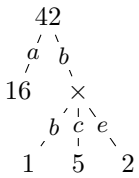
IB015 – Domácí úkol 10: prefixové stromy

Termín: do 21. 11. 23.59; 5 pokusů odevzdání; způsob odevzdání je podrobně popsán níže.

Úvod a představení datové struktury

Prefixový strom, neboli *trie*, je datová struktura pro ukládání dvojic *klíč*, *hodnota* takovým způsobem, že podle klíče dokážeme hodnotu rychle najít (rychleji než například v seznamu dvojic) a zároveň dokážeme hodnoty i rychle přidávat a mazat. Od jiných datových struktur sloužících k podobnému účelu se prefixový strom liší tím, že klíče musí mít další strukturu – musí to být sekvence hodnot. Příklady datových typů, které se dají použít jako klíč, jsou tak typicky řetězce (sekvence znaků) a seznamy. Klíč, pod kterým je hodnota uložena budeme pak číst na cestě od kořene stromu k této hodnotě.

Ukažme si na příkladu, jak takový strom může vypadat a jak s ním pak pracujeme (klíče jsou v tomto případě řetězce).



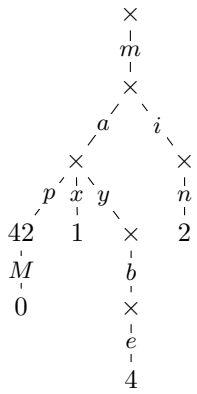
Na hranách stromu jsou znaky (`Char`), klíče tedy budou řetězce (`[Char]`). Tento strom má kořen (nahoře) s hodnotou 42. To znamená, že prázdnému řetězci odpovídá hodnota 42. Pokud projdeme dále po hraně označené znakem `'a'`, dostaneme hodnotu 16 – s klíčem `"a"` je tedy asociována hodnota 16. O něco zajímavější je, co se stane, pokud půjdeme z kořene doprava pod `'b'` – zde sice není žádná asociovaná hodnota (znázorněno jako ×), ale můžeme dále pokračovat, například pod `'c'` do hodnoty 5. To znamená, že klíč `"bc"` odpovídá hodnotě 5 – všimněte si, že jsme dali dohromady postupně znaky nacházející se na cestě, kterou jsme se od kořene dostali do uzlu s hodnotou 5. Podobně bychom mohli postupovat pro ostatní hodnoty, celkově bychom dostali následující dvojice (*klíč*, *hodnota*): `[("", 42), ("a", 16), ("bb", 1), ("bc", 5), ("be", 2)]`.

Vyhledávání Pokud tedy chceme ve stromě vyhledávat, každý znak vstupního klíče nám udává instrukce pro to, kterého z následníků vybrat v podstromu, kam jsme došli po předchozích znacích. Hledáme-li tedy `"bc"`, půjdeme nejdřív z kořene do následníka s hranou označenou `'b'` a následně odsud do následníka s hranou označenou `'c'`.

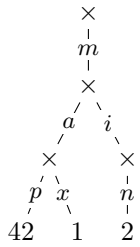
U prefixových stromů budeme navíc vyžadovat, aby byli potomci daného uzlu seřazení (podle příslušné části klíče) a bez duplicit. To nám, mimo jiné, umožní někdy ukončit vyhledávání v potomcích dříve. Uvažme, že v našem stromě hledáme hodnotu `"ba"` – nejprve opět projdeme z kořene do druhého následníka; tu však zjistíme, že hned první potomek má příslušnou část klíče `'b'` a tedy se (díky seřazenosti potomků) potomek pod `'a'` v seznamu jistě nenachází. Zbytek seznamu tedy procházet nemusíme a hledání rovnou ukončíme s neúspěchem.

Vkládání Podobně, pokud bychom nyní chtěli do stromu vložit dvojici `("ba", 0)`, šli bychom nejprve pod `'b'` do uzlu bez hodnoty a zde bychom museli vytvořit novou hranu pod `'a'` a na ní uzel s hodnotou 0.

