Zadání bakalářské práce

Název:	Ověřená binární halda
Student:	Luboš Zápotočný
Vedoucí:	doc. RNDr. Dušan Knop, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Jednou věcí je napsat kód, který má nějaký účel. Jinou věcí ale je mít kód, který zaručeně dělá to, co má a jen to, co má. Jedním z prostředků jak toto o svém kódu zaručit je použití anotačního jazyka ACSL a tvořit takzvaný verifikovaný kód. Jazyk ACSL nám umožňuje pomocí anotací u jednotlivých funkcí specifikovat jejich očekávané chování. Tyto anotace společně s kódem (napsaným v jazyce C) lze externím nástrojem (Frama-C) zpracovat a následně v tomto nástroji provést formální důkaz, že kód splňuje formální definice pro dané funkce.

Cílem práce je seznámit se s jazykem ACSL a vytvořit verifikovanou implementaci minimové binární haldy a použít ji v Dijkstrově algoritmu pro hledání nejkratší cesty v grafu.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 15. února 2022 v Praze.

Bakalářská práce

OVĚŘENÁ BINÁRNÍ HALDA

Luboš Zápotočný

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: doc. RNDr. Dušan Knop, Ph.D.
12. května 2022

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2022 Luboš Zápotočný. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Zápotočný Luboš. *Ověřená binární halda.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Obsah

Poděkování v

Prohlášení vi

Abstrakt vii

Seznam zkratek viii

Úvod 1

1 Binární halda 2

2 ACSL a verifikační prostředí 4
- 2.1 ACSL . . . 4
- 2.2 WP . . . 4
- 2.3 RTE . . . 4
- 2.4 Smoke testy . . . 5

3 Implementace důkazu binární haldy 6
- 3.1 Hledání minimálního prvku . . . 8
- 3.2 Probublání prvku nahoru . . . 10
- 3.3 Vložení prvku . . . 15
- 3.4 Probublání prvku dolů . . . 18
- 3.5 Odstranění minimálního prvku . . . 23
- 3.6 Změna hodnoty prvku . . . 25
  - 3.6.1 Snížení hodnoty prvku . . . 27
  - 3.6.2 Zvýšení hodnoty prvku . . . 28
- 3.7 Konstrukce haldy . . . 29

4 Metody ladění při vývoji důkazu 32
- 4.1 Zúžení vstupních podmínek . . . 32
- 4.2 Kontradikce . . . 33

5 Hledání nejkratší cesty 34
- 5.1 Dijkstrův algoritmus . . . 34

6 Vývojové prostředí 35
- 6.1 Docker . . . 35

Závěr 36

Obsah přiloženého média 38

Seznam obrázků

1.1 Ukázka binární minimové haldy . . . 2
1.2 Ukázka uložení binární minimové haldy v poli . . . 3

3.1 Úspešné dokončení důkazu hledání minimálního prvku v prostředí Frama-C . . . 10
3.2 Ukázka intuitivního řezu haldy . . . 11
3.3 Ukázka vrchního řezu haldy pomocí potomka . . . 12
3.4 Ukázka spodního řezu haldy pomocí potomka . . . 13
3.5 Úspešné dokončení důkazu probublání nahoru v prostředí Frama-C . . . 15
3.6 Ukázka vrchního řezu haldy pomocí potomka při přidání vrcholu . . . 16
3.7 Úspešné dokončení důkazu vložení prvku do haldy v prostředí Frama-C . . . 17
3.8 Ukázka vrchního řezu haldy pomocí rodiče . . . 19
3.9 Ukázka spodního řezu haldy pomocí rodiče . . . 20
3.10 Úspešné dokončení důkazu probublání dolů v prostředí Frama-C . . . 22
3.11 Ukázka spodního řezu haldy pomocí rodiče v průběhu odstraňování minimálního prvku . . . 23
3.12 Úspešné dokončení důkazu odstranění minimálního prvku z haldy v prostředí Frama-C . . . 25
3.13 Úspešné dokončení důkazu změny hodnoty prvku haldy v prostředí Frama-C . . . 26
3.14 Úspešné dokončení důkazu snížení hodnoty prvku haldy v prostředí Frama-C . . . 28
3.15 Úspešné dokončení důkazu zvýšení hodnoty prvku haldy v prostředí Frama-C . . . 29

4.1 Testování platnosti predikátu IsDescendant v prostředí Frama-C . . . 33

Seznam výpisů kódu

1 ACSL predikát validní haldy . . . 7
2 Příkaz pro spuštění kompletního důkazu knihovny binární minimové haldy . . . 7
3 Výstup spuštění kompletního důkazu knihovny binární minimové haldy . . . 8
4 Hledání minimálního prvku . . . 9
5 Hledání minimálního prvku . . . 9
6 Probublání prvku nahoru . . . 11
7 Predikát validního vrchního řezu v haldě pomocí potomka . . . 12
8 Predikát validního spodního řezu v haldě pomocí potomka . . . 13
9 Predikáty tranzitivního haldového uspořádání mezi řezy haldou . . . 14
10 Vložení prvku . . . 17
11 Probublání prvku dolů . . . 18
12 Predikát validního vrchního řezu v haldě pomocí rodiče . . . 19
13 Predikát validního spodního řezu v haldě pomocí rodiče . . . 20

14 Kód a ACSL anotace odstranění minimálního prvku z haldy 24
15 Kód a ACSL anotace změny hodnoty prvku haldy 25
16 Datová struktura prvku haldy 26
17 Kód a ACSL anotace prohození dvou prvků v hladě 27
18 Kód a ACSL anotace snížení hodnoty prvku v hladě 28
19 Kód a ACSL anotace zvýšení hodnoty prvku v hladě 29
20 Kód a ACSL anotace konstrukce haldy 30
21 ACSL predikát tranzitivní vlastnosti být potomkem 32
22 ACSL kontrola kontradikce 33

Seznam pseudokódů

1 Pseudokód Dijkstrova algoritmu s binární haldou 34

Chtěl bych poděkovat doc. RNDr. Dušan Knop, Ph.D. za odborné vedení bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 12. května 2022 ....

Abstrakt

Tato práce popisuje implementaci binární minimové haldy a formální důkazy korektnosti jednotlivých operací. Ověřená binární halda je následně použita v Dijkstrově algoritmu pro hledání nejkratší cesty v grafu.

Klíčová slova Binární halda, Frama-C, ACSL, Dijkstrův algoritmus, Docker

Abstract

This thesis describes an implementation of the binary minimum heap and formal proofs of the correctness of individual operations. The verified binary heap is then used in Dijkstra's algorithm for finding the shortest path in a graph.

Keywords Binary heap, Frama-C, ACSL, Dijkstra's algorithm, Docker

Seznam zkratek

ACSL	ANSI/ISO C Specification Language
Frama-C	Framework for Modular Analysis of C programs

Úvod

Vývoj softwaru počítače doprovází také neúmyslně vznikající chyby. Nejznámější softwarové chyby způsobily například havárii nosné rakety Ariane 5 při zkušebním letu v roce 1996, ztrátu satelitu Mars Climate Orbiter nebo smrt šesti pacientů zdravotnického přístroje Therac-25. Všechny tyto a další podobné případy spojuje chybně naprogramovaný software počítače.

Aktivita, která má za účel minimalizovat výskyt těchto chyb se nazývá softwarové testování. Principem tohoto testování je vyzkoušet naprogramovaný algoritmus v kontrolovaném prostředí na velkém množství kombinací vstupních parametrů a zajistit, že kód generuje správné výsledky. Slavný citát Edsgera Wybe Dijkstry zní: „Testováním programu lze prokázat existenci chyby, nikoli její nepřítomnost“. V praxi se často používá metoda testování softwaru, která pokrývá pouze podmnožinu všech možných kombinací vstupních parametrů. Tento styl testování softwaru zajišťuje primárně kontrolu funkčnosti při následné modifikaci zdrojového kódu. „Test driven development“ označuje metodu vývoje softwaru, při které jsou testovací scénáře vytvářeny dříve, než samotný kód. Tato metoda zajišťuje pokrytí testovacími scénáři velké části výsledného software, ale stále nezajišťuje korektnost daného algoritmu. [1]

Statická analýza kódu kontroluje zdrojový kód algoritmu a z kódu odvozuje některé vlastnosti bez nutnosti spuštění daného kódu. Formální analýza je rozšířením statické analýzy o možnost definovat vlastnosti vstupních parametrů, výstupních hodnot a možnost provádět induktivní důkazy za pomocí cyklů. Tato metodika testování korektnosti algoritmů bude v této práci aplikována na datovou strukturu binární minimové haldy.

Binární minimová halda je jedna ze základních datových struktur. Halda se používá například k implementaci prioritní fronty. Jednotlivé prvky jsou tedy řazeny dle jejich priority. Tato struktura lze následně použít v Dijkstrově algoritmu pro hledání nejkratší cesty v grafu.

Tato práce zkoumá jednotlivé operace nad binární minimovou haldou a vysvětluje postup tvorby formálních důkazů v jazyce ACSL. Hlavním cílem této práce je vytvoření kompletní softwarové knihovny v programovacím jazyce C implementující operace nad binární minimovou haldou, opatřit jednotlivé operace v této knihovně ACSL anotacemi a provést formální důkaz korektnosti daných algoritmů v prostředí Frama-C.

Kapitola 1
Binární halda

Tato kapitola shrnuje základní pojmy teorie grafů, které tvoří podklady pro následující kapitoly. V průběhu textu je pojem vrchol haldy a prvek haldy považován za ekvivalentní.

▶ **Definice 1.1** (Souvislý graf)**.** *Graf $G = (V, E)$ je souvislý, jestliže v něm pro každé jeho dva vrcholy u, v existuje u-v-cesta.*

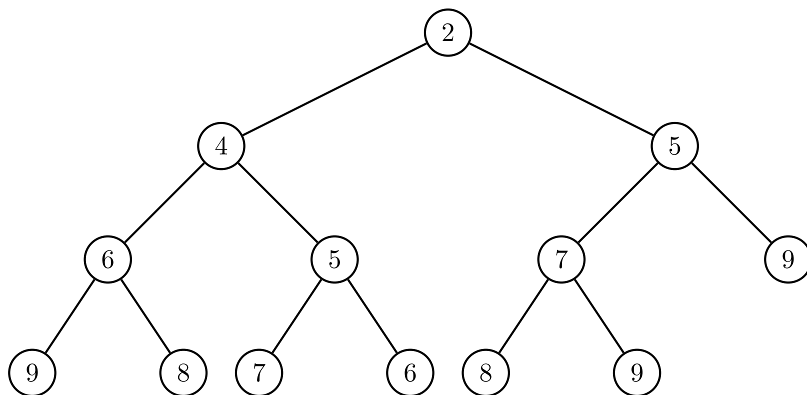
▶ **Definice 1.2** (Strom)**.** *Graf $G = (V, E)$ nazveme stromem, pokud je souvislý a neobsahuje žádnou kružnici (čili je acyklický).*

▶ **Definice 1.3** (Binární strom)**.** *Strom nazveme binární, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý. Vrcholy rozdělíme podle vzdálenosti od kořene do hladin: v nulté hladině leží kořen, v první jeho synové atd.*

▶ **Definice 1.4** (Binární minimová halda)**.** *Binární minimová halda je datová struktura tvaru binárního stromu, v jehož každém vrcholu je uložen jeden prvek. Přitom platí:*

Tvar haldy: Všechny hladiny kromě poslední jsou plně obsazené. Poslední hladina je zaplněna zleva.

