

IMP2 Binary Heap

Simon Griebel

March 17, 2025

Abstract

In this submission array-based binary minimum heaps are formalized. The correctness of the following heap operations is proven: insert, get-min, delete-min and make-heap. These are then used to verify an in-place heapsort. The formalization is based on IMP2, an imperative program verification framework implemented in Isabelle/HOL. The verified heap functions are iterative versions of the partly recursive functions found in “Algorithms and Data Structures – The Basic Toolbox” by K. Mehlhorn and P. Sanders and “Introduction to Algorithms” by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.

Contents

1	Introduction	3
2	Heap Related Definitions and Theorems	3
2.1	Array Bounds	3
2.2	Parent and Children	3
2.2.1	Definitions	3
2.2.2	Lemmas	4
2.3	Heap Invariants	5
2.3.1	Definitions	5
2.3.2	Lemmas	6
2.3.3	First Heap Element	7
3	General Lemmas on Arrays	8
3.1	Lemmas on <i>mset-ran</i>	8
3.2	Lemmas on <i>swap</i> and <i>eq-on</i>	9
4	Imperative Heap Implementation	10
4.1	Simple Functions	11
4.1.1	Parent, Children and Swap	11
4.1.2	<i>get-min</i>	11
4.2	Modifying Functions	12

4.2.1	<i>sift-up</i> and <i>insert</i>	12
4.2.2	<i>sift-down</i> , <i>del-min</i> and <i>make-heap</i>	14
4.3	Heapsort Implementation	16
4.3.1	Auxiliary Lemmas	16
4.3.2	Implementation	17

```

theory IMP2-Binary-Heap
  imports IMP2.IMP2 IMP2.IMP2-Aux-Lemmas
begin

```

1 Introduction

In this submission imperative versions of the following array-based binary minimum heap functions are implemented and verified: insert, get-min, delete-min, make-heap. The latter three are then used to prove the correctness of an in-place heapsort, which sorts an array in descending order. To do that in Isabelle/HOL, the proof framework IMP2 [2] is used. Here arrays are modeled by $int \Rightarrow int$ functions. The imperative implementations are iterative versions of the partly recursive algorithms described in [3] and [1].

This submission starts with the basic definitions and lemmas, which are needed for array-based binary heaps. These definitions and lemmas are parameterised with an arbitrary (transitive) comparison function (where such a function is needed), so they are not only applicable to minimum heaps. After some more general, useful lemmas on arrays, the imperative minimum heap functions and the heapsort are implemented and verified.

2 Heap Related Definitions and Theorems

2.1 Array Bounds

A small helper function is used to define valid array indices. Note that the lower index bound l is arbitrary and not fixed to 0 or 1. The upper index bound r is not a valid index itself, so that the empty array can be denoted by having $l = r$.

abbreviation $bounded :: int \Rightarrow int \Rightarrow int \Rightarrow bool$ **where**
 $bounded\ l\ r\ x \equiv l \leq x \wedge x < r$

2.2 Parent and Children

2.2.1 Definitions

For the notion of an array-based binary heap, the parent and child relations on the array indices need to be defined.

definition $parent :: int \Rightarrow int \Rightarrow int$ **where**
 $parent\ l\ c = l + (c - l - 1) \text{ div } 2$

definition $l-child :: int \Rightarrow int \Rightarrow int$ **where**
 $l-child\ l\ p = 2 * p - l + 1$

definition $r\text{-child} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ **where**

$$r\text{-child } l \ p = 2 * p - l + 2$$

2.2.2 Lemmas

lemma $\text{parent-upper-bound}$: $\text{parent } l \ c < c \longleftrightarrow l \leq c$

$\langle \text{proof} \rangle$

lemma $\text{parent-upper-bound-alt}$: $l \leq \text{parent } l \ c \Longrightarrow \text{parent } l \ c < c$

$\langle \text{proof} \rangle$

lemma $\text{parent-lower-bound}$: $l \leq \text{parent } l \ c \longleftrightarrow l < c$

$\langle \text{proof} \rangle$

lemma $\text{grand-parent-upper-bound}$: $\text{parent } l \ (\text{parent } l \ c) < c \longleftrightarrow l \leq c$

$\langle \text{proof} \rangle$

corollary parent-bounds : $l < x \Longrightarrow x < r \Longrightarrow \text{bounded } l \ r \ (\text{parent } l \ x)$

$\langle \text{proof} \rangle$

lemma $l\text{-child-lower-bound}$: $p < l\text{-child } l \ p \longleftrightarrow l \leq p$