Mazání Mazání je podobné. Uvažme, že bychom mazali hodnotu odpovídající klíči `"bc"`: projdeme postupně až do uzlu, který odpovídá tomuto klíči, a zde smažeme hodnotu. Abychom ale nedostali strom se zbytečnými uzly, budeme navíc chtít aby *prefixový strom neobsahoval listy bez hodnot*. Jelikož jsme právě takový list vytvořili, musíme jej odstranit – odebereme i celou hranu vedoucí tohoto listu. Pokud by nám tím opět vznikl list bez hodnoty, odstraníme i ten a tak podobně. Jediný případ, kdy list může nemít hodnotu, je pokud je to zároveň kořen, a tedy je celý strom prázdný. Ukažme si to ještě na jednom příkladě.



Tento strom kóduje dvojice `[("map", 42), ("mapM", 0), ("max", 1), ("maybe", 4), ("min", 2)]`. Pokud se rozhodneme odstranit hodnotu odpovídající `"maybe"`, budeme muset smazat i celou cestu z uzlu `"ma"` pod `"ybe"` (cesty můžeme indentifikovat zřetězením klíčů na příslušných hranách, uzly pak cestami z kořene). Samotný uzel `"ma"` samozřejmě musíme nechat – sice nemá hodnotu, ale má následníky. Podobně, pokud bychom mazali hodnotu odpovídající `"mapM"`, tak smažeme koncový uzel `"mapM"`, uzel `"map"` ale necháme, protože má hodnotu (42). Po smazání obou hodnot bychom tedy měli následující strom:



Pokud prefixový strom neobsahuje listy bez hodnot budeme říkat, že je ***redukovaný***. Formálněji můžeme redukovanost definovat následovně.

- Uzel je redukovaný, pokud buď má hodnotu, nebo má hodnotu některý z jeho (ne nutně přímých) potomků.
- Prefixový strom je redukovaný, pokud je prázdný a tvořený jediným uzlem, nebo pokud jsou všechny jeho uzly redukované (všimněte si, že kořenový uzel prázdného jednouzlového stromu redukovaný není).

Pro naše účely bude ještě důležité si uvědomit, že zachování redukovanosti prefixového stromu u operace mazání hodnoty uzlu je lokální záležitost – mazat můžeme tento uzel, případě i s jeho některými předky, ale nikdy nepotřebujeme mazat uzly, které se nenachází na cestě z kořene do modifikovaného uzlu.

Validita prefixových stromů

Na prefixové stromy budeme mít určité požadavky k tomu, abychom je považovali za ***validní***. Konkrétně půjde o to, že následníci uzlu musí být seřazení (podle příslušné části klíče) a nesmí se v nich vyskytovat duplicity.

Tyto požadavky máme primárně proto, aby byla reprezentace každé množiny dvojic jednoznačná – nezávisle na tom, v jakém pořadí hodnoty do stromu vkládáme, existuje pouze jediný validní redukovaný prefixový strom, který danou množinu hodnot představuje (všimněte si, že pokud bychom upustili od požadavku na redukovanost, pak stromů existuje nekonečně mnoho, protože vždy můžeme přidat prázdné listy, redukovanost ale není zahrnuta ve validitě).

Reprezentace v Haskellu

Pro reprezentaci prefixových stromů budeme používat parametrizovaný datový typ `Trie`:

```
data Trie e a = TNode (Maybe a) [Edge e a]
              deriving (Show, Eq)
```

Každý uzel `TNode` obsahuje volitelnou hodnotu a seznam hran, které z něj odchází, spolu s dalšími uzly, které se za těmito hranami nachází. První typový argument představuje typ používaný pro ohodnocení hran, druhý je pak typem hodnot uložených ve stromě.