Haldové uspořádání: Je-li u vrchol a v jeho syn, platí: $\mathrm{Value}(u) \leq \mathrm{Value}(v)$.



■ **Obrázek 1.1** Ukázka binární minimové haldy

Obrázek 1.1 zobrazuje ukázku binární minimové haldy. Haldu lze uložit do lineárního pole, kde kořen haldy je uložen na indexu 0. Halda z obrázku 1.1 implementovaná pomocí pole je vidět na obrázku 1.2. Průchod haldou lze realizovat následujícími aritmetickými operacemi:

$\text{Parent}(i) = \lfloor (i-1)/2 \rfloor$

$\text{LeftChild}(i) = 2i + 1$

$\text{RightChild}(i) = 2i + 2$

0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	5	6	5	7	9	9	8	7	6	8	9

Obrázek 1.2 Ukázka uložení binární minimové haldy v poli

Kapitola 2

ACSL a verifikační prostředí

2.1 ACSL

ACSL je akronym pro „ANSI/ISO C Specification Language“. Algoritmy v jazyce C lze anotovat pomocí jazyka ACSL a následně provádět statickou analýzu a formální verifikaci korektnosti daného kódu. Anotace je speciální typ komentáře v jazyce C, který začíná znakem „@“. Jednořádkové anotace začínají znaky „//@“. Víceřádkové anotace začínají znaky „/*@“ a končí znaky „*/“. [2]

Mezi časté anotace patří *kontrakt funkce*, který popisuje chování dané funkce. Kontrakt funkce nejčastěji obsahuje popis vstupních podmínek, výstupních ujištění a popis míst v paměti, která mohou být modifikována voláním této funkce. Další speciální anotací je invariant cyklu, který pomáhá deduktivní analýze dokončit induktivní důkaz platnosti specifikované vlastnosti plynoucí z průběhu cyklu.

ACSL anotace některých vlastností lze seskupit do predikátu. Součástí predikátu jsou vstupní parametry predikátu a definice vlastností těchto parametrů. Predikát je tedy platný pouze v případě, kdy platí všechny z definovaných vlastností uvnitř predikátu. Takto vytvořený predikát lze následně používat v ostatních ACSL anotacích. [2]

2.2 WP

Weakest precodition plugin verifikačního prostředí Frama-C používá externí rozhodovací nástroje pro dokázání platnosti ACSL anotací v programovacím jazyce C. [3]

Hoareova trojice je způsob popisu chování algoritmu. Tato trojice definuje výstupní vlastnosti proměnných, paměti a návratové hodnoty po provedení algoritmu, za předpokladu, že vstupní podmínky byly splněny. Obecně lze takovou trojici zapsat jako „P stmt Q“. Kde P jsou vstupní podmínky algoritmu, *stmt* je obsah funkce implementující daný algoritmus a Q jsou výstupní vlastnosti, které budou po ukončení volání funkce platné. [3]

2.3 RTE

Runtime Error Annotation Generation plugin verifikačního prostředí Frama-C automaticky generuje ACSL anotace pro běžné chyby, které se mohou vyskytnou až při běhu programu. Například dělení nulou, přetečení číselného typu nebo přístup či zápis na nepovolené místo v paměti počítače. [4]

RTE anotace nemusejí být součástí zdrojového kódu, ale mohou být vygenerovány při spuštění grafického prostředí nebo pomocí příkazové řádky přepínačem „-rte“.

2.4 Smoke testy

Smoke testy jsou speciální součástí kontroly zdrojového kódu pomocí ACSL anotací. Vstupní podmínky definované anotacemi ACSL mohou zapříčinit, že Frama-C a WP plugin dokáží dokončit formální důkaz některé vlastnosti algoritmu, které není možné dosáhnout, jelikož jsou vstupní podmínky nesplnitelné. Základním postupem pro kontrolu nesplnitelnosti vstupních podmínek je kontrola vstupních podmínek a invariantů cyklů za pomoci testu dokazatelnosti kontradikce. Tyto testy selhávají s chybou v případě, když WP plugin v některém bodě programu úspěšně dokáže splnitelnost „false“. [3]

Kapitola 3

Implementace důkazu binární haldy

Tvar haldy umožňuje haldu efektivně uložit do lineárního pole a jednotlivé vrcholy očíslovat indexy pole. Tato ověřená implementace používá haldu s kořenem na indexu 0. Haldu uloženou v poli lze procházet pomocí následujících aritmetických operací:

$\mathrm{Parent}(i) = \lfloor (i-1)/2 \rfloor$

$\mathrm{LeftChild}(i) = 2i + 1$

$\mathrm{RightChild}(i) = 2i + 2$

Tyto operace jsou implementované pomocí logických funkcí v ACSL anotacích a zároveň jsou vytvořené ekvivalentní funkce v jazyce C. Velká část použitých logických funkcí nebo predikátů má také oddělené funkce v jazyce C pro zajištění konzistence s ACSL anotacemi. Některé z pomocných funkcí nejsou v textu explicitně zmíněné nebo ukázané, jelikož práce popisuje důkazní postupy hlavních operací nad haldou. Pomocné funkce, které nebudou v práci zmíněné mají často jednořádkovou implementaci a mají dostatečně vypovídající jména, aby bylo možné dané funkce používat v ukázkách kódu. Tyto pomocné funkce také obsahují ACSL anotace, aby bylo zajištěno, že všechny funkce v knihovně jsou formálně ověřené. Kompletní zdrojový kód knihovny je k dispozici na přiloženém médiu.

Binární halda musí splňovat tvar haldy a haldové uspořádání. Tvar haldy není v důkazech formálně dokazován, jelikož je použita implementace haldy pomocí pole. Predikát o správném haldovém uspořádání, zobrazený ve výpisu kódu 1, je základním predikátem, který určuje, zda v haldě platí haldové uspořádání. Tento predikát je vstupní podmínkou pro některé operace nad haldou a všechny operace nad haldou až na částečnou opravu pomocí probublání dolů, o kterém pojednává sekce 3.4, tento predikát splňují také jako výstupní ujištění korektnosti algoritmu.

```
/*@
  predicate HasHeapProperty(Heap heap, integer parent, integer child) =
    HeapElementValue(heap, parent) <= HeapElementValue(heap, child);

  predicate ValidHeap(Heap heap) =
    \forall integer ancestor, descendant;
      0 <= ancestor < descendant < HeapElementsCount(heap)
      && IsParent(ancestor, descendant) ==>
        HasHeapProperty(heap, ancestor, descendant);
*/
```

Výpis kódu 1 ACSL predikát validní haldy

Predikát o správném haldovém uspořádání z výpisu kódu 1 začíná univerzálním kvantifikátorem ∀, ve kterém se vyskytují vždy dvě celá čísla *ancestor* a *descendant* reprezentující indexy vrcholů haldy implementované pomocí pole. Pokud platí, že vrchol s indexem *ancestor* je rodičovským vrcholem vrcholu s indexem *descendant*, musí mezi těmito vrcholy platit haldové uspořádání. Tento predikát popisuje haldové uspořádání dle definice 1.4 binární minimové haldy.

Výsledná implementace binární minimové haldy obsahuje téměř 500 cílů pro důkaz korektnosti, splňuje korektní chování při použití RTE anotací, neobsahuje kontradikci ve vstupních podmínkách a neobsahuje nedosažitelný kód. RTE anotace se generují při spuštění kontroly korektnosti pomocí přepínače „-rte“. Kontradikce vstupních podmínek může způsobit úspěšné dokázání korektnosti funkce, která ale nikdy nepůjde spustit, jelikož nelze splnit dané vstupní podmínky. Tuto kontrolu provádí „Smoke testy“, které lze spustit pomocí přepínače „-wp-smoke-tests“.

Formální důkaz korektnosti knihovny lze spustit pomocí příkazové řádky nebo pomocí grafického prostředí programu Frama-C. V rámci této práce vznikl konfigurační soubor pro Docker, který umožňuje vývojáři editovat kód, psát ACSL anotace a spouštět formální verifikaci v grafickém prostředí Frama-C pomocí webového prohlížeče. Vývojové prostředí popisuje kapitola 6.

Kompletní důkaz korektnosti celé knihovny binární minimové haldy lze spustit v příkazové řádce příkazem zobrazeným ve výpisu kódu 2.

```
frama-c -rte -wp -wp-prover alt-ergo,cvc4 \
  -wp-par 8 -wp-cache none -wp-timeout 30 \
  -wp-smoke-tests -wp-smoke-timeout 30 \
  src/min_heap.c
```

Výpis kódu 2 Příkaz pro spuštění kompletního důkazu knihovny binární minimové haldy

Výpis kódu 3 zobrazuje výsledek provedení kompletního formálního důkazu knihovny binární minimové haldy. Tento výpis zobrazuje seznam všech funkcí, ve kterých byly automaticky vygenerované RTE anotace společně s počtem nalezených cílů, které mají být v průběhu provádění důkazu korektnosti splněny. Knihovna splňuje všechny zapsané a vygenerované cíle. Výsledkem je tedy korektní implementace binární minimové haldy.

```
...
[rte] annotating function HeapBubbleDown
[rte] annotating function HeapBubbleUp
[rte] annotating function HeapBuild
[rte] annotating function HeapChange
[rte] annotating function HeapDecrease
[rte] annotating function HeapElementValue
[rte] annotating function HeapExternalNodeCount
[rte] annotating function HeapExtractMin
[rte] annotating function HeapFindMin
[rte] annotating function HeapHasBothChildren
[rte] annotating function HeapHasChild
[rte] annotating function HeapHasLeftChild
[rte] annotating function HeapHasParent
[rte] annotating function HeapHasRightChild
[rte] annotating function HeapIncrease
[rte] annotating function HeapInsert
[rte] annotating function HeapInternalNodeCount
[rte] annotating function HeapLeftChild
[rte] annotating function HeapLowerChild
[rte] annotating function HeapParent
[rte] annotating function HeapRightChild
[rte] annotating function HeapSwap
[rte] annotating function _HeapElementValue
[rte] annotating function swapHeapElements
[rte] annotating function swapi
[wp] 488 goals scheduled
[wp] Proved goals:  488 / 488
  Qed:             207  (0.30ms-7ms-121ms)
  Alt-Ergo 2.2.0:  242  (20ms-30s) (9312) (failed: 2)
  CVC4 1.7:        185  (30ms-8.8s-30s) (957095)
```

Výpis kódu 3 Výstup spuštění kompletního důkazu knihovny binární minimové haldy

Jednotlivé hlavní funkce jsou popsány v následujících kapitolách. Kompletní kód knihovny je přiložen na médiu společně s konfigurací Docker vývojového prostředí.

3.1 Hledání minimálního prvku

Hledání minimálního prvku je operace nad haldou, která dokáže v čase $\mathcal{O}(1)$ nalézt prvek s minimální hodnotou v haldě.

Haldové uspořádání zajišťuje, že hodnota prvku v rodičovském vrcholu je vždy menší nebo rovna hodnotě prvku ve vrcholu potomka daného vrcholu. Haldové uspořádání je relací částečného uspořádání, proto v této relaci platí tranzitivita. Kořen haldy tedy obsahuje prvek s minimální hodnotou.