$\langle \text{proof} \rangle$

corollary $l\text{-child-lower-bound-alt}$: $l \leq x \Longrightarrow x \leq p \Longrightarrow x < l\text{-child } l \ p$

$\langle \text{proof} \rangle$

lemma $\text{parent-l-child[simp]}$: $\text{parent } l \ (l\text{-child } l \ n) = n$

$\langle \text{proof} \rangle$

lemma $r\text{-child-lower-bound}$: $l \leq p \Longrightarrow p < r\text{-child } l \ p$

$\langle \text{proof} \rangle$

corollary $r\text{-child-lower-bound-alt}$: $l \leq x \Longrightarrow x \leq p \Longrightarrow x < r\text{-child } l \ p$

$\langle \text{proof} \rangle$

lemma $\text{parent-r-child[simp]}$: $\text{parent } l \ (r\text{-child } l \ n) = n$

$\langle \text{proof} \rangle$

lemma smaller-l-child : $l\text{-child } l \ x < r\text{-child } l \ x$

$\langle \text{proof} \rangle$

lemma *parent-two-children*:

$$(c = l\text{-child } l \ p \vee c = r\text{-child } l \ p) \longleftrightarrow \text{parent } l \ c = p$$

<proof>

2.3 Heap Invariants

2.3.1 Definitions

The following heap invariants and the following lemmas are parameterised with an arbitrary (transitive) comparison function. For the concrete function implementations at the end of this submission \leq on ints is used.

For the *make-heap* function, which transforms an unordered array into a valid heap, the notion of a partial heap is needed. Here the heap invariant only holds for array indices between a certain valid array index m and r . The standard heap invariant is then simply the special case where $m = l$.

definition *is-partial-heap*

$$:: ('a::\text{order} \Rightarrow 'a::\text{order} \Rightarrow \text{bool}) \Rightarrow (\text{int} \Rightarrow 'a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \textbf{ where}$$

$$\textit{is-partial-heap cmp heap } l \ m \ r = (\forall x. \textit{bounded } m \ r \ x \longrightarrow \textit{bounded } m \ r \ (\textit{parent } l \ x) \longrightarrow \textit{cmp } (\textit{heap } (\textit{parent } l \ x)) \ (\textit{heap } x))$$

abbreviation *is-heap*

$$:: ('a::\text{order} \Rightarrow 'a::\text{order} \Rightarrow \text{bool}) \Rightarrow (\text{int} \Rightarrow 'a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \textbf{ where}$$

$$\textit{is-heap cmp heap } l \ r \equiv \textit{is-partial-heap cmp heap } l \ l \ r$$

During all of the modifying heap functions the heap invariant is temporarily violated at a single index i and it is then gradually restored by either *sift-down* or *sift-up*. The following definitions formalize these weakened invariants.

The second part of the conjunction in the following definitions states, that the comparison between the parent of i and each of the children of i evaluates to *True* without explicitly using the child relations.

definition *is-partial-heap-except-down*

$$:: ('a::\text{order} \Rightarrow 'a::\text{order} \Rightarrow \text{bool}) \Rightarrow (\text{int} \Rightarrow 'a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \textbf{ where}$$

$$\begin{aligned} \textit{is-partial-heap-except-down cmp heap } l \ m \ r \ i = & (\forall x. \textit{bounded } m \ r \ x \longrightarrow \\ & ((\textit{parent } l \ x \neq i \longrightarrow \textit{bounded } m \ r \ (\textit{parent } l \ x) \longrightarrow \textit{cmp } (\textit{heap } (\textit{parent } l \\ & x)) \ (\textit{heap } x)) \wedge \\ & (\textit{parent } l \ x = i \longrightarrow \textit{bounded } m \ r \ (\textit{parent } l \ (\textit{parent } l \ x)) \\ & \longrightarrow \textit{cmp } (\textit{heap } (\textit{parent } l \ (\textit{parent } l \ x))) \ (\textit{heap } x)))) \end{aligned}$$

abbreviation *is-heap-except-down*

$:: ('a::order \Rightarrow 'a::order \Rightarrow bool) \Rightarrow (int \Rightarrow 'a::order) \Rightarrow int \Rightarrow int \Rightarrow int \Rightarrow bool$ **where**
is-heap-except-down *cmp heap l r i* \equiv *is-partial-heap-except-down* *cmp heap l l r i*