Hrany `Trie` jsou reprezentovány dvojicí, která obsahuje označení hrany a následníka dostupného pod touto hranou. Pro přehlednost si pro ni zavedeme typový alias.

```
type Edge e a = (e, Trie e a)
```

Náš strom z posledního obrázku výše pak můžeme reprezentovat v Haskellu takto:

```
trie3 :: Trie Char Integer
trie3 = TNode Nothing [
        ('m', TNode Nothing [
              ('a', TNode Nothing [('p', TNode (Just 42) []),
                                   ('x', TNode (Just 1) [])]),
              ('i', TNode Nothing [('n', TNode (Just 2) [])])]
        )]
```

Funkce k implementaci

Funkcí k implementaci je hodně, ale nelekejte se, mnoho funkcí lze implementovat s pomocí jiných funkcí ze zadání či za pomoci společných pomocných funkcí. Nejprve proto implementaci dobře rozmyslete. Myslete též na to, že při rozmýšlení práce se stromy se může velmi hodit tužka a papír.

Není-li uvedeno jinak, předpokládejte, že všechny hodnoty typu `Trie` na vstupu jsou *validní*, ostatní případy neřešte (nemusí však být *redukované*). Pokud je výstupem prefixový strom, pak tento musí být *validní* ve všech případech. Typové signatury funkcí neměňte.

- `emptyTrie :: Trie e a`

  Tato konstanta vrátí prázdný prefixový strom – tj. strom bez hodnoty s jediným uzlem.

- `isTrieValid :: Ord e => Trie e a -> Bool`

  Rozhodne, zda je `Trie` validní. Přesněji můžeme validitu definovat následovně.

  - Uzel `Trie e a` je validní pokud:
    - seznam následníků je seřazený podle první složky dvojice (v uspořádání daném `Ord e`);
    - žádní dva následníci nemají stejný klíč.
  - Celá `Trie` je validní, pokud jsou validní všechny její uzly. Alternativně řečeno, `Trie` je validní, pokud je její kořenový uzel validní a zároveň jsou všechny jeho podstromy validní `Trie`.

  V tomto případě samozřejmě na vstupu nemusí být validní hodnota.

  ```
  isTrieValid emptyTrie ~>* True
  isTrieValid trie2 ~>* True
  isTrieValid (TNode Nothing [('a', TNode Nothing [])]) ~>* True
  isTrieValid (TNode Nothing [('b', TNode (Just 1) []), ('a', TNode (Just 2) [])])
      ~>* False
  ```

- `trieLookup :: Ord e => [e] -> Trie e a -> Maybe a`

  Vyhledá hodnotu danou klíčem v prvním argumentu v prefixovém stromě daném druhým argumentem. Pokud se hodnota ve stromě nenachází, vrátí `Nothing`, v opačném případě tuto hodnotu vrátí zabalenou v `Just`.

  Vaše funkce musí strom procházet efektivně, tedy cíleně k hledanému uzlu, nesmíte procházet větve, v nichž se hodnota určitě nemůže nacházet.

  Nápověda: Ačkoli se funkce `lookup` a `find` (obě z `Data.List`) mohou zdát užitečné, v implementaci `trieLookup` využití nenajdou, protože nedokáží využít toho, že jsou následníci trie seřazení. Seřazenost nám, mimo jiné, umožňuje ukončit hledání potomků dříve. Naopak funkce `lookup` a `find` prochází seznam vždy do prvního výskytu hledaného prvku a pokud tam žádný takový není tak do jeho konce.

```
trieLookup "map" trie2 ~>* Just 42
trieLookup "filter" trie2 ~>* Nothing
```

- `trieMember :: Ord e => [e] -> Trie e a -> Bool`

  Rozhodne, zda prefixový strom daný druhým argumentem obsahuje hodnotu odpovídají klíči v prvním argumentu. Vaše funkce musí strom procházet efektivně, tedy cíleně k hledanému uzlu, nesmíte procházet větve, v nichž se hodnota určitě nemůže nacházet.