Nalezení minimálního prvku znamená získat prvek v haldě implementované polem na indexu kořene. Zdrojový kód a ACSL anotace této funkce zobrazuje výpis kódu 4.

```
/*@
  requires 0 < HeapElementsCount(heap);
  requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
  requires ValidHeap(heap);

  assigns \nothing;

  ensures extreme_exists:
    \exists integer i;
      0 <= i < HeapElementsCount(heap) ==>
        \result == HeapElements(heap)[i];

  ensures correct_extreme:
    \forall integer i;
      0 < i < HeapElementsCount(heap) ==>
        HasHeapProperty(heap, 0, i);
*/
HeapElement HeapFindMin(Heap heap) {
  return heap.elements[0];
}
```

Výpis kódu 4 Hledání minimálního prvku

Haldu je možné pomocí operace odstranění minimálního prvku popsaného v sekci 3.5 úplně vyprázdnit. Výpis kódu 4 obsahuje ACSL vstupní podmínku, která kontroluje, že funkce může být volána pouze za předpokladu, že v haldě existuje alespoň jeden prvek.

ACSL anotace také zajišťují, že nalezený prvek je doopravdy minimálním prvkem z celé haldy. Pro dokončení důkazu korektnosti byl zaveden axiom o existenci minimálního prvku, který zobrazuje výpis kódu 5. Celá implementace korektní binární minimové haldy používá pouze *Alt-Ergo* a *CVC4* externí dokazovací nástroje. Tyto nástroje nejsou tento axiom schopny dokázat. *Z3* externí dokazovací nástroj je schopný tento axiom dokázat, ale jelikož by jeho využití bylo opodstatněné pouze v tomto případě, byl zaveden tento axiom místo lemma, které by bylo možné dokázat pomocí nástroje Z3. Tento axiom následné pomáhá při dokončení důkazu korektnosti algoritmu hledání minimálního prvku v haldě.

```
/*@
  // alt-ergo or cvc4 are not able to prove this implication on more
  // than ~80 elements. Z3 is able to prove this implication, but
  // having no need for Z3 in whole codebase, axiom was chosen to keep
  // code simplified

  axiomatic heap_structure_and_heap_property {
    axiom root_is_extreme:
      \forall Heap heap;
        ValidHeap(heap) ==>
          \forall integer index;
            0 <= index < HeapElementsCount(heap) ==>
              HasHeapProperty(heap, 0, index);
  }
*/
```

Výpis kódu 5 Hledání minimálního prvku

Axiom ve výpisu kódu 5 zajišťuje, že pokud platí v haldě haldové uspořádání, tak kořen haldy obsahuje minimální prvek.

```
/*@ requires 0 < HeapElementsCount(heap);
    requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
    requires ValidHeap(heap);
    ensures
      extreme_exists:
        ∃ ℤ i;
          0 ≤ i < HeapElementsCount(\old(heap)) →
          \result ≡ *(HeapElements(\old(heap)) + i);
    ensures
      correct_extreme:
        ∀ ℤ i;
          0 < i < HeapElementsCount(\old(heap)) →
          HasHeapProperty(\old(heap), 0, i);
    assigns \nothing;
 */
HeapElement HeapFindMin(Heap heap)
{
  HeapElement __retres;
  /*@ assert rte: mem_access: \valid_read(heap.elements + 0); */
  __retres = *(heap.elements + 0);
  return __retres;
}
```

Obrázek 3.1 Úspešné dokončení důkazu hledání minimálního prvku v prostředí Frama-C

Na obrázku 3.1 je vidět úspěšně dokončený důkaz korektnosti hledání minimálního prvku v haldě.

Ostatní funkce a anotace v implementaci binární haldy nevyužívají axiomy a lze dokázat jejich korektnost pomocí nástrojů *Alt-Ergo* nebo *CVC4*.

3.2 Probublání prvku nahoru

Probublání prvku haldy směrem nahoru (ke kořeni) je operace nad haldou, která dokáže v čase $\mathcal{O}(\log(n))$ opravit haldové uspořádání za předpokladu, že se snížila pouze hodnota prvku uloženého ve vrcholu, který má být probublán.

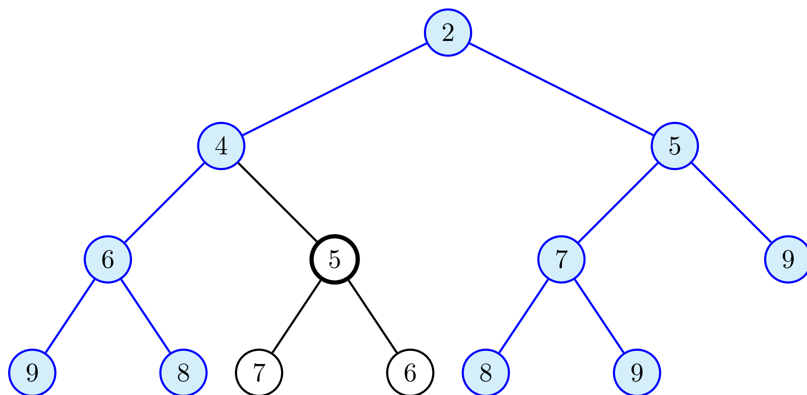
Algoritmus předpokládá, že haldové uspořádání lze porušit pouze mezi jednou dvojicí prvků ve vrcholech u a v, kde u je rodičovský vrchol a v je jeho potomek. Tedy v celé haldě s výjimkou této dvojice musí platit haldové uspořádání. V této dvojici může platit, že prvek ve vrcholu v má menší hodnotu než prvek ve vrcholu u. Dále se předpokládá, že tento algoritmus bude proveden po zmenšení hodnoty některého prvku haldy nebo po vložení nového prvku do haldy. V obou případech je potřeba daný prvek probublat nahoru do správné pozice v haldě. Pokud by se hodnota prvku zvýšila, měl by být aplikován algoritmus probublání dolů, o kterém pojednává sekce 3.4.

```
void HeapBubbleUp(Heap heap, int index) {
  int parent;

  while (HeapHasParent(heap, index)) {
    parent = HeapParent(index);

    if (HeapElementValue(heap, parent) <= HeapElementValue(heap, index)) {
      break;
    }

    HeapSwap(heap, index, parent);

    index = parent;
  }
}
```

Výpis kódu 6 Probublání prvku nahoru

Pro důkaz korektnosti algoritmu probublání nahoru, zobrazeného ve výpisu kódu 6, jsou použity *řezy* haldou. Tyto řezy slouží pro rozdělení grafu haldy na dva podgrafy, ve kterých platí haldové uspořádání. Řezy dohromady tvoří původní graf haldy bez jedné problematické hrany (u, v), na které nemusí platit haldové uspořádání.

Rozdělení haldy pro operaci probublání nahoru by mohlo vypadat podobně jak znázorňuje obrázek 3.2. Tento přístup ale není vhodný pro algoritmické zpracování a dokazovaní, jelikož se toto rozdělení složitě popisuje. Zavedeme proto horní a spodní řez haldy pomocí potomka, které pracují s indexy jednotlivých vrcholů haldy reprezentované polem.



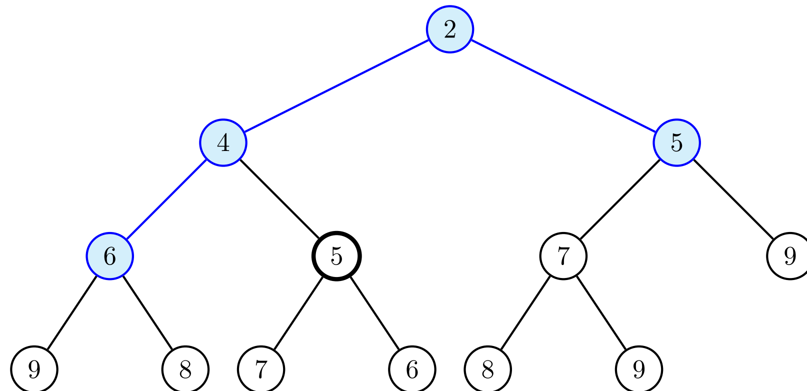
Obrázek 3.2 Ukázka intuitivního řezu haldy

▶ **Definice 3.1** (Vrchní řez haldy pomocí potomka)**.** *Mějme graf $G = (V, E)$ reprezentující binární haldu a $i \in \mathbb{N}$. Vrchním řezem haldy pomocí potomka nazveme graf $G' = (V', E')$, kde*

$$V' = \{v : v \in V \wedge \text{Index}(v) < i\}$$

$$E' = \{(u, v) : (u, v) \in E \wedge u \in V' \wedge v \in V'\}.$$

Obrázek 3.3 zobrazuje ukázku vrchního řezu haldy pomocí potomka pro index 4 na kterém se aktuálně nachází prvek s hodnotou 5.



Obrázek 3.3 Ukázka vrchního řezu haldy pomocí potomka

▶ Pozorování 3.2. V grafu vrchního řezu haldy pomocí potomka platí haldové uspořádání.

ACSL predikát zobrazený ve výpisu kódu 7 popisuje vrchní řez haldy pomocí potomka. Tento predikát nevytváří nový podgraf s výše definovanými vlastnostmi, ale pouze ověřuje, že daná část (podgraf) haldy splňuje haldové uspořádání. Takto vytvořený predikát je následně použit v kontraktu funkce jako jedna ze vstupních podmínek algoritmu probublání nahoru a je také použit jako invariant cyklu. Predikát je tedy platný při volání funkce, po každém kroku cyklu a je také platný na konci vykonávání funkce. Tato induktivní vlastnost následně napomáhá při dokončení důkazu korektnosti algoritmu. Nemusí ale platit, že graf vrchního řezu haldy pomocí potomka je na konci vykonávání funkce prázdný. Tato situace nastává pouze v případě, když je problematický prvek probublán až do kořene haldy.

```
/*@
  predicate HeapUpperChildCut(Heap heap, integer index) =
    \forall integer ancestor, descendant;
      0 <= ancestor < descendant < HeapElementsCount(heap)
      && descendant < index
      && IsParent(ancestor, descendant) ==>
        HasHeapProperty(heap, ancestor, descendant);
*/
```

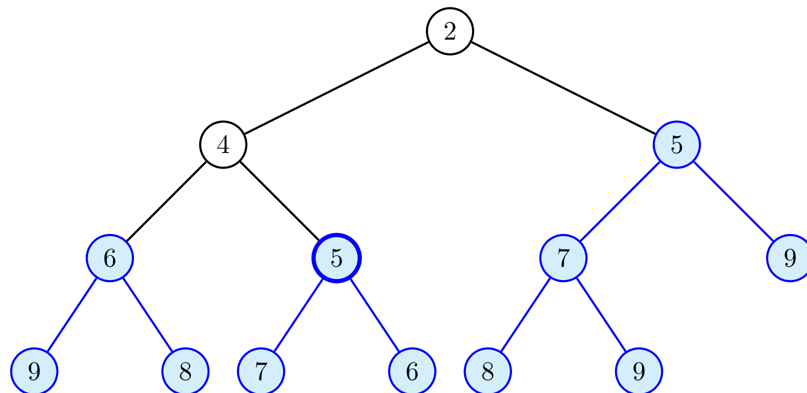
Výpis kódu 7 Predikát validního vrchního řezu v haldě pomocí potomka

▶ **Definice 3.3** (Spodní řez haldy pomocí potomka)**.** *Mějme graf $G = (V, E)$ reprezentující binární haldu a $i \in \mathbb{N}$. Spodním řezem haldy pomocí potomka nazveme graf $G' = (V', E')$, kde*

$$V' = \{v : v \in V \land \text{Index}(v) > i\} \cup \{\text{Parent}(v) : v \in V \land \text{Index}(v) > i\}$$

$$E' = \{(u, v) : (u, v) \in E \land u \in V' \land v \in V'\}.$$

Obrázek 3.4 zobrazuje ukázku spodního řezu haldy pomocí potomka pro index 4 na kterém se nachází prvek s hodnotou 5.