As mentioned the notion of a partial heap is only needed for *make-heap*, which only uses *sift-down* internally, so there doesn't need to be an additional definition for the partial heap version of the *sift-up* invariant.

definition *is-heap-except-up*

$:: ('a::order \Rightarrow 'a::order \Rightarrow bool) \Rightarrow (int \Rightarrow 'a::order) \Rightarrow int \Rightarrow int \Rightarrow int \Rightarrow bool$ **where**
is-heap-except-up *cmp heap l r i* $= (\forall x. \text{bounded } l r x \longrightarrow ((x \neq i \longrightarrow \text{bounded } l r (\text{parent } l x) \longrightarrow \text{cmp } (\text{heap } (\text{parent } l x)) (\text{heap } x)) \wedge (\text{parent } l x = i \longrightarrow \text{bounded } l r (\text{parent } l (\text{parent } l x)) \longrightarrow \text{cmp } (\text{heap } (\text{parent } l (\text{parent } l x))) (\text{heap } x))))$

2.3.2 Lemmas

lemma *empty-partial-heap[simp]*: *is-partial-heap* *cmp heap l r r*
 $\langle \text{proof} \rangle$

lemma *is-partial-heap-smaller-back*:

is-partial-heap *cmp heap l m r* $\implies r' \leq r \implies$ *is-partial-heap* *cmp heap l m r'*
 $\langle \text{proof} \rangle$

lemma *is-partial-heap-smaller-front*:

is-partial-heap *cmp heap l m r* $\implies m \leq m' \implies$ *is-partial-heap* *cmp heap l m' r*
 $\langle \text{proof} \rangle$

The second half of each array is a partial binary heap, since it contains only leaves, which are all trivial binary heaps.

lemma *snd-half-is-partial-heap*:

$(l + r) \text{ div } 2 \leq m \implies$ *is-partial-heap* *cmp heap l m r*
 $\langle \text{proof} \rangle$

lemma *modify-outside-partial-heap*:

assumes

heap = heap' on $\{m..<r\}$

is-partial-heap *cmp heap l m r*

shows *is-partial-heap* *cmp heap' l m r*

<proof>

The next few lemmas formalize how the heap invariant is weakened, when the heap is modified in a certain way.

This lemma is used by *make-heap*.

lemma *partial-heap-added-first-el*:

assumes

$l \leq m \ m \leq r$

is-partial-heap cmp heap l (m + 1) r

shows *is-partial-heap-except-down cmp heap l m r m*

<proof>

This lemma is used by *del-min*.

lemma *heap-changed-first-el*:

assumes *is-heap cmp heap l r l ≤ r*

shows *is-heap-except-down cmp (heap(l := b)) l r l*

<proof>

This lemma is used by *insert*.

lemma *heap-appended-el*:

assumes

is-heap cmp heap l r

heap = heap' on {l..<r}

shows *is-heap-except-up cmp heap' l (r+1) r*

<proof>

2.3.3 First Heap Element

The next step is to show that the first element of the heap is always the “smallest” according to the given comparison function. For the proof a rule for strong induction on lower bounded integers is needed. Its proof is based on the proof of strong induction on natural numbers found in [4].

lemma *strong-int-gr-induct-helper*:

assumes $k < (i::int) \ (\wedge i. k < i \implies (\wedge j. k < j \implies j < i \implies P j)) \implies P i$

shows $\wedge j. k < j \implies j < i \implies P j$

<proof>

theorem *strong-int-gr-induct*:

assumes

$k < (i::int)$

$(\wedge i. k < i \implies (\wedge j. k < j \implies j < i \implies P j)) \implies P i$

shows $P i$

<proof>

Now the main theorem, that the first heap element is the “smallest” according to the given comparison function, can be proven.

theorem *heap-first-el:*

assumes

is-heap cmp heap l r

transp cmp

l < x x < r

shows *cmp (heap l) (heap x)*

<proof>

3 General Lemmas on Arrays

Some additional lemmas on *mset-ran*, *swap* and *eq-on* are needed for the final proofs.