  Dodatečný požadavek: implementujte tuto funkci bez použití explicitní rekurze v její implementaci, pouze za pomoci volání vhodné jiné *zadané* funkce z této úlohy nebo pomocí kombinace více zadaných funkcí.

  ```
  trieMember "mapM" trie2 ~>* True
  trieMember "mapM" trie3 ~>* False
  ```

- `trieMapMaybe :: ([e] -> a -> Maybe b) -> Trie e a -> Trie e b`

  Volání funkce `trieMapMaybe f t` funguje tak, že pro každou hodnotu v `t` zavolá funkci `f` a předá jí klíč vedoucí k této hodnotě a hodnotu samotnou. Pokud je pak výsledek volání `f` na těchto argumentech `Nothing`, bude odpovídají hodnota ve výsledném prefixovém stromě chybět. Pokud je výsledek volání `f` ve tvaru `Just x`, pak se na odpovídající pozici ve výsledném prefixovém stromě objeví hodnota `x`.

  Jedná se o obdobu funkce `mapMaybe :: (a -> Maybe b) -> [a] -> [b]` z modulu Data.Maybe. Je tedy podobná i funkci `map`, která má ale navíc schopnost vyhazovat elementy tím, že pro ně vrátíme `Nothing`. V tomto případě budeme místo seznamů mapovat přes prefixové stromy. Dalším rozdílem je i to, že do volané funkce, která rozhoduje o osudu hodnoty, vstupuje i klíč této hodnoty.

  Všimněte si, že voláním funkce `trieMapMaybe` nelze přidávat klíče ani měnit, pod jakým klíčem je hodnota uložena, lze ji však použít jak ke transformaci hodnot, tak k jejich odstraňování.

  Nápověda: zaveďte si pomocnou funkci, která obsahuje jako argument klíč vedoucí do aktuálně zpracovávaného uzlu.

  Dodatečný požadavek: pokud je vstupní prefixový strom redukovaný, pak i výstupní prefixový strom musí být redukovaný a to i za předpokladu, že odstraňujeme uzly. Může tedy být nutné odstranit i předky odstraněných uzlů. Všimněte si, že tento požadavek nijak neovlivňuje stromy, které redukované nejsou (pro ně funkce může, ale nemusí produkovat redukované výsledky).

  ```
  import Data.Maybe
  trieMapMaybe (\_ v -> listToMaybe v) trie4
      ~>* TNode Nothing [('a', TNode (Just 'h') []),
                         ('b', TNode Nothing [('b', TNode (Just 'a') [])])]
  ```

- `trieMap :: ([e] -> a -> b) -> Trie e a -> Trie e b`

  Obdoba `map`, ale pro prefixové stromy. Do volané funkce opět vstupuje i klíč dané hodnoty (viz `trieMapMaybe`). Pokud je vstupem redukovaný prefixový strom, tak i výstupe musí být redukovaný.

  Dodatečný požadavek: implementujte tuto funkci bez použití explicitní rekurze v její implementaci, pouze za pomoci volání vhodné jiné *zadané* funkce z této úlohy nebo pomocí kombinace více zadaných funkcí.

  ```
  trieLookup "map" (trieMap (\k v -> (length k, v `div` 2)) trie2) ~>* Just (3, 21)
  ```

- `trieFilter :: ([e] -> a -> Bool) -> Trie e a -> Trie e a`

  Obdoba `filter`, ale pro prefixové stromy. Do volané funkce opět vstupuje i klíč dané hodnoty (viz `trieMapMaybe`).

  Dodatečný požadavek 1: pokud je vstupní prefixový strom redukovaný, pak i výstupní prefixový strom musí být redukovaný.

  Dodatečný požadavek 2: implementujte tuto funkci bez použití explicitní rekurze v její implementaci, pouze za pomoci volání vhodné jiné *zadané* funkce z této úlohy nebo pomocí kombinace více zadaných funkcí.

  ```
  trieMember "map" (trieFilter (\_ v -> v == 42) trie2) ~>* True
  trieMember "map" (trieFilter (\_ v -> v /= 42) trie2) ~>* False
  ```

Pokud zároveň platí dodatečný požadavek 1, tak též:

```
trieFilter (\_ v -> v == 16) trie1 ~>* TNode Nothing [('a', TNode (Just 16) [])]
```