Obrázek 3.4 Ukázka spodního řezu haldy pomocí potomka

► Pozorování 3.4. V grafu spodního řezu haldy pomocí potomka platí haldové uspořádání.

ACSL predikát zobrazený ve výpisu kódu 8 popisuje spodní řez haldy pomocí potomka. Tento predikát je velmi podobný predikátu vrchního řezu haldy pomocí potomka. Pokrývá ale vrcholy haldy s indexem větším než problematický vrchol a jejich rodičovské vrcholy. Nemusí ale platit, že graf spodního řezu haldy pomocí potomka na konci vykonávání funkce obsahuje celou původní haldu. Tato situace nastává pouze v případě, když je problematický prvek probublán až do kořene haldy.

```
/*@
  predicate HeapLowerChildCut(Heap heap, integer index) =
    \forall integer ancestor, descendant;
      0 <= ancestor < descendant < HeapElementsCount(heap)
      && index < descendant
      && IsParent(ancestor, descendant) ==>
        HasHeapProperty(heap, ancestor, descendant);
*/
```

Výpis kódu 8 Predikát validního spodního řezu v haldě pomocí potomka

ACSL predikát spodního řezu haldy pomocí potomka správně odděluje vrcholy haldy, které mají index větší než problematický vrchol. Zároveň ale neomezuje indexy rodičovských vrcholů. Proto je ve spodním řezu pomocí potomka správně obsažen problematický vrchol jako rodičovský vrchol a také některé rodičovské vrcholy, které mají menší index než problematický vrchol. Hrana mezi problematickým vrcholem a jeho rodičovským vrcholem zde správně není obsažena.

Poslední nutnou vlastností pro úspěšné dokončení důkazu je tranzitivní haldové uspořádání mezi vrchním a spodním řezem haldy. Tato vlastnost je automaticky splněna pro všechny hrany v haldě s výjimkou hrany, na které leží problematický vrchol v roli potomka. Proto je do kódu zavedena vstupní podmínka a invariantu cyklu, které kontrolují, zda platí haldové uspořádání pro rodičovský vrchol problematického vrcholu a jednotlivé potomky problematického vrcholu.

Predikáty tranzitivního haldového uspořádání zobrazené ve výpisu kódu 9 umožňují probublávat problematický vrchol nahoru. Zaručují totiž, že rodičovský prvek, který s problematickým vrcholem vyměníme, bude po provedení výměny splňovat haldové uspořádání se svými novými potomky.

```
/*@
  predicate HeapCutHeapPropertyLeftChild(Heap heap, integer index) =
    HeapHasParent(heap, index)
    && HeapHasLeftChild(heap, index) ==>
      HasHeapProperty(heap, Parent(index), LeftChild(index));

  predicate HeapCutHeapPropertyRightChild(Heap heap, integer index) =
    HeapHasParent(heap, index)
    && HeapHasRightChild(heap, index) ==>
      HasHeapProperty(heap, Parent(index), RightChild(index));
*/
```

Výpis kódu 9 Predikáty tranzitivního haldového uspořádání mezi řezy haldou

Platnost těchto čtyř invariantů cyklu dokazuje, že algoritmus probublání nahoru vždy vytvoří maximálně jednu novou problematickou dvojici vrcholů. Haldové uspořádání mezi touto dvojicí vrcholů je opraveno probubláním prvku na správnou pozici v haldě.

V prostředí Frama-c lze formálně dokázat korektnost celé funkce probublání prvku nahoru. Jelikož platí výše zmíněné čtyři invarianty cyklu, po ukončení cyklu se halda nachází v jednom z těchto stavů:

1. problematický prvek probublal až do kořene haldy,
2. problematický prvek nemohl dále probublat, tudíž je na správné pozici v haldě.

Spojením platnosti invariantů cyklu a znalostí o aktuálně opravené hraně dává dostatečné důkazy pro verifikační prostředí, že pro každou hranu (u, v) dané haldy, kde u je rodičovský vrchol v, platí haldové uspořádání. Úspěšně dokončený důkaz korektnosti funkce probublání nahoru je zobrazen na obrázku 3.5.

```
/*@ requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
    requires 0 ≤ index < HeapElementsCount(heap);
    requires HeapUpperChildCut(heap, index);
    requires HeapLowerChildCut(heap, index);
    requires HeapCutHeapPropertyLeftChild(heap, index);
    requires HeapCutHeapPropertyRightChild(heap, index);
    ensures ValidHeap(\old(heap));
    assigns *(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
 */
void HeapBubbleUp(Heap heap, int index)
{
  int parent;
  /*@ loop invariant 0 ≤ index < HeapElementsCount(heap);
      loop invariant HeapUpperChildCut(heap, index);
      loop invariant HeapLowerChildCut(heap, index);
      loop invariant HeapCutHeapPropertyLeftChild(heap, index);
      loop invariant HeapCutHeapPropertyRightChild(heap, index);
      loop assigns index, parent,
                   *(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
      loop variant index;
  */
  while (1) {
    int tmp_1;
    tmp_1 = HeapHasParent(heap,index);
    if (! tmp_1) {
      break;
    }
    {
      int tmp;
      int tmp_0;
      parent = HeapParent(index);
      { /* sequence */
        tmp = HeapElementValue(heap,parent);
        tmp_0 = HeapElementValue(heap,index);
      }
      if (tmp <= tmp_0) {
        break;
      }
      /*@
      assert
      heap_cut_parent_heap_property_right_child:
        IsLeftChild(index, parent) ∧ HeapHasRightChild(heap, parent) →
        HasHeapProperty(heap, parent, RightChild(parent));
       */
      ;
      /*@
      assert
      heap_cut_parent_heap_property_left_child:
        IsRightChild(index, parent) →
        HasHeapProperty(heap, parent, LeftChild(parent));
       */
      ;
      HeapSwap(heap,index,parent);
      index = parent;
    }
  }
  return;
}
```

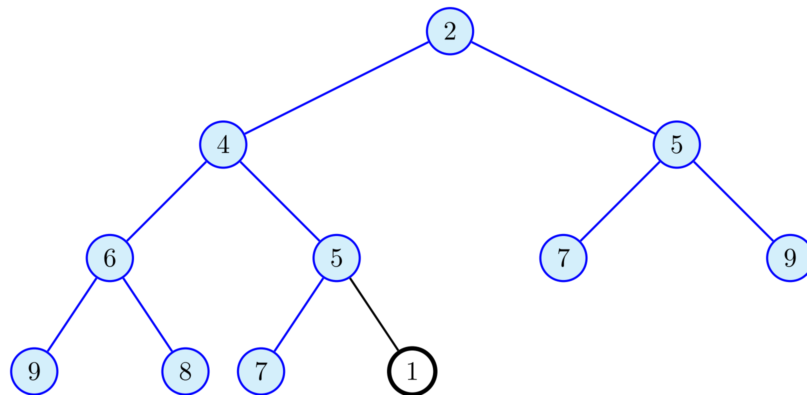
Obrázek 3.5 Úspešné dokončení důkazu probublání nahoru v prostředí Frama-C

3.3 Vložení prvku

Vložení prvku do haldy je operace nad haldou, která v čase $\mathcal{O}(1)$ vloží nový prvek do haldy a následně v čase $\mathcal{O}(\log(n))$ tento prvek umístí na správnou pozici v haldě. Celková asymptotická časová složitost je tedy $\mathcal{O}(\log(n))$.

Algoritmus přidává nový vrchol do nejspodnější hladiny haldy tak, aby byl udržen tvar haldy. Pro takto vložený vrchol nemusí platit haldové uspořádání s jeho rodičovským vrcholem. Mohlo se stát, že hodnota prvku v nově přidaném vrcholu má menší hodnotou než hodnota prvku v jeho rodičovském vrcholu. Proto je nutné provést probublání nahoru, o kterém pojednává sekce 3.2.

Algoritmus probublání nahoru má tři vstupní podmínky. V haldě musí platit predikát o horním řezu haldy pomocí potomka, který v případě přidání nového vrcholu platí. Před přidáním vrcholu byla halda validní a platilo v ní haldové uspořádání. Nový prvek je vložen na nový největší index v haldě. Tedy pro tento nový index platí vrchní řez haldou pomocí potomka, protože tento řez popisuje původní haldu, ve které platilo haldové uspořádání. Spodní řez haldy pomocí potomka také platí, protože se jedná o prázdný graf. Tranzitivní haldové uspořádání u aktuálně přidaného vrcholu také platí, protože tento vrchol nemá žádné potomky.



Obrázek 3.6 Ukázka vrchního řezu haldy pomocí potomka při přidání vrcholu

Na obrázku 3.6 je zobrazeno přidání nového vrcholu do haldy a je zvýrazněn horní řez haldy pomocí potomka.

Algoritmus probublání nahoru tento nový vrchol vymění s jeho rodičovským vrcholem, protože hodnota prvku v nově přidaném vrcholu má menší hodnotu než prvek v jeho rodičovském vrcholu. Tímto prvním krokem algoritmu se nově přidaný vrchol dostal výše v haldě, ale stále platí všechny čtyři invarianty cyklu. Horní řez haldy pomocí potomka se pouze zmenšil a spodní řez haldy pomocí potomka se zvětšil o vrcholy, které se v předchozím kroku vyskytovaly v horním řezu pomocí potomka a také o vrchol původního rodiče nového vrcholu. Tranzitivní haldové uspořádání začíná platit mezi rodičovským prvkem aktuálně probublaného vrcholu a jeho potomky.

Algoritmem probublání se nový vrchol dostane do správné pozice a ve výsledné haldě, o jeden vrchol větší, je opraveno haldové uspořádání.

```
/*@
    requires 0 <= HeapElementsCount(heap) < HeapElementsCapacity(heap);
    requires \valid(HeapElements(heap) + (0 .. HeapElementsCapacity(heap) - 1));
    requires ValidHeap(heap);
    requires correctly_indexed:
            HeapElementIndex(element) == HeapElementsCount(heap);

    assigns HeapElements(heap)[0..HeapElementsCount(heap)];

    ensures count_increase:
            HeapElementsCount(\result) == HeapElementsCount(heap) + 1;
    ensures ValidHeap(\result);
*/
Heap HeapInsert(Heap heap, HeapElement element) {
  int index = heap.elementsCount;

  heap.elements[index] = element;
  heap.elementsCount++;

  HeapBubbleUp(heap, index);

  return heap;
}
```

Výpis kódu 10 Vložení prvku

Výpis kódu 10 zobrazuje funkci pro přidání nového prvku do haldy společně s kontraktem této funkce. ACSL anotace funkce probublání nahoru zajišťují, že po dokončení probublání nahoru je předaná halda opravená a platí v ní haldové uspořádání. Jelikož je volání funkce probublání nahoru poslední akcí funkce vložení prvku, lze ujištění validnosti haldy vložit také do kontraktu funkce vkládání prvku. Tímto je zaručena korektnost funkce vkládání prvku do haldy. Obrázek 3.7 zobrazuje úspěšně dokončený důkaz korektnosti funkce v prostředí Frama-C.

```
/*@ requires 0 ≤ HeapElementsCount(heap) < HeapElementsCapacity(heap);
    requires
      \valid(HeapElements(heap) + (0 .. HeapElementsCapacity(heap) - 1));
    requires ValidHeap(heap);
    requires
      correctly_indexed: HeapElementIndex(element) ≡ HeapElementsCount(heap);
    ensures
      count_increase:
        HeapElementsCount(\result) ≡ HeapElementsCount(\old(heap)) + 1;
    ensures ValidHeap(\result);
    assigns *(HeapElements(heap) + (0 .. HeapElementsCount(heap)));
 */
Heap HeapInsert(Heap heap, HeapElement element)
{
  int index = heap.elementsCount;
  /*@ assert rte: mem_access: \valid(heap.elements + index); */
  *(heap.elements + index) = element;
  /*@ assert rte: signed_overflow: heap.elementsCount + 1 ≤ 2147483647; */
  (heap.elementsCount) ++;
  HeapBubbleUp(heap,index);
  return heap;
}
```

Obrázek 3.7 Úspešné dokončení důkazu vložení prvku do haldy v prostředí Frama-C

3.4 Probublání prvku dolů

Probublání prvku haldy směrem dolů (od kořene) je operace nad haldou, která dokáže v čase $\mathcal{O}(\log(n))$ opravit haldové uspořádání za předpokladu, že se zvýšila pouze hodnota prvku uloženého ve vrcholu, který má být probublán.