3.1 Lemmas on *mset-ran*

abbreviation *arr-mset* :: $(int \Rightarrow 'a) \Rightarrow int \Rightarrow int \Rightarrow 'a$ multiset **where**

arr-mset arr l r \equiv *mset-ran arr {l..<r}*

lemma *in-mset-imp-in-array:*

$x \in \# (arr\text{-}mset\ arr\ l\ r) \iff (\exists i. \text{bounded } l\ r\ i \wedge arr\ i = x)$

<proof>

lemma *arr-mset-remove-last:*

$l \leq r \implies arr\text{-}mset\ arr\ l\ r = arr\text{-}mset\ arr\ l\ (r + 1) - \{\#arr\ r\ \#\}$

<proof>

lemma *arr-mset-append:*

$l \leq r \implies arr\text{-}mset\ arr\ l\ (r + 1) = arr\text{-}mset\ arr\ l\ r + \{\#arr\ r\ \#\}$

<proof>

corollary *arr-mset-append-alt:*

$l \leq r \implies arr\text{-}mset\ (arr(r := b))\ l\ (r + 1) = arr\text{-}mset\ arr\ l\ r + \{\#b\ \#\}$

<proof>

lemma *arr-mset-remove-first:*

$i \leq r \implies arr\text{-}mset\ arr\ (i - 1)\ r = arr\text{-}mset\ arr\ i\ r + \{\#arr\ (i - 1)\ \#\}$

<proof>

lemma *arr-mset-split:*

assumes $l \leq m \ m \leq r$
shows $\text{arr-mset arr } l \ r = \text{arr-mset arr } l \ m + \text{arr-mset arr } m \ r$
 $\langle \text{proof} \rangle$

That the first element in a heap is the “smallest”, can now be expressed using multisets.

corollary *heap-first-el-alt*:

assumes
transp cmp
is-heap cmp heap l r
 $x \in \# (\text{arr-mset heap } l \ r)$
 $\text{heap } l \neq x$
shows $\text{cmp } (\text{heap } l) \ x$
 $\langle \text{proof} \rangle$

3.2 Lemmas on *swap* and *eq-on*

lemma *eq-on-subset*:

$\text{arr1} = \text{arr2 on } R \implies S \subseteq R \implies \text{arr1} = \text{arr2 on } S$
 $\langle \text{proof} \rangle$

lemma *swap-swaps*:

$\text{arr}' = \text{swap arr } x \ y \implies \text{arr}' \ y = \text{arr } x \wedge \text{arr}' \ x = \text{arr } y$
 $\langle \text{proof} \rangle$

lemma *swap-only-swaps*:

$\text{arr}' = \text{swap arr } x \ y \implies z \neq x \implies z \neq y \implies \text{arr}' \ z = \text{arr } z$
 $\langle \text{proof} \rangle$

lemma *swap-commute*: $\text{swap arr } x \ y = \text{swap arr } y \ x$

$\langle \text{proof} \rangle$

lemma *swap-eq-on*:

$\text{arr1} = \text{arr2 on } S \implies x \notin S \implies y \notin S \implies \text{arr1} = \text{swap arr2 } x \ y \text{ on } S$
 $\langle \text{proof} \rangle$

corollary *swap-parent-eq-on*:

assumes
 $\text{arr1} = \text{arr2 on } - \{l..<r\}$
 $l < c \ c < r$
shows $\text{arr1} = \text{swap arr2 } (\text{parent } l \ c) \ c \text{ on } - \{l..<r\}$
 $\langle \text{proof} \rangle$

corollary *swap-child-eq-on*:

assumes

$arr1 = arr2$ on $-\{l..<r\}$
 $c = l\text{-child } l\ p \vee c = r\text{-child } l\ p$
 $l \leq p\ c < r$

shows $arr1 = swap\ arr2\ p\ c$ on $-\{l..<r\}$

$\langle proof \rangle$

lemma *swap-child-mset*:

assumes

$arr\text{-mset } arr1\ l\ r = arr\text{-mset } arr2\ l\ r$
 $c = l\text{-child } l\ p \vee c = r\text{-child } l\ p$
 $l \leq p\ c < r$

shows $arr\text{-mset } arr1\ l\ r = arr\text{-mset } (swap\ arr2\ p\ c)\ l\ r$

$\langle proof \rangle$

The following lemma shows, which propositions have to hold on the pre-swap array, so that a comparison between two elements holds on the post-swap array. This is useful for the proofs of the loop invariants of *sift-up* and *sift-down*. The lemma is kept quite general (except for the argument order) and could probably be more closely related to the parent relation for more concise proofs.

lemma *cmp-swapI*:

fixes $arr::'a::order \Rightarrow 'a::order$

assumes

$m < n \wedge x < y$

$m < n \wedge x < y \implies x = m \implies y = n \implies P\ (arr\ n)\ (arr\ m)$

$m < n \wedge x < y \implies x \neq m \implies x \neq n \implies y \neq m \implies y \neq n \implies P\ (arr\ m)\ (arr\ n)$