- `trieToList :: Trie e a -> [([e], a)]`

  Převede daný prefixový strom na seznam dvojic (*klíč*, *hodnota*), který je uspořádaný podle přirozeného uspořádání hodnot ve stromě[1]. Toto uspořádání vypadá tak, že hodnota v daném uzlu se v seznamu objeví před všemi potomky tohoto uzlu a hodnoty potomků a jejich podstromů se objevují postupně v pořadí daném seznamem následníků uzlu. Pokud je na vstupu validní prefixový strom pak toto uspořádání odpovídá tomu, že výsledný seznam bude seřazený podle klíčů.

  U této funkce *nemusí být vstupem validní prefixový strom* (funkci samotnou by to ale nijak nemělo ovlivnit). Pokud tedy strom na vstupu není validní, pak výsledný seznam nemusí být uspořádaný podle klíčů (pořadí je pevně dané vstupním stromem).

  Nápověda: zaveďte si pomocnou funkci, která obsahuje jako argument klíč vedoucí do aktuálně zpracovávaného uzlu.

  ```
  trieToList trie2 ~>* [("map", 42), ("mapM", 0), ("max", 1), ("maybe", 4), ("min", 2)]
  trieToList (TNode Nothing [('b', TNode (Just 1) [('c', TNode (Just 3) [])]),
                             ('a', TNode (Just 2) [])])
      ~>* [("b", 1), ("bc", 3), ("a", 2)]
  ```

- `trieMerge :: Ord e => ([e] -> a -> a -> Maybe a) -> Trie e a -> Trie e a -> Trie e a`

  Sloučí dva prefixové stromy s pomocí dané funkce. Přesněji, volání `trieMerge f t1 t2` spojí stromy `t1` a `t2` následujícím způsobem:

  - hodnoty, které se nachází pouze v `t1` nebo pouze v `t2` budou ve výsledném stromě (pod stejným klíčem);
  - pro hodnoty, které se nachází v obou stromech (nazvěme si je `v1` a `v2`, kde `v1` je z `t1` a `v2` je z `t2`) se zavolá funkce `f` na těchto hodnotách a s jejich klíčem: `f key v1 v2`. Pokud tato funkce vrátí `Nothing` pak v odpovídajícím uzlu ve výsledném stromě žádná hodnota nebude, pokud vrátí `Just x` pak se v odpovídajícím uzlu objeví hodnota `x`.

  Vaše funkce musí stromy spojovat efektivně, nesmí tedy zbytečně procházet podstromy, které musí být ve výsledku obsaženy celé, či například převádět vstupní stromy na seznam a z něj pak budovat výstupní stromy.

  Dodatečný požadavek: pokud jsou oba vstupní prefixové stromy redukované, pak i výstupní prefixový strom musí být redukovaný.

  ```
  trieMerge (\_ a b -> Just (a - b)) trie1
      (TNode Nothing [('a', TNode (Just 8) []), ('b', TNode (Just 0) [])])
      ~>* (TNode (Just 42) [('a', TNode (Just 8) []),
                            ('b', TNode (Just 0) [('b', TNode (Just 1) []),
                                                  ('c', TNode (Just 5) []),
                                                  ('e', TNode (Just 2) [])])])
  ```

- `singletonTrie :: [e] -> a -> Trie e a`

  Vytvoří prefixový strom obsahující jedinou danou hodnotu s daným klíčem. Výsledný prefixový strom musí být redukovaný (a validní).

- `trieInsert :: Ord e => [e] -> a -> Trie e a -> Trie e a`

  Vloží danou hodnotu do daného prefixového stromu pod daným klíčem. Pokud se již hodnota s daným klíčem ve stromě nachází, pak je přepsána novou hodnotou.

  Vaše funkce musí do stromu vkládat efektivně, tedy musí mířit cíleně k místu vložení a nesmí procházet větve, do nichž určitě hodnotu nebude vkládat. Pokud je vstupem redukovaný prefixový strom, tak i výstupem musí být redukovaný strom.