Algoritmus předpokládá, že haldové uspořádání lze porušit maximálně u dvou dvojic vrcholů u a v nebo u a w, kde u je rodičovský vrchol vrcholů v a w. Tedy v celé haldě s výjimkou těchto dvou dvojic vrcholů musí platit haldové uspořádání. V těchto dvojicích může platit, že prvek uložený ve vrcholu u má větší hodnotu než některý prvek uložený ve vrcholech v nebo w. Může nastat situace, kdy prvek ve vrcholu u má větší hodnotu než oba prvky ve vrcholech v a w. Dále se předpokládá, že tento algoritmus bude proveden po zvětšení hodnoty některého prvku haldy nebo při vytváření haldy. V obou případech je potřeba tento vrchol probublat dolů do správné pozice v haldě. Pokud by se hodnota snížila, měl by být aplikován algoritmus probublání nahoru, o kterém pojednává sekce 3.2.

```
void HeapBubbleDown(Heap heap, int index) {
    int child;

    while (HeapHasChild(heap, index)) {
        child = HeapLowerChild(heap, index);

        if (HeapElementValue(heap, index) <= HeapElementValue(heap, child)) {
            break;
        }

        HeapSwap(heap, index, child);

        index = child;
    }
}
```

Výpis kódu 11 Probublání prvku dolů

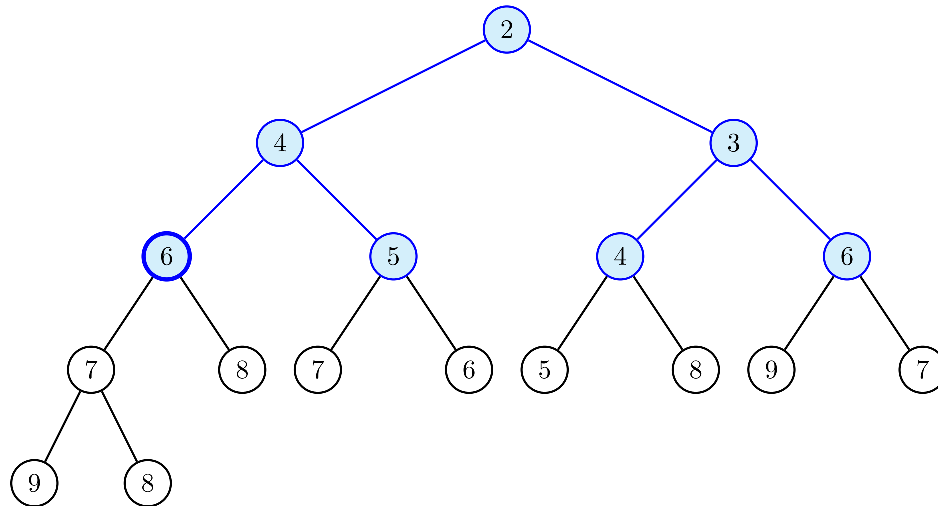
Pro důkaz korektnosti algoritmu probublání dolů, zobrazeného ve výpisu kódu 11, jsou použity *řezy* haldou, podobné řezům v důkazu korektnosti algoritmu probublání nahoru, které byly představeny v sekci 3.2. Řezy haldou pro důkaz probublání dolů jsou mírně odlišné od řezů v důkazu korektnosti algoritmu probublání nahoru. Řezy haldou pomocí rodiče totiž povolují až dvě problematické hrany v haldě.

▶ **Definice 3.5** (Vrchní řez haldy pomocí rodiče)**.** *Mějme graf $G = (V, E)$ reprezentující binární haldu a $i \in \mathbb{N}$. Vrchním řezem haldy pomocí rodiče nazveme graf $G' = (V', E')$, kde*

$V' = \{v : v \in V \land \text{Index}(v) < i\} \cup \{\text{LeftChild}(v) : v \in V \land \text{Index}(v) < i\} \cup \{\text{RightChild}(v) : v \in V \land \text{Index}(v) < i\}$

$E' = \{(u, v) : (u, v) \in E \land u \in V' \land v \in V'\}$.

Obrázek 3.8 zobrazuje ukázku vrchního řezu haldy pomocí rodiče pro index 3 na kterém se nachází prvek s hodnotou 6.



Obrázek 3.8 Ukázka vrchního řezu haldy pomocí rodiče

▶ Pozorování 3.6. V grafu vrchního řezu haldy pomocí rodiče platí haldové uspořádání.

ACSL predikát zobrazený ve výpisu kódu 12 popisuje vrchní řez haldy pomocí rodiče. Tento predikát nevytváří nový podgraf s výše definovanými vlastnostmi, ale pouze ověřuje, že daná část (podgraf) haldy splňuje haldové uspořádání. Takto vytvořený predikát je následně použit v kontraktu funkce jako jedna ze vstupních podmínek algoritmu probublání dolů a je také použit jako invariant cyklu. Predikát je tedy platný při volání funkce, po každém kroku cyklu a je také platný na konci vykonávání funkce. Tato induktivní vlastnost následně napomáhá při dokončení důkazu korektnosti algoritmu. Nemusí ale platit, že graf vrchního řezu haldy pomocí rodiče na konci vykonávání funkce obsahuje celou původní haldu. Tato situace nastává pouze v případě, když je problematický prvek probublán do vrcholu s maximálním indexem v haldě.

```
/*@
  predicate HeapUpperParentCut(Heap heap, integer index) =
    \forall integer ancestor, descendant;
      0 <= ancestor < index
      && ancestor < descendant < HeapElementsCount(heap)
      && IsParent(ancestor, descendant) ==>
        HasHeapProperty(heap, ancestor, descendant);
*/
```

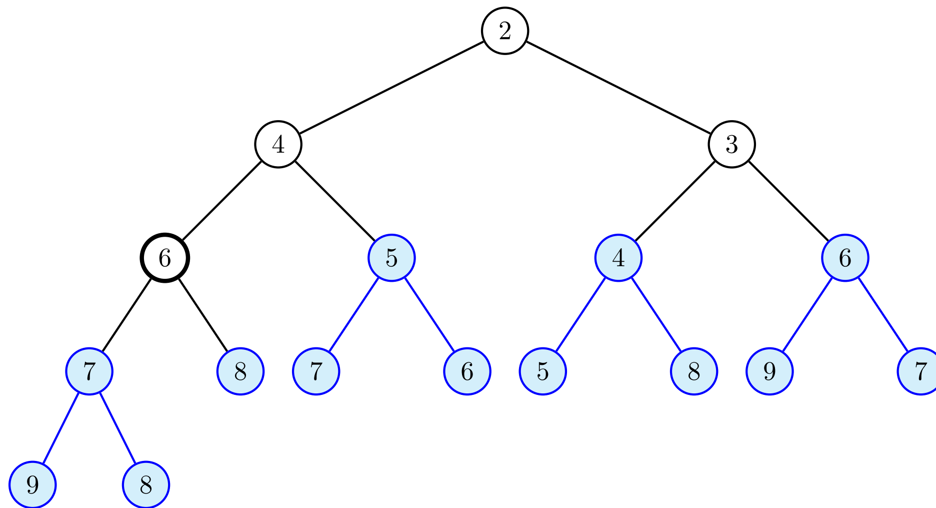
Výpis kódu 12 Predikát validního vrchního řezu v haldě pomocí rodiče

▶ **Definice 3.7** (Spodní řez haldy pomocí rodiče)**.** *Mějme graf $G = (V, E)$ reprezentující binární haldu a $i \in \mathbb{N}$. Spodním řezem haldy pomocí rodiče nazveme graf $G' = (V', E')$, kde*

$$V' = \{v : v \in V \wedge \text{Index}(v) > i\}$$

$$E' = \{(u, v) : (u, v) \in E \wedge u \in V' \wedge v \in V'\}.$$

Obrázek 3.9 zobrazuje ukázku spodního řezu haldy pomocí rodiče pro index 3 na kterém se nachází prvek s hodnotou 6.



Obrázek 3.9 Ukázka spodního řezu haldy pomocí rodiče

▶ Pozorování 3.8. V grafu spodního řezu haldy pomocí rodiče platí haldové uspořádání.

ACSL predikát zobrazený ve výpisu kódu 13 popisuje spodní řez haldy pomocí rodiče. Tento predikát je velmi podobný predikátu vrchního řezu haldy pomocí rodiče. Pokrývá ale vrcholy haldy s indexem větším než problematický vrchol. Nemusí ale platit, že graf spodního řezu haldy pomocí rodiče je na konci vykonávání funkce prázdný. Tato situace nastává pouze v případě, když je problematický prvek probublán do vrcholu s maximálním indexem v haldě.

```
/*@
  predicate HeapLowerParentCut(Heap heap, integer index) =
    \forall integer ancestor, descendant;
      index < ancestor < HeapElementsCount(heap)
      && ancestor < descendant < HeapElementsCount(heap)
      && IsParent(ancestor, descendant) ==>
        HasHeapProperty(heap, ancestor, descendant);
*/
```

Výpis kódu 13 Predikát validního spodního řezu v haldě pomocí rodiče

Algoritmus probublání dolů je v binární haldě aplikován po zvýšení hodnoty některého prvku nebo při rychlé konstrukci haldy, o které pojednává sekce 3.7. Při zvýšení hodnoty prvku haldy můžeme využít horní a dolní řez haldou pomocí rodiče a pomocí invariantů cyklu formálně dokázat korektnosti algoritmu.

Problém představuje algoritmus rychlé konstrukce haldy. Tento algoritmu postupně probublává dolů rodičovské vrcholy od vrcholu s největším indexem až po index kořene haldy. Při tomto způsobu konstrukce je haldové uspořádání zaručeno pouze v dolním řezu haldy pomocí rodiče.

Kontrakt funkce probublání dolů rozlišuje základní a rozšířené vstupní podmínky. Základní podmínky musejí být splněny při všech voláních funkce probublání dolů.

Pokud volající splňuje pouze základní vstupní podmínky kontraktu funkce probublání dolů, tak funkce zajišťuje pouze částečné opravení haldy. Základní podmínky požadují platný predikát

o spodním řezu haldy pomocí rodiče. Pokud je tato základní podmínka splněna, kontrakt funkce volajícího ujišťuje, že ve výsledné haldě bude platit původní spodní řez haldy pomocí rodiče rozšířený o právě probublávaný vrchol. Tato vlastnost je vhodná při postupné konstrukci haldy, ve které se halda konstruuje postupným probublávání prvků dolů.