$m < n \wedge x < y \implies x = m \implies y \neq m \implies y \neq n \implies P\ (arr\ y)\ (arr\ n)$

$m < n \wedge x < y \implies x = n \implies y \neq m \implies y \neq n \implies P\ (arr\ m)\ (arr\ y)$

$m < n \wedge x < y \implies x \neq m \implies x \neq n \implies y = n \implies P\ (arr\ m)\ (arr\ x)$

$m < n \wedge x < y \implies x \neq m \implies x \neq n \implies y = m \implies P\ (arr\ x)\ (arr\ n)$

shows $P\ (swap\ arr\ x\ y\ m)\ (swap\ arr\ x\ y\ n)$

$\langle proof \rangle$

4 Imperative Heap Implementation

The following imperative heap functions are based on [3] and [1]. All functions, that are recursive in these books, are iterative in the following implementations. The function definitions are done with IMP2 [2]. From now on the heaps only contain ints and only use \leq as comparison function. The aux-

iliary lemmas used from now on are heavily modeled after the proof goals, that are generated by the vcg tool (also part of IMP2).

4.1 Simple Functions

4.1.1 Parent, Children and Swap

In this section the parent and children relations are expressed as IMP2 procedures. Additionally a simple procedure, that swaps two array elements, is defined.

```
procedure-spec prnt (l, x) returns p
assumes True
ensures p = parent l0 x0
defines  $\langle p = ((x - l - 1) / 2 + l) \rangle$ 
 $\langle proof \rangle$ 
```

```
procedure-spec left-child (l, x) returns lc
assumes True
ensures lc = l-child l0 x0
defines  $\langle lc = 2 * x - l + 1 \rangle$ 
 $\langle proof \rangle$ 
```

```
procedure-spec right-child (l, x) returns rc
assumes True
ensures rc = r-child l0 x0
defines  $\langle rc = 2 * x - l + 2 \rangle$ 
 $\langle proof \rangle$ 
```

```
procedure-spec swp (heap, x, y) returns heap
assumes True
ensures heap = swap heap0 x0 y0
defines  $\langle tmp = heap[x]; heap[x] = heap[y]; heap[y] = tmp \rangle$ 
 $\langle proof \rangle$ 
```

4.1.2 *get-min*

In this section *get-min* is defined, which simply returns the first element (the minimum) of the heap. For this definition an additional theorem is proven, which enables the use of *Min-mset* in the postcondition.

theorem *heap-minimum*:

```
assumes
  l < r
  is-heap ( $\leq$ ) heap l r
```

shows $heap\ l = Min\text{-}mset\ (arr\text{-}mset\ heap\ l\ r)$
 $\langle proof \rangle$

procedure-spec $get\text{-}min\ (heap, l, r)$ **returns** min
assumes $l < r \wedge is\text{-}heap\ (\leq)\ heap\ l\ r$
ensures $min = Min\text{-}mset\ (arr\text{-}mset\ heap_0\ l_0\ r_0)$
for $heap[]\ l\ r$
defines $\langle min = heap[l] \rangle$
 $\langle proof \rangle$

4.2 Modifying Functions

4.2.1 *sift-up* and *insert*

The next heap function is *insert*, which internally uses *sift-up*. In the beginning of this section *sift-up-step* is proven, which states that each *sift-up* loop iteration correctly transforms the weakened heap invariant. For its proof two additional auxiliary lemmas are used. After *sift-up-step* *sift-up* and then *insert* are verified.

sift-up-step can be proven directly by the smt-solver without auxiliary lemmas, but they were introduced to show the proof details. The analogous proofs for *sift-down* were just solved with smt, since the proof structure should be very similar, even though the *sift-down* proof goals are slightly more complex.

lemma *sift-up-step-aux1*:

fixes $heap::int \Rightarrow int$
assumes
 $is\text{-}heap\text{-}except\text{-}up\ (\leq)\ heap\ l\ r\ x$
 $parent\ l\ x \geq l$
 $(heap\ x) \leq (heap\ (parent\ l\ x))$
 $bounded\ l\ r\ k$
 $k \neq (parent\ l\ x)$
 $bounded\ l\ r\ (parent\ l\ k)$
shows $(swap\ heap\ (parent\ l\ x)\ x\ (parent\ l\ k)) \leq (swap\ heap\ (parent\ l\ x)\ x\ k)$
 $\langle proof \rangle$

lemma *sift-up-step-aux2*:

