2 (\text{real } n) \rceil$   
**shows**  $\S \text{vebt-assn } n S ti \S \text{TBOUND}$   
 $(\text{mmap } (\lambda x. \text{if}_m \text{vebt-memberi } ti x \text{ then return } x$   
 $\quad \text{else vebt-predi } ti x \gg (\lambda x. \text{return } (\text{the } x))) \text{ ys}) (b \text{ ys } n)$   
 <proof>

**lemma** *TBOUND-test*[*cond-TBOUND*]:  $\llbracket \forall x \in \text{set } xs. x < 2^{\wedge} n; n > 0 \rrbracket \implies$   
 $\S \uparrow (n > 0) \S \text{TBOUND } (\text{test } n xs ys) (10 * 2^{\wedge} n + ($   
 $\quad (\text{length } (0 \# xs) * (13 + 13 * \text{nat } \lceil \log 2 n \rceil) + 1) +$   
 $\quad (1 + \text{length } ys * (5 + 5 * \text{nat } \lceil \log 2 (\text{real } n) \rceil) + 9 + 7 * \text{nat } \lceil \log 2 (\text{real } n) \rceil)))$   
 <proof>

**lemma** *test-hoare-with-time*:  $\llbracket \forall x \in \text{set } xs. x < 2^{\wedge} n; n > 0 \rrbracket \implies$

```

    <emp> (test n xs ys) <lr. ↑(r = map (λy. (GREATEST y'. y'∈insert 0 (set xs) ∧ y'≤y)) ys) *
true >
    T[10 * 2 ^ n +
      (length (0 # xs) * (13 + 13 * nat [log 2 (real n)])) + 1 +
      (1 + length ys * (5 + 5 * nat [log 2 (real n)] + 9 + 7 * nat [log 2 (real n)]))]
    <proof>

```

end

## 24 Conclusion

We have formalized van Emde Boas trees in Isabelle, proving correct a functional and an imperative version, together with space and run-time bounds. This work amends a list [4] of formally verified CLRS algorithms [3].

Closing we sketch some enhancements of van Emde Boas trees in Isabelle. An examination of the data structure points out that there is probably a *join* operation with the semantics  $set-vebt (vebt-join s t) = set-vebt s \cup set-vebt t$ . We make the restriction of joining only valid trees with equal degree numbers. Obviously, the join of two leaves is trivial. If one tree is empty or singleton, a join is implemented by immediately returning the other tree or performing an insertion before. Otherwise, summary and subtrees are to be joined recursively and afterwards we have to determine minimum and maximum. Certainly, this last step can be complicated, because argument trees may also coincide on minima or maxima.

One may also consider the treatment of associated satellite data. Those are to be stored in ordinary subtrees, whereas the definition of summary trees does not have to be changed. We can transfer this to Isabelle by introducing another data type representing van Emde Boas trees. The adapted *naive-member* and *membermima* still refer to integer keys, but we add an auxiliary function *assoc* such that  $assoc t x y$  holds iff the key  $x$  is associated with the value  $y$ . A *both-member-options* is also defined and can be used for specifying a suitable validness invariant. We may show a conjecture like  $both-member-options t x \longleftrightarrow \exists y. assoc t x y$ . Besides, valid trees enforce keys to be associated with at most one value. All canonical functions  $f$  are shifted to the new type and return a key-value pair  $(x, y)$  or the modified tree. Proofs for being  $x$  the desired successor etc. are obtained by reuse and adaptation of prior proofs. In addition, modified canonical functions  $f'$  may only return the associated values  $y$ . We show the proposition  $\exists x. f t i = (x, y) \longleftrightarrow f' t i = y$ . All writing operations require a reasoning regarding the proper (non-)modification of associations. The modified functions  $f'$  are to be exposed to a user later on.

Moreover, we did not consider lazy implementation. Currently, *vebt-buildup n* generates a full van Emde Boas tree of degree  $n$ . A *lazy implementation* would construct a subtree only if needed. From this just a constant amount of additional effort per recursive step will arise. Thereby, proven running time bounds of  $\mathcal{O}(\log \log u)$  will be preserved. Beside this, a lazy implementation can also be obtained by exporting verified Isabelle code to Haskell, which heavily applies the lazy evaluation technique.