Rozšířené vstupní podmínky požadují platný predikát o horním řezu haldy pomocí rodiče a tranzitivitu haldového uspořádání mezi horním a spodním řezem haldy. Za předpokladu splnění rozšířených podmínek kontrakt funkce zajišťuje, že po provedení funkce probublání dolů bude výsledná halda validní a bude v ní platit haldové uspořádání. Tyto rozšířené vstupní podmínky jsou splněny při zvýšení hodnoty prvku v haldě, jelikož zvýšením hodnoty nebylo porušeno haldové uspořádání v horním řezu haldy pomocí rodiče ani ve spodním řezu haldy pomocí rodiče.

Základní i rozšířené výstupní ujištění v kontraktu funkce je možné dokázat najednou pomocí ACSL konstrukce *behavior*, která umožňuje některé invarianty cyklu dokazovat pouze za předpokladu, že byl algoritmus zavolán s platnými rozšířenými vstupními podmínkami. Obrázek 3.10 zobrazuje úspěšně dokončený důkaz korektnosti funkce probublání dolů v prostředí Frama-C.

```
/*@ requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
    requires 0 ≤ index < HeapElementsCount(heap);
    requires HeapLowerParentCut(heap, index);
    ensures
      partially_valid_heap:
        ∀ ℤ ancestor, ℤ descendant;
          \old(index) ≤ ancestor < descendant <
          HeapElementsCount(\old(heap)) ∧ IsParent(ancestor, descendant) →
          HasHeapProperty(\old(heap), ancestor, descendant);
    assigns *(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));

    behavior full_repair:
      assumes HeapUpperParentCut(heap, index);
      assumes
        heap_cut_heap_property_left_child:
          HeapHasParent(heap, index) ∧ HeapHasLeftChild(heap, index) →
          HasHeapProperty(heap, Parent(index), LeftChild(index));
      assumes
        heap_cut_heap_property_right_child:
          HeapHasParent(heap, index) ∧ HeapHasRightChild(heap, index) →
          HasHeapProperty(heap, Parent(index), RightChild(index));
      ensures ValidHeap(\old(heap));
 */
void HeapBubbleDown(Heap heap, int index)
{
  int child;
  /*@ loop invariant 0 ≤ index < HeapElementsCount(heap);
      loop invariant
        partial_heap_upper_parent_cut:
          ∀ ℤ ancestor, ℤ descendant;
            \at(index,Pre) ≤ ancestor < index ∧
            ancestor < descendant < HeapElementsCount(heap) ∧
            IsParent(ancestor, descendant) →
            HasHeapProperty(heap, ancestor, descendant);
      loop invariant HeapLowerParentCut(heap, index);
      loop invariant
        partial_heap_parent_cut_heap_property_left_child:
          HeapHasParent(heap, index) ∧ \at(index,Pre) ≤ Parent(index) ∧
          HeapHasLeftChild(heap, index) →
          HasHeapProperty(heap, Parent(index), LeftChild(index));
      loop invariant
        partial_heap_parent_cut_heap_property_right_child:
          HeapHasParent(heap, index) ∧ \at(index,Pre) ≤ Parent(index) ∧
          HeapHasRightChild(heap, index) →
          HasHeapProperty(heap, Parent(index), RightChild(index));
      loop assigns index, child,
                   *(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
      for full_repair: loop invariant HeapUpperParentCut(heap, index);
      for full_repair: loop invariant
        HeapCutHeapPropertyLeftChild(heap, index);
      for full_repair: loop invariant
        HeapCutHeapPropertyRightChild(heap, index);
      loop variant HeapElementsCount(heap) - index;
  */
  while (1) {
    int tmp_1;
    tmp_1 = HeapHasChild(heap,index);
    if (! tmp_1) {
      break;
    }
    {
      int tmp;
      int tmp_0;
      child = HeapLowerChild(heap,index);
      { /* sequence */
        tmp = HeapElementValue(heap,index);
        tmp_0 = HeapElementValue(heap,child);
      }
      if (tmp <= tmp_0) {
        break;
      }
      /*@
      assert
      HeapHasLeftChild(heap, child) →
      HasHeapProperty(heap, child, LeftChild(child)); */
      ;
      /*@
      assert
      HeapHasRightChild(heap, child) →
      HasHeapProperty(heap, child, RightChild(child)); */
      ;
      HeapSwap(heap,index,child);
      index = child;
    }
  }
  return;
}
```

Obrázek 3.10 Úspešné dokončení důkazu probublání dolů v prostředí Frama-C

3.5 Odstranění minimálního prvku

Kořen binární minimové haldy vždy obsahuje prvek s nejmenší hodnotou. Algoritmy, které haldu využívají jako prioritní frontu, potřebují tento minimální prvek najít a odstranit z fronty. Hledání minimálního prvku zajišťuje algoritmus popsaný v sekci 3.1. Algoritmus odstranění minimálního prvku a důkaz korektnosti je popsán v této sekci.

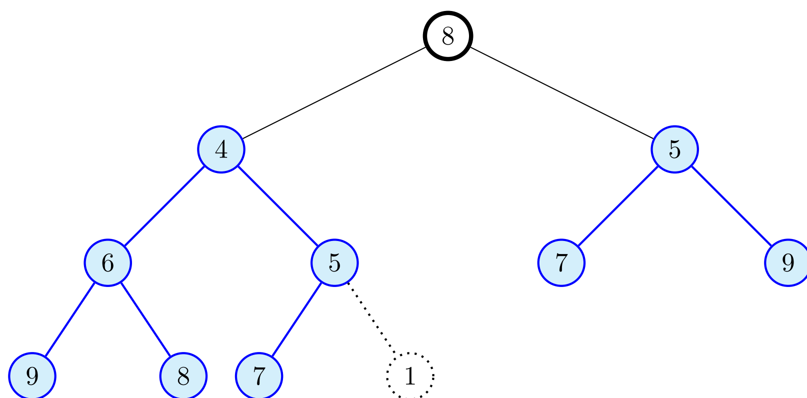
Odstranění minimálního prvku z haldy je operace nad haldou, která v čase $\mathcal{O}(1)$ odstraní daný prvek z haldy a v čase $\mathcal{O}(\log(n))$ opraví haldové uspořádání. Celková asymptotická časová složitost je tedy $\mathcal{O}(\log(n))$.

Kořen haldy lze odstranit pouze v případě, když je kořen jediným prvkem v haldě. Pokud má kořen nějaké potomky, odstraněním kořene by se porušil tvar haldy.

► Pozorování 3.9. Z haldy lze odstranit prvek s největším indexem bez porušení tvaru haldy.

Algoritmus v prvním kroku vymění prvek uložený v kořeni haldy s prvkem uloženým ve vrcholu s největším indexem. Ve druhém kroku je vrchol s největším indexem odstraněn z haldy. Poslední krok algoritmu je probublání kořene haldy dolů.

Výměnou prvku v kořeni haldy s prvkem na největším indexu haldy mohlo dojít k porušení haldového uspořádání. Algoritmus probublání dolů je aplikován s rozšířenými vstupními podmínkami, jelikož je splněn horní a dolní řez haldy pomocí rodiče a tranzitivní haldové uspořádání triviálně platí, jelikož kořen nemá žádného rodiče. Horní řez haldy pomocí rodiče je splněn, jelikož se v případě probublání kořene jedná o prázdný graf. Spodní řez haldy pomocí rodiče obsahuje všechny vrcholy bez kořene a všechny hrany, na kterých neleží kořen. Jelikož v původní haldě před výměnou prvku kořene za prvek z vrcholu s největším indexem platilo haldové uspořádání, tak v podgrafu spodního řezu zmenšené haldy pomocí rodiče musí platit haldové uspořádání.



■ **Obrázek 3.11** Ukázka spodního řezu haldy pomocí rodiče v průběhu odstraňování minimálního prvku

Obrázek 3.11 zobrazuje stav haldy po výměně prvku v kořeni haldy za prvek ve vrcholu s největším indexem. Tečkovaně je naznačena hrana a vrchol, které se v rámci algoritmu odstraňují a tučně je zvýrazněn nový kořen haldy, který bude v následujícím kroku algoritmu probublán dolů.

```
/*@
  requires 0 < HeapElementsCount(heap);
  requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
  requires ValidHeap(heap);

  assigns HeapElements(heap)[0 .. HeapElementsCount(heap) - 1];

  ensures count_decrease: HeapElementsCount(\result) == HeapElementsCount(heap) - 1;
  ensures ValidHeap(\result);
*/
Heap HeapExtractMin(Heap heap) {
  int last = heap.elementsCount - 1;

  HeapSwap(heap, 0, last);

  heap.elementsCount--;

  if (0 < heap.elementsCount) {
    HeapBubbleDown(heap, 0);
  }

  return heap;
}
```

Výpis kódu 14 Kód a ACSL anotace odstranění minimálního prvku z haldy

Výpis kódu 14 zobrazuje kompletní kód funkce odstranění minimálního prvku z haldy společně s kontraktem funkce. Vstupní podmínka pro volání této funkce je nenulový počet prvků v haldě a platný predikát o haldovém uspořádání. Výstupní anotace zajišťují, že funkce zmenšuje počet prvků v haldě a také, že ve výsledné zmenšené haldě stále platí haldové uspořádání.

Implementace tohoto algoritmu odhalila, že pro volání probublání dolů kořene je nutné v haldě po odstranění prvku s maximálním indexem alespoň jeden prvek stále mít. Některé pseudokódy tuto podmínku nezohledňují, jelikož se spoléhají na nulový počet kroků cyklu ve funkci probublání dolů. [5] Z hlediska tvorby formálního důkazu korektnosti tato podmínka musela být zohledněna již ve funkci odstranění minimálního prvku. Tato chyba byla pomocí ACSL odhalena a bylo ji možné ihned opravit a dokončit důkaz korektnosti. Obrázek 3.12 zobrazuje úspěšně dokončený důkaz korektnosti algoritmu odstranění minimálního prvku z haldy.

```
/*@ requires 0 < HeapElementsCount(heap);
    requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
    requires ValidHeap(heap);
    ensures
      count_decrease:
        HeapElementsCount(\result) ≡ HeapElementsCount(\old(heap)) - 1;
    ensures ValidHeap(\result);
    assigns *(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
  */
Heap HeapExtractMin(Heap heap)
{
  /*@ assert rte: signed_overflow: -2147483648 ≤ heap.elementsCount - 1; */
  int last = heap.elementsCount - 1;
  HeapSwap(heap,0,last);
  /*@ assert rte: signed_overflow: -2147483648 ≤ heap.elementsCount - 1; */
  (heap.elementsCount) --;
  if (0 < heap.elementsCount) {
    HeapBubbleDown(heap,0);
  }
  return heap;
}
```

Obrázek 3.12 Úspešné dokončení důkazu odstranění minimálního prvku z haldy v prostředí Frama-C

3.6 Změna hodnoty prvku

Algoritmy, které používají binární minimovou haldu, mohou měnit hodnoty jednotlivých prvků v haldě. Jedná se o změnu priority prvku, který lze v haldě jednoznačně identifikovat pomocí indexu pole. Funkce pro změnu hodnoty používá dvě oddělené funkce pro snížení hodnoty prvku a funkci pro zvýšení hodnoty prvku.

```
/*@
  requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
  requires 0 <= index < HeapElementsCount(heap);
  requires ValidHeap(heap);

  assigns HeapElements(heap)[0 .. HeapElementsCount(heap) - 1];

  ensures value_changed:
    \exists integer i;
      0 <= i < HeapElementsCount(heap) ==>
        HeapElementValue(element) == HeapElementValue(heap, i);
  ensures ValidHeap(heap);
*/
void HeapChange(Heap heap, int index, HeapElement element) {
  if (_HeapElementValue(element) < HeapElementValue(heap, index)) {
    HeapDecrease(heap, index, element);
    return;
  }

  HeapIncrease(heap, index, element);
}
```