fixes $heap::int \Rightarrow int$
assumes
 $is\text{-}heap\text{-}except\text{-}up\ (\leq)\ heap\ l\ r\ x$
 $parent\ l\ x \geq l$
 $heap\ x \leq (heap\ (parent\ l\ x))$

bounded l r k
parent l k = parent l x
bounded l r (parent l (parent l k))

shows

swap heap (parent l x) x (parent l (parent l k)) ≤ swap heap (parent l x)
x k
 ⟨*proof*⟩

lemma *sift-up-step*:

fixes *heap::int ⇒ int*

assumes

is-heap-except-up (≤) heap l r x
parent l x ≥ l
(heap x) ≤ (heap (parent l x))

shows *is-heap-except-up (≤) (swap heap (parent l x) x) l r (parent l x)*
 ⟨*proof*⟩

sift-up restores the heap invariant, that is only violated at the current position, by iteratively swapping the current element with its parent until the beginning of the array is reached or the current element is bigger than its parent.

procedure-spec *sift-up (heap, l, r, x) returns heap*

assumes *is-heap-except-up (≤) heap l r x ∧ bounded l r x*

ensures *is-heap (≤) heap l₀ r₀ ∧*

arr-mset heap₀ l₀ r₀ = arr-mset heap l₀ r₀ ∧

heap₀ = heap on - {l₀..<r₀}

for *heap[] l x r*

defines ‹

p = prnt(l, x);

while (x > l ∧ heap[x] ≤ heap[p])

 @*variant* ‹*x - l*›

 @*invariant* ‹*is-heap-except-up (≤) heap l r x ∧ p = parent l x ∧*

bounded l r x ∧ arr-mset heap₀ l₀ r₀ = arr-mset heap l r ∧

heap₀ = heap on - {l₀..<r}›

{

heap = swp(heap, p, x);

x = p;

p = prnt(l, x)

}›

⟨*proof*⟩

insert inserts an element into a heap by appending it to the heap and restoring the heap invariant with *sift-up*.

procedure-spec *insert (heap, l, r, el) returns (heap, l, r)*

```

assumes is-heap ( $\leq$ ) heap l r  $\wedge$   $l \leq r$ 
ensures is-heap ( $\leq$ ) heap l r  $\wedge$ 
    arr-mset heap l r = arr-mset heap0 l0 r0 + {#el0#}  $\wedge$ 
     $l = l_0 \wedge r = r_0 + 1 \wedge \text{heap}_0 = \text{heap on} - \{l..<r\}$ 
for heap l r el
defines  $\langle$ 
    heap[r] = el;
    x = r;
    r = r + 1;
    heap = sift-up(heap, l, r, x)
 $\rangle$ 
 $\langle$ proof $\rangle$ 

```

4.2.2 *sift-down*, *del-min* and *make-heap*

The next heap functions are *del-min* and *make-heap*, which both use *sift-down* to restore/establish the heap invariant. *sift-down* is proven first (this time without additional auxiliary lemmas) followed by *del-min* and *make-heap*.

sift-down restores the heap invariant, that is only violated at the current position, by iteratively swapping the current element with its smallest child until the end of the array is reached or the current element is smaller than its children.

```

procedure-spec sift-down(heap, l, r, x) returns heap
assumes is-partial-heap-except-down ( $\leq$ ) heap l x r x  $\wedge$   $l \leq x \wedge x \leq r$ 
ensures is-partial-heap ( $\leq$ ) heap l0 x0 r0  $\wedge$ 
    arr-mset heap0 l0 r0 = arr-mset heap l0 r0  $\wedge$ 
    heap0 = heap on - {l0..r0}
defines  $\langle$ 
    lc = left-child(l, x);
    rc = right-child(l, x);
    while ( $lc < r \wedge (\text{heap}[lc] < \text{heap}[x] \vee (rc < r \wedge \text{heap}[rc] < \text{heap}[x]))$ )
         $\langle$ variant  $\langle r - x \rangle$ 
         $\langle$ invariant  $\langle$ is-partial-heap-except-down ( $\leq$ ) heap l x0 r x  $\wedge$ 
             $x_0 \leq x \wedge x \leq r \wedge lc = l\text{-child } l \ x \wedge rc = r\text{-child } l \ x \wedge$ 
            arr-mset heap0 l r = arr-mset heap l r  $\wedge$ 
            heap0 = heap on - {l..r} $\rangle$ 
        {
            smallest = lc;
            if ( $rc < r \wedge \text{heap}[rc] < \text{heap}[lc]$ ) {
                smallest = rc
            }
        };
    heap = swp(heap, x, smallest);
    x = smallest;