Obviously, a lazy implementation would drastically reduce memory usage. Each insertion allocates  $\mathcal{O}(\log \log u)$  space and hence an implementation that does not store empty subtrees

gives us memory consumption in  $\mathcal{O}(n \cdot \log \log u)$  where  $n$  is the number of elements currently stored. Furthermore, one may replace ordinary arrays by *dynamic perfect hashing* [9] allowing treatment of elements in (amortized) constant time and linear space. Unfortunately, a linear memory consumption in  $\mathcal{O}(n)$  is achieved at cost of some worst case runtime bounds [10]. By this,  $\mathcal{O}(\log \log u)$  is turned to an amortized bound for *vebt-insert* and *vebt-delete*, since the complexity of those functions is indeed affected by the amortization. An implementation of this van Emde Boas tree variant requires verified dynamic perfect hashing and amortization in Isabelle to build on.

We used Imperative HOL due to its support of arrays and type reflexive references that are necessary for setting up a recursive tree data structure. For generating verified code, however, there also exist other frameworks, e.g. Isabelle LLVM [11] [12]. It supports refinement-based verification of correctness and worst-case time complexities. Additionally, verified programs can be exported to LLVM code, which itself is compiled to executable machine code. Strikingly, code of the introsort algorithm generated by this formalization stayed competitive with the GNU C++ library [12].



## References

- [1] P. van Emde Boas. “Preserving order in a forest in less than logarithmic time”. In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. 1975, pp. 75–84. DOI: [10.1109/SFCS.1975.26](https://doi.org/10.1109/SFCS.1975.26).
- [2] P. E. Boas, R. Kaas, and E. Zijlstra. “Design and implementation of an efficient priority queue”. In: *Mathematical Systems Theory* 10.1 (1976), pp. 99–127. DOI: [10.1007/bf01683268](https://doi.org/10.1007/bf01683268).
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.
- [4] T. Nipkow, M. Eberl, and M. P. L. Haslbeck. “Verified Textbook Algorithms. A Biased Survey”. In: *ATVA 2020, Automated Technology for Verification and Analysis*. Ed. by D. V. Hung and O. Sokolsky. Vol. 12302. LNCS. Invited paper. Springer, 2020, pp. 25–53.
- [5] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, Bohua Zhan. *Functional Algorithms, Verified!* <https://functional-algorithms-verified.org/>. 2021.
- [6] P. Lammich and R. Meis. “A Separation Logic Framework for Imperative HOL”. In: *Archive of Formal Proofs* (Nov. 2012). [https://isa-afp.org/entries/Separation\\_Logic\\_Imperative\\_HOL.html](https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html), Formal proof development. ISSN: 2150-914x.
- [7] P. Lammich. “Refinement to Imperative HOL”. In: *Journal of Automated Reasoning* 62 (Apr. 2019). DOI: [10.1007/s10817-017-9437-1](https://doi.org/10.1007/s10817-017-9437-1).
- [8] B. Zhan and M. P. L. Haslbeck. *Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle*. 2018. arXiv: [1802.01336](https://arxiv.org/abs/1802.01336) [cs.LG].
- [9] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. “Dynamic Perfect Hashing: Upper and Lower Bounds”. In: vol. 0. Jan. 1988, pp. 524–531. DOI: [10.1109/SFCS.1988.21968](https://doi.org/10.1109/SFCS.1988.21968).
- [10] M. Straka. “Functional Data Structures and Algorithms”. [https://ufal.mff.cuni.cz/~straka/theses/doctoral-functional\\_data\\_structures\\_and\\_algorithms.pdf](https://ufal.mff.cuni.cz/~straka/theses/doctoral-functional_data_structures_and_algorithms.pdf). PhD thesis. Charles University in Prague, Faculty of Mathematics and Physics, 2013.
- [11] P. Lammich. “Generating Verified LLVM from Isabelle/HOL”. English. In: *ITP 2019: Interactive Theorem Proving*. Interactive Theorem Proving, ITP 2019 ; Conference date: 08-09-2019 Through 13-09-2019. June 2019.
- [12] M. P. L. Haslbeck and P. Lammich. “For a Few Dollars More”. In: *Programming Languages and Systems*. Ed. by N. Yoshida. Cham: Springer International Publishing, 2021, pp. 292–319. ISBN: 978-3-030-72019-3.