Výpis kódu 15 Kód a ACSL anotace změny hodnoty prvku haldy

Výpis kódu 15 zobrazuje funkci pro změnu hodnoty prvku v haldě, která pomocí kontroly aktuální hodnoty prvku v haldě a požadované nové hodnotě prvku volá funkci pro zmenšení hodnoty prvku nebo funkci pro zvětšení hodnoty prvku. Obě zmíněné funkce zajišťují změnu hodnoty prvku haldy a následné opravení haldového uspořádání, které se mohlo narušit změnou

hodnoty prvku. Jelikož je volání těchto funkcí poslední akce ve funkci pro změnu hodnoty prvku v haldě, lze přidat stejné anotace pro ujištění o změně hodnoty prvku a opravení haldového uspořádání do kontraktu funkce o změně hodnoty prvku v haldě.

```
/*@ requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
    requires 0 ≤ index < HeapElementsCount(heap);
    requires ValidHeap(heap);
    ensures
      value_changed:
        ∃ ℤ i;
          0 ≤ i < HeapElementsCount(\old(heap)) ∧
          HeapElementValue(\old(element)) ≡ HeapElementValue(\old(heap), i);
    ensures ValidHeap(\old(heap));
    assigns *(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
 */
void HeapChange(Heap heap, int index, HeapElement element)
{
  int tmp;
  int tmp_0;
  { /* sequence */
    tmp = _HeapElementValue(element);
    tmp_0 = HeapElementValue(heap,index);
  }
  if (tmp < tmp_0) {
    HeapDecrease(heap,index,element);
    {
      goto return_label;
    }
  }
  HeapIncrease(heap,index,element);
  return_label: return;
}
```

Obrázek 3.13 Úspešné dokončení důkazu změny hodnoty prvku haldy v prostředí Frama-C

Obrázek 3.13 zobrazuje úspěšné dokončení důkazu korektnosti funkce pro změnu hodnoty prvku v haldě. Následné je vhodné dokázat korektnost jednotlivých funkcí pro snížení hodnoty prvku v haldě a pro zvýšení hodnoty prvku v haldě.

```
typedef struct _HeapElement {
  int index;

  int id;
  int value;
} HeapElement;
```

Výpis kódu 16 Datová struktura prvku haldy

Výpis kódu 16 zobrazuje datovou strukturu prvku haldy. Tato struktura mimo jiné obsahuje aktuální informaci o indexu daného prvku v haldě, na kterém je daný prvek uložen. Znalost indexu daného prvku umožňuje rychlý přístup k danému prvku v haldě implementované pomocí pole. Úpravu vrcholu je možné provést v čase $\mathcal{O}(1)$ pomocí přímého přístupu na index pole bez nutnosti daný prvek v haldě vyhledávat.

Aktuální informace o indexu daného prvku v haldě je automaticky aktualizována při jakékoli změně či prohození dvou prvků v haldě, jak je znázorněno na výpisu kódu 17.

```
/*@
  requires 0 <= a < HeapElementsCount(heap);
  requires 0 <= b < HeapElementsCount(heap);
  requires \valid(HeapElements(heap) + a);
  requires \valid(HeapElements(heap) + b);

  assigns HeapElements(heap)[a], HeapElements(heap)[b];

  ensures indexes_swapped_a:
    HeapElementIndex(heap, a) == \old(HeapElementIndex(heap, a));

  ensures indexes_swapped_b:
    HeapElementIndex(heap, b) == \old(HeapElementIndex(heap, b));

  ensures HeapElementId(heap, a) == \old(HeapElementId(heap, b));
  ensures HeapElementId(heap, b) == \old(HeapElementId(heap, a));
  ensures HeapElementValue(heap, a) == \old(HeapElementValue(heap, b));
  ensures HeapElementValue(heap, b) == \old(HeapElementValue(heap, a));
*/
void HeapSwap(Heap heap, int a, int b) {
  if (a != b) {
    swapi(&(heap.elements[a].index), &(heap.elements[b].index));
    swapHeapElements(heap.elements + a, heap.elements + b);
  }
}
```

Výpis kódu 17 Kód a ACSL anotace prohození dvou prvků v hladě

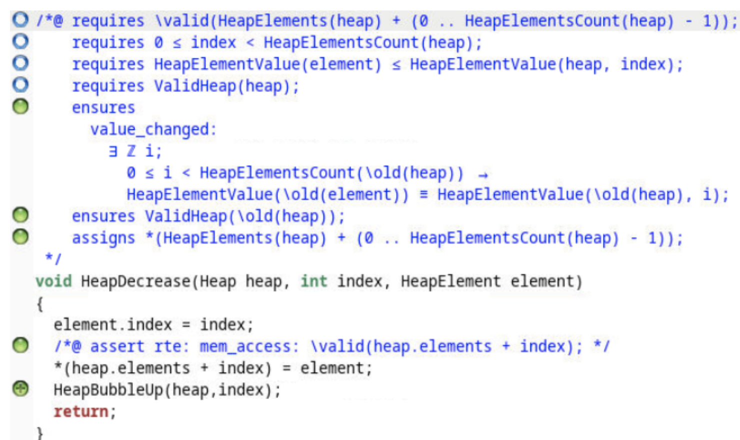
3.6.1 Snížení hodnoty prvku

Snížení hodnoty prvku haldy je operace nad haldou, která dokáže v čase $\mathcal{O}(1)$ snížit hodnotu prvku uloženého ve vrcholu na daném indexu a následně v čase $\mathcal{O}(\log(n))$ opravit haldové uspořádání. Celková časová asymptotická složitost je tedy $\mathcal{O}(\log(n))$.

```
/*@
  requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
  requires 0 <= index < HeapElementsCount(heap);
  requires HeapElementValue(element) <= HeapElementValue(heap, index);
  requires ValidHeap(heap);

  assigns HeapElements(heap)[0 .. HeapElementsCount(heap) - 1];

  ensures value_changed:
    \exists integer i;
      0 <= i < HeapElementsCount(heap) ==>
        HeapElementValue(element) == HeapElementValue(heap, i);
  ensures ValidHeap(heap);
*/
void HeapDecrease(Heap heap, int index, HeapElement element) {
  element.index = index;
  heap.elements[index] = element;

  HeapBubbleUp(heap, index);
}
```

Výpis kódu 18 Kód a ACSL anotace snížení hodnoty prvku v hladě

Výpis kódu 18 zobrazuje funkci pro snížení hodnoty prvku v haldě a její kontrakt. Jedna ze vstupních podmínek této funkce ověřuje, že hodnota prvku, která nahrazuje původní hodnotu prvku je doopravdy nižší nebo stejná. Pokud by tato podmínka neplatila, volání algoritmu probublání nahoru by neopravilo haldové uspořádání a nebylo by možné dokončit důkaz korektnosti algoritmu. Obrázek 3.14 zobrazuje úspěšně dokončený důkaz korektnosti algoritmu snížení hodnoty prvku v haldě.



Obrázek 3.14 Úspešné dokončení důkazu snížení hodnoty prvku haldy v prostředí Frama-C

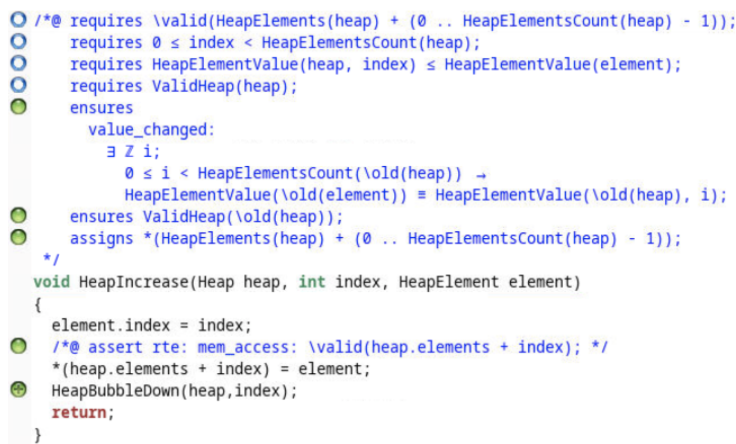
3.6.2 Zvýšení hodnoty prvku

Zvýšení hodnoty prvku haldy je operace nad haldou, která dokáže v čase $\mathcal{O}(1)$ zvýšit hodnotu vrcholu na daném indexu a následně v čase $\mathcal{O}(\log(n))$ opravit haldové uspořádání. Celková časová asymptotická složitost je tedy $\mathcal{O}(\log(n))$.

```
/*@
  requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
  requires 0 <= index < HeapElementsCount(heap);
  requires HeapElementValue(heap, index) <= HeapElementValue(element);
  requires ValidHeap(heap);

  assigns HeapElements(heap)[0 .. HeapElementsCount(heap) - 1];

  ensures value_changed:
    \exists integer i;
      0 <= i < HeapElementsCount(heap) ==>
        HeapElementValue(element) == HeapElementValue(heap, i);
  ensures ValidHeap(heap);
*/
void HeapIncrease(Heap heap, int index, HeapElement element) {
  element.index = index;
  heap.elements[index] = element;

  HeapBubbleDown(heap, index);
}
```

Výpis kódu 19 Kód a ACSL anotace zvýšení hodnoty prvku v hladě

Výpis kódu 19 zobrazuje funkci pro zvýšení hodnoty prvku v haldě a její kontrakt. Jedna ze vstupních podmínek této funkce ověřuje, že hodnota prvku, která nahrazuje původní hodnotu prvku je doopravdy vyšší nebo stejná. Pokud by tato podmínka neplatila, volání algoritmu probublání dolů by neopravilo haldové uspořádání a nebylo by možné dokončit důkaz korektnosti algoritmu. Obrázek 3.15 zobrazuje úspěšně dokončený důkaz korektnosti algoritmu zvýšení hodnoty prvku v haldě.



Obrázek 3.15 Úspešné dokončení důkazu zvýšení hodnoty prvku haldy v prostředí Frama-C

3.7 Konstrukce haldy

Algoritmus konstrukce haldy je prvotní akce, která vytváří strukturu haldy a zajišťuje haldové uspořádání mezi prvky, ze kterých má být halda sestavena.