```

```

    lc = left-child(l, x);
    rc = right-child(l, x)
  }
  <proof>

```

del-min needs an additional lemma which shows, that it actually removes (only) the minimum from the heap.

lemma *del-min-mset*:

fixes *heap*::int \Rightarrow int

assumes

$l < r$

is-heap (\leq) *heap* *l* *r*

mod-heap = *heap*(*l* := *heap* (*r* - 1))

arr-mset mod-heap *l* (*r* - 1) = *arr-mset new-heap* *l* (*r* - 1)

shows

arr-mset new-heap *l* (*r* - 1) = *arr-mset heap* *l* *r* - {#*Min-mset* (*arr-mset heap* *l* *r*)#}

<proof>

del-min removes the minimum element from the heap by replacing the first element with the last element, shrinking the array by one and subsequently restoring the heap invariant with *sift-down*.

procedure-spec *del-min* (*heap*, *l*, *r*) **returns** (*heap*, *l*, *r*)

assumes $l < r \wedge$ *is-heap* (\leq) *heap* *l* *r*

ensures *is-heap* (\leq) *heap* *l* *r* \wedge

arr-mset heap *l* *r* = *arr-mset heap*₀ *l*₀ *r*₀ - {#*Min-mset* (*arr-mset heap*₀ *l*₀ *r*₀)#} \wedge

$l = l_0 \wedge r = r_0 - 1 \wedge$

*heap*₀ = *heap on* - {*l*₀..*r*₀}

for *heap* *l* *r*

defines \langle

$r = r - 1;$

heap[*l*] = *heap*[*r*];

heap = *sift-down*(*heap*, *l*, *r*, *l*)

\rangle

<proof>

make-heap transforms an arbitrary array into a heap by iterating through all array positions from the middle of the array up to the beginning of the array and calling *sift-down* for each one.

procedure-spec *make-heap* (*heap*, *l*, *r*) **returns** *heap*

assumes $l \leq r$

ensures *is-heap* (\leq) *heap* *l*₀ *r*₀ \wedge

arr-mset heap *l*₀ *r*₀ = *arr-mset heap*₀ *l*₀ *r*₀ \wedge

```

      heap0 = heap on - {l0..< r0}
for heap[] l r
defines <
  y = (r + l)/2 - 1;
  while (y ≥ l)
    @variant <y - l + 1>
    @invariant <is-partial-heap (≤) heap l (y + 1) r ∧
      arr-mset heap l r = arr-mset heap0 l0 r0 ∧
      l - 1 ≤ y ∧ y < r ∧ heap0 = heap on - {l0..<r}>
    {
      heap = sift-down(heap, l, r, y);
      y = y - 1
    }
  >
  <proof>

```

4.3 Heapsort Implementation

The final part of this submission is the implementation of the in-place heapsort. Firstly it builds the \leq -heap and then it iteratively removes the minimum of the heap, which is put at the now vacant end of the shrinking heap. This is done until the heap is empty, which leaves the array sorted in descending order.

4.3.1 Auxiliary Lemmas

Firstly the notion of a sorted array is needed. This is more or less the same as *ran-sorted* generalized for arbitrary comparison functions.

definition *array-is-sorted* :: (int ⇒ int ⇒ bool) ⇒ (int ⇒ int) ⇒ int ⇒ int ⇒ bool **where**

array-is-sorted cmp a l r ≡ ∀ i. ∀ j. bounded l r i → bounded l r j → i < j → cmp (a i) (a j)

This lemma states, that the heapsort doesn't change the elements contained in the array during the loop iterations.

lemma *heap-sort-mset-step*:

fixes arr::int ⇒ int

assumes

l < m m ≤ r

arr-mset arr' l (m - 1) = arr-mset arr l m - {#Min-mset (arr-mset arr l m)#}

arr = arr' on - {l..<m}

mod-arr = arr'(m - 1 := Min-mset (arr-mset arr l m))

shows arr-mset arr l r = arr-mset mod-arr l r

<proof>

This lemma states, that each loop iteration leaves the growing second half of the array sorted in descending order.

lemma *heap-sort-second-half-sorted-step*:

fixes $arr::int \Rightarrow int$

assumes

$l_0 < m \ m \leq r_0$

$arr = arr' \text{ on } - \{l_0..<m\}$

$\forall i. \forall j. \text{bounded } m \ r_0 \ i \longrightarrow \text{bounded } m \ r_0 \ j \longrightarrow i < j \longrightarrow arr \ j \leq arr \ i$