  Dodatečný požadavek: implementujte tuto funkci bez použití explicitní rekurze v její implementaci, pouze za pomoci volání vhodné jiné *zadané* funkce z této úlohy nebo pomocí kombinace více zadaných funkcí.

[1] Jde o uspořádání nazývané obvykle DFS preorder, s tímto názvem se setkáte později ve studiu.

```
trieLookup "filter" (trieInsert "filter" 99 trie2) ~>* Just 99
trieLookup "map" trie2 ~>* Just 42
trieLookup "map" (trieInsert "map" 101 trie2) ~>* Just 101
```

- `trieDelete :: Ord e => [e] -> Trie e a -> Trie e a`

  Odstraní hodnotu asociovanou s daným klíčem ze stromu.

  Vaše funkce musí strom procházet efektivně, tedy cíleně k odstraňovanému uzlu, nesmíte procházet větve, v nichž se hodnota určitě nemůže nacházet.

  Dodatečný požadavek: pokud je vstupní prefixový strom redukovaný, pak i výstupní prefixový strom musí být redukovaný.

  ```
  trieLookup "map" (trieDelete "map" trie2) ~>* Nothing
  trieDelete "map" emptyTrie == emptyTrie ~>* True
  ```

Poznámky a tipy

- Importovat smíte moduly z balíku `base`.[2]
- Neduplikujte kód! Snažte se vždy využít funkce, které jste již naprogramovali. Pokud to nejde přímo, ale přesto vidíte v řešení podobu, vytkněte podobnou část do pomocné funkce.
- Nevynalézejte znovu kolo! Snažte se využívat knihovních funkcí, především nemá smysl je zbytečně reimplementovat. Doporučujeme použít vyhledávač `hoogle.fi.muni.cz`. Pozornost věnujte především modulu `Data.Maybe`.
- Pomocné funkce definujte lokálně, pokud jsou využívány jedinou funkcí.
- Funkce jsou v kostře zadefinovány jako `undefined`, takže projdou překladem, ale jejich zavolání způsobí chybu.
- Nejste-li si jisti nějakou částí zadání, zeptejte se v diskusním fóru.
- Nezapomeňte, že **opisování je zakázáno** a bude postihováno podle disciplinárního řádu.
- Přebíráte-li kód odjinud, uveďte zdroj, jinak bude na vaši práci pohlíženo jako na plagiát.
- Než řešení odevzdáte, **pečlivě si přečtěte následující sekci** a ujistěte se, že váš kód splňuje všechny náležitosti. Neztrácejte body jen kvůli nepozornému čtení pokynů.

Odevzdání a hodnocení

Na úlohu máte opět 5 pokusů. Úloha má několik automaticky testovaných aspektů, které jsou samostatně bodově hodnoceny. Body můžete získávat i za řešení bez splnění dodatečných požadavků, určitě se ale vyplatí se nad nimi zamyslet a splnit je, protože si tím můžete také ušetřit dost práce.

Úloha nemá žádnou učitelem bodovanou část, pouze části bodované automaticky. Tipy k psaní hezkého kódu ve sbírce se vám nicméně budou hodit i tak, protože v hezkém kódu se snáze hledají chyby.

Funkcionalita	Body
`emptyTrie`	0.05
`isTrieValid`	0.2
`trieLookup`	0.2
`trieMember`	0
+ dodatečný požadavek – bez rekurze	0.05
`trieMapMaybe`	0.1
+ dodatečný požadavek – redukovanost	0.1
`trieMap`	0.1
+ dodatečný požadavek – bez rekurze	0.15
`trieFilter`	0.1
+ dodatečný požadavek – redukovanost	0.1
+ dodatečný požadavek – bez rekurze	0.15
`trieToList`	0.15
`trieMerge`	0.1
+ dodatečný požadavek – redukovanost	0.1

[2] Výjimku tvoří moduly, které jsou „Unsafe“, ty však určitě nebudete potřebovat.