Implementace pomocí postupného vkládání jednotlivých prvků má časovou složitost $\mathcal{O}(n \log(n))$,

jelikož musíme provést algoritmus vložení prvku, který má časovou složitost $\mathcal{O}(\log(n))$ pro každý prvek, ze kterého má být halda sestavena. [6]

Tento algoritmus lze efektivněji implementovat pomocí postupného opravování haldy probubláváním rodičovských vrcholů dolů. [6] Algoritmus postupně zvětšuje podgraf spodního řezu haldy pomocí rodiče až do té doby, než daný řez obsahuje všechny prvky, ze kterých měla být halda sestavena. Anotace algoritmu probublání dolů obsahují základní vstupní podmínky pro částečné opravení haldy, které jsou zkonstruované právě pro toto použití. Algoritmus konstrukce haldy postupně probublává dolů všechny rodičovské vrcholy od největšího indexu až po kořen haldy.

```
/*@
  requires 0 <= elementsCount <= elementsCapacity;
  requires \valid(elements + (0 .. elementsCount - 1));
  requires correctly_indexed:
    \forall integer i;
      0 <= i < elementsCount ==>
        HeapElementIndex(elements[i]) == i;

  assigns elements[0 .. elementsCount - 1];

  ensures count_same: HeapElementsCount(\result) == elementsCount;
  ensures capacity_same: HeapElementsCapacity(\result) == elementsCapacity;
  ensures ValidHeap(\result);
*/
Heap HeapBuild(HeapElement *elements, int elementsCount, int elementsCapacity) {
  Heap heap;
  heap.elements = elements;
  heap.elementsCount = elementsCount;
  heap.elementsCapacity = elementsCapacity;

  /*@
    loop invariant -1 <= index <= ((int)\floor(HeapElementsCount(heap) / 2)) - 1;

    loop invariant partially_valid_heap:
      \forall integer ancestor, descendant;
        index < ancestor < descendant < HeapElementsCount(heap)
        && IsParent(ancestor, descendant) ==>
          HasHeapProperty(heap, ancestor, descendant);

    loop assigns index, HeapElements(heap)[0 .. HeapElementsCount(heap) - 1];
    loop variant index;
  */
  for (int index = HeapInternalNodeCount(heap) - 1; index >= 0; index--) {
    HeapBubbleDown(heap, index);
  }

  return heap;
}
```

Výpis kódu 20 Kód a ACSL anotace konstrukce haldy

Invarianty cyklu ve výpisu kódu 20 zajišťují haldové uspořádání ve spodním řezu haldy pomocí rodiče s postupně snižujícím se indexem. Tento index bude po posledním kroku cyklu nastaven na hodnotu -1 a invariant spodního řezu haldy pomocí rodiče bude obsahovat celou haldu, tedy bude zajišťovat haldové uspořádání v celé haldě.

Kapitola 4

Metody ladění při vývoji důkazu

Důkazní postupy v jazyce ACSL mohou být výrazně odlišené od matematických a formálních důkazů v knihách a publikacích. Při tvorbě nebo přepisu predikátů, invariantů cyklů nebo při kontrole vstupních podmínek je výhodné pro vývojáře znát některé techniky, které mohou pomoci odhalit chyby v kódu nebo v důkazech.

4.1 Zúžení vstupních podmínek

Při návrhu funkce lze definovat vstupní podmínky, které musejí být splněné, aby mohla být daná funkce volána. Tato podmínka může kontrolovat například hodnoty předávaných parametrů. Pokud by funkce zaručovala platnost nějakého tvrzení při dokončení funkce, může nastat situace, že žádný z důkazních nástrojů nebude schopný danou vlastnost formálně ověřit. Pokud se jedná o chybu v kódu nebo anotaci funkce, lze ji často odhalit zúžením vstupních podmínek.

Mějme induktivní predikát z výpisu kódu 21, který tvoří tranzitivní uzávěr vlastnosti být potomkem v binární minimové haldě. Induktivní predikát obsahuje základní podmínky platnosti a následně induktivně popsanou vlastnost.

```
/*@
  inductive IsDescendant(Heap heap, integer descendant, integer ancestor) {
    case children:
      \forall Heap heap, integer child;
        0 < child < HeapElementsCount(heap) ==>
          IsDescendant(heap, child, Parent(child));

    case descendants:
      \forall Heap heap, integer ancestor, element;
        0 <= ancestor < element < HeapElementsCount(heap) ==>
          IsDescendant(heap, Parent(element), ancestor) ==>
            IsDescendant(heap, element, ancestor);
  }
*/
```

Výpis kódu 21 ACSL predikát tranzitivní vlastnosti být potomkem

Pomocí prázdné funkce se vstupní podmínkou na počet prvků v haldě lze tento predikát otestovat. Na obrázku 4.1 je vidět příklad použití této metody. Verifikační prostředí Frama-C správně nebylo schopné dokázat, že vrchol s indexem 3 je potomkem vrcholu s indexem 2. Tento predikát tedy nejspíše neobsahuje chybu. Nejedná se ale o kompletní formální důkaz korektnosti, ale pouze o ujištění pro vývojáře, že predikát funguje na ukázkových datech.

```
/*@ requires \valid(HeapElements(heap) + (0 .. HeapElementsCount(heap) - 1));
    requires 5 = HeapElementsCount(heap);
    assigns \nothing;
 */
void testIsDescendantPredicate(Heap heap)
{
  /*@ assert IsDescendant(heap, 1, 0); */ ;
  /*@ assert IsDescendant(heap, 2, 0); */ ;
  /*@ assert IsDescendant(heap, 3, 1); */ ;
  /*@ assert IsDescendant(heap, 4, 1); */ ;
  /*@ assert IsDescendant(heap, 3, 0); */ ;
  /*@ assert IsDescendant(heap, 4, 0); */ ;
  /*@ assert IsDescendant(heap, 3, 2); */ ;
  return;
}
```

Obrázek 4.1 Testování platnosti predikátu IsDescendant v prostředí Frama-C

4.2 Kontradikce

V průběhu tvorby důkazu je vhodné jednou za čas vložit do kódu testovací anotaci zobrazenou ve výpisu kódu 22. Pokud nastane situace, kdy by verifikační prostředí Frama-C bylo toto tvrzení schopno dokázat, nejspíše byly zavedeny vstupní podmínky funkce, které nelze splnit. Nebo byla chybně napsaná ACSL implikace v kódu před touto kontrolou.

```
...

//@ assert \false;

...
```

Výpis kódu 22 ACSL kontrola kontradikce

Kapitola 5

Hledání nejkratší cesty

5.1 Dijkstrův algoritmus

Dijkstrův algoritmu je grafový algoritmus pomocí kterého lze nalézt nejkratší cestu mezi dvěma vrcholy ohodnoceného orientovaného grafu. Algoritmus předpokládá nezáporné ohodnocení hran. Algoritmus hledá nejkratší cesty z vrcholu s do všech vrcholů grafu. Algoritmus poté vypisuje pouze hrany z cílového vrcholu. Použitím binární minimové haldy lze dosáhnout asymptotické časové složitosti $\mathcal{O}(|E| \log(|V|))$. [6]

Následující pseudokód algoritmu 1 zobrazuje Dijkstrův algoritmus pro hledání nejkratší cesty v grafu s využitím binární minimové haldy.

Algoritmus 1 Pseudokód Dijkstrova algoritmu s binární haldou

Požaduje $G = (V, E), v_0 \in V$
  for all $v \in V$ **do**
    Value(v) $\leftarrow +\infty$
    Predecessor(v) $\leftarrow \perp$
  end for
  Value(v_0) $\leftarrow 0$ ▷ Vzdálenost počáteční vrcholu od počátečního vrcholu
  $H \leftarrow HeapBuild(V)$
  while H is not empty **do**
    $v \leftarrow$ HeapFindMin(H)
    $H \leftarrow$ HeapExtractMin(H)
    for all $w \in N_G(v)$ **do** ▷ Pro všechny sousedící vrcholy w vrcholu v
      if $w \in H \wedge$ Value(w) $>$ Value(v) + Weight((v, w)) **then**
        Value(w) $\leftarrow$ Value(v) + Weight((v, w))
        Predecessor(w) $\leftarrow v$
        HeapChange(H, Index(w), w)
      end if
    end for
  end while

Dijkstrův algoritmus byl implementován s použitím plně verifikované knihovny binární minimové haldy. Ale již nebyl verifikován pro korektnost. Kompletní zdrojový kód implementace Dijkstrova algoritmu se nachází na přiloženém médiu v souboru „shortest_path.c“.

Kapitola 6
Vývojové prostředí

6.1 Docker

Docker a kotejnerové technologie umožňují spouštět kontejnery na společném jádře operačního systému. Liší se tímto od tradičních virtuálních strojů, které virtualizují jádro daného operačního systémů. [7]

V rámci práce vznikl konfigurační soubor pro Docker, který umožňuje vývojáři editovat kód, psát ACSL anotace a spouštět formální verifikaci v grafickém prostředí Frama-C pomocí webového prohlížeče. Vývojové prostředí obsahuje reverzní proxy, která zajišťuje HTTPS komunikaci a jednotný přístup k editoru kódu a grafickému prostředí Frama-C pomocí webového prohlížeče. Pro správné použití této reverzní proxy je nutné nastavení lokálního překladu doménových jmen, například v souboru „/etc/hosts“ na Linuxu, nebo nastavení DNS serveru. Domény, které je potřeba nastavit jsou:

1. vscode.local
2. frama-c.local

Editor kódu generuje při prvním vytvoření přístupové heslo do souboru „config.yaml“ v automaticky vytvořeném adresáři „./config/code-server“. Grafické prostředí Frama-C je přístupné pomocí technologie pro vzdálenou plochu VNC.

Závěr

Tato práce se zaměřila na problematiku formálního důkazu korektnosti implementace datové struktury binární minimové haldy. Podařilo se implementovat knihovnu v jazyce C s ACSL anotacemi, které dokazují korektnost jednotlivých knihovních funkcí. Použitím této knihovny lze psát nové programy, které bude možné kompletně formálně ověřit, jelikož používají formálně ověřenou knihovnu.

Formálně ověřená implementace binární minimové haldy byla použita v Dijkstrově algoritmu pro hledání nejkratší cesty v grafu. Rozšířením této práce by mohl být důkaz korektnosti tohoto algoritmu. Binární halda byla navržena pro ukládání celočíselných hodnot jednotlivých prvků. Další modifikace knihovny by mohla obsahovat rozšíření pro možnost uložení desetinných čísel.

Součástí práce také vzniklo univerzální vývojové prostředí s možností editace kódu a spouštění formálního ověření bez nutnosti instalace jednotlivých programů přímo do operačního systému. Toto vývojové prostředí lze spustit na vzdáleném počítači (serveru) a následně editaci kódu a spouštění verifikace programů provádět pomocí webového prohlížeče vzdáleně.

Bibliografie

1. BECK, Kent. *Test Driven Development*. 1st Edition. Boston: Addison Wesley, 2002. ISBN 0-321-14653-0.
2. *ACSL: ANSI/ISO C Specification Language: Version 1.17* [online] [cit. 2022-05-09]. Dostupné z: https://frama-c.com/download/acsl-implementation-24.0-Chromium.pdf.
3. *WP Plug-in Manual: Frama-C 24.0* [online] [cit. 2022-05-09]. Dostupné z: https://frama-c.com/download/wp-manual-24.0-Chromium.pdf.
4. *Frama-C's RTE plug-in: for Frama-C 24.0* [online] [cit. 2022-05-09]. Dostupné z: https://frama-c.com/download/rte-manual-24.0-Chromium.pdf.
5. TOMÁŠ VALLA, Martin Mareš a. *Průvodce labyrintem algoritmů*. Praha: CZ.NIC, z.s.p.o., 2017. ISBN 978-80-88168-19-5.
6. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. Fourth Edition. Cambridge: The MIT Press, 2022. ISBN 9780262367509.
7. NEGUS, Christopher. *Linux Bible*. 10th Edition. Indianapolis: John Wiley & Sons, Inc., 2020. ISBN 978-1-119-57889-5.

Obsah přiloženého média

```
 thesis.pdf ............................................... text práce ve formátu PDF
 bachelor-thesis-text ..................... zdrojová forma práce ve formátu LaTeX
 bachelor-thesis-code .... zdrojové kódy implementace a konfigurace vývojového prostředí
   docker-compose.yml ...................... konfigurační soubor vývojového prostředí
   src
     min_heap.h ........... hlavičkový soubor ověřené knihovny binární minimové haldy
     min_heap.c ................ zdrojový kód ověřené knihovny binární minimové haldy
     shortest_path.c ................. zdrojový kód implementace Dijkstrova algoritmu
```