$\forall x \in \#arr\text{-mset } arr \ l_0 \ m. \forall y \in \#arr\text{-mset } arr \ m \ r_0. \neg x < y$

$\text{bounded } (m - 1) \ r_0 \ i$

$\text{bounded } (m - 1) \ r_0 \ j$

$i < j$

$\text{mod-arr} = (arr'(m - 1 := \text{Min-mset } (arr\text{-mset } arr \ l_0 \ m)))$

shows $\text{mod-arr } j \leq \text{mod-arr } i$

<proof>

The following lemma shows that all elements in the first part of the array (the binary heap) are bigger than the elements in the second part (the sorted part) after every iteration. This lemma and the invariant of the *heap-sort* loop use $\neg x < y$ instead of $x \geq y$ since *vcg-cs* doesn't terminate in the latter case.

lemma *heap-sort-fst-part-bigger-snd-part-step*:

fixes $arr::int \Rightarrow int$

assumes

$l_0 < m$

$m \leq r_0$

$arr\text{-mset } arr' \ l_0 \ (m - 1) = arr\text{-mset } arr \ l_0 \ m - \{\#\text{Min-mset } (arr\text{-mset } arr \ l_0 \ m)\#\}$

$arr = arr' \text{ on } - \{l_0..<m\}$

$\forall x \in \#arr\text{-mset } arr \ l_0 \ m. \forall y \in \#arr\text{-mset } arr \ m \ r_0. \neg x < y$

$\text{mod-arr} = arr'(m - 1 := \text{Min-mset } (arr\text{-mset } arr \ l_0 \ m))$

$x \in \#arr\text{-mset } \text{mod-arr} \ l_0 \ (m - 1)$

$y \in \#arr\text{-mset } \text{mod-arr} \ (m - 1) \ r_0$

shows $\neg x < y$

<proof>

4.3.2 Implementation

Now finally the correctness of the *heap-sort* is shown. As mentioned, it starts by transforming the array into a minimum heap using *make-heap*. Then in each iteration it removes the first element from the heap with *del-min* after

its value was retrieved with *get-min*. This value is then put at the position freed by *del-min*.

program-spec *heap-sort*

```

assumes  $l \leq r$ 
  ensures array-is-sorted ( $\geq$ ) arr  $l_0$   $r_0$   $\wedge$ 
     $arr\text{-}mset\ arr_0\ l_0\ r_0 = arr\text{-}mset\ arr\ l_0\ r_0 \wedge$ 
     $arr_0 = arr\ on - \{l_0 ..<r_0\} \wedge l = l_0 \wedge r = r_0$ 
for  $l\ r\ arr[]$ 
defines  $\langle$ 
  arr = make-heap(arr,  $l$ ,  $r$ );
   $m = r$ ;
  while ( $m > l$ )
    @variant  $\langle m - l + 1 \rangle$ 
    @invariant  $\langle is\text{-}heap\ (\leq)\ arr\ l\ m \wedge$ 
      array-is-sorted ( $\geq$ ) arr  $m\ r_0 \wedge$ 
       $(\forall x \in\# arr\text{-}mset\ arr\ l_0\ m. \forall y \in\# arr\text{-}mset\ arr\ m\ r_0. \neg x < y) \wedge$ 
       $arr\text{-}mset\ arr_0\ l_0\ r_0 = arr\text{-}mset\ arr\ l_0\ r_0 \wedge$ 
       $l \leq m \wedge m \leq r_0 \wedge l = l_0 \wedge arr_0 = arr\ on - \{l_0 ..<r_0\} \rangle$ 
    {
       $min = get\text{-}min(arr, l, m)$ ;
       $(arr, l, m) = del\text{-}min(arr, l, m)$ ;
       $arr[m] = min$ 
    }
   $\rangle$ 
   $\langle proof \rangle$ 
end

```

References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. 2009.
- [2] P. Lammich and S. Wimmer. IMP2 – Simple Program Verification in Isabelle/HOL. *Archive of Formal Proofs*, Jan. 2019. <http://isa-afp.org/entries/IMP2.html>, Formal proof development.
- [3] K. Mehlhorn and P. Sanders. Algorithms and Data Structures: The Basic Toolbox. 2007.
- [4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, page 184. 2018. <http://isabelle.in.tum.de/dist/Isabelle2018/doc/tutorial.pdf>.