Funkcionalita	Body
`singletonTrie`	0.05
`trieInsert`	0.2
+ dodatečný požadavek – bez rekurze	0.2
`trieDelete`	0.2
+ dodatečný požadavek – redukovanost	0.2

Odevzdání

Tento domácí úkol se odevzdává přes příslušnou odevzdávárnu v Informačním systému.

- Do odevzdávárny vkládejte **jediný** soubor s příponou `.hs` obsahující vaši implementaci **všech** požadovaných funkcí.
  - Pokud chcete odevzdat znovu, můžete soubor přepsat či přidat nový, nezáleží na tom – již vyhodnocené soubory se znovu nevyhodnocují.
  - Žádné jiné soubory nevkládejte.
  - Již vložené soubory nepřejmenovávejte, mohou se pak vyhodnotit znovu.
- Vaše řešení **musí zachovat všechny definice datových typů tak, jak jsou v zadání**, jediná povolená modifikace je přidání instance typové třídy.
- Rovněž musíte **zachovat typy zadaných funkcí**.
- Řešení musí jít přeložit překladačem GHC 9.0.1. Je možné, že na svých počítačích máte starší verzi, což by nemělo vadit, jde-li o verzi 8.4 a vyšší. I přesto vám doporučujeme, abyste si před odevzdáním svůj kód zkusili zkompilovat a spustit na Aise (nezapomeňte přidat modul s novým GHC), kde máte k dispozici GHC ve verzi 9.0.1. V případě selhání testů už při kompilaci nepřijdete o žádný pokus.
- **Všechny globální funkce musí mít typovou signaturu**.
- V odevzdaném souboru neuvádějte hlavičku `module` (pokud nevíte, o co se jedná, vůbec to nevadí).

Vyhodnocování

- Úkol se vyhodnocuje průběžně, vyhodnocení po nahrání souboru ale **není** okamžité.
  - Automatický testovací nástroj kontroluje soubory v odevzdávárně v pravidelných intervalech několika minut. Podle času odevzdání a vytížení vyhodnocovacího serveru může vyhodnocení trvat několik desítek minut.
  - Pokud se vám výsledky neobjevily cca do půl hodiny a neblíží se zatím čas deadline, dejte nám vědět ve fóru.
- Po vyhodnocení se získané body a případný výpis testů, které na vašem řešení selhaly, **objeví v poznámkovém bloku**.
  - U nesprávně implementovaných funkcí se dozvíte příklad vstupu, na němž se váš výsledek neshoduje s očekávaným.
  - Nejprve jsou v bloku uvedeny vstupy, následně výstup učitelského řešení a váš výstup.
- Máte **pět možností odevzdání**, započítává se nejlepší z nich.
  - Pokud při odevzdání neprojde kontrola syntaxe, tak se do tohoto limitu nepočítá, pokud však kontrola projde, je automaticky započítáno a nelze to zvrátit.
- Další odevzdání provedete tak, že do odevzdávárny nahrajete novou verzi.
- Vzhledem k prodlevám při vyhodnocování neodkládejte práci na poslední chvíli, ať možnost vícenásobného odevzdání v případě potřeby vůbec stihnete využít.
  - S blížícím se termínem uzavření odevzdáváren očekávejte větší (i několikahodinové) prodlevy.
  - Není žádná garance rychlosti vyhodnocování (může se tedy stát, že výsledky řešení odevzdaného v neděli ve 21.00 neuvidíte do konce deadline).

S odevzdávárnou zacházejte s rozvahou, abyste nepřišli o možnosti odevzdání. I když nahrajete nové řešení ještě před zveřejněním výsledku v poznámkovém bloku, vyhodnocovací nástroj už může mít (a pravděpodobně má) vaše dřívější odevzdání ve frontě. Z jeho pohledu tak došlo ke dvěma odevzdáním a vy si vyplýtváte jeden pokus. Podobně se vám mohou započítat odevzdání navíc, pokud do odevzdávárny omylem vložíte více než jeden soubor nebo pokud soubor přejmenujete (každý soubor se vezme jako samostatné odevzdání).