# Kapitola 1

# Halda

## 1.1 Halda

Halda je často používaná datová struktura, která slouží k rychlému hledání minima. Dostaneme $n$ prvků a chceme najít nejmenší z nich, tedy jejich minimum.

Pokud hledáme jen jeden nejmenší prvek, tak ho najdeme průchodem všech prvků. V každém kroku porovnáme procházený prvek s dosud nalezeným minimem. Pokud je procházený prvek menší, tak jsme nalezli nové dosavadní minimum. V podsekci **??** o časové složitosti jsme si ukázali, že je pro nalezení minima potřeba alespoň $n-1$ porovnání. Proto je hledání nejmenšího prvku pomocí průchodu nejlepším možným řešením.

Co když chceme najít i druhý nejmenší prvek? Nebo co když chceme najít $k$ nejmenších prvků?

Jednoduchým řešením je nalézt nejmenší prvek, odebrat ho z množiny prvků[1] a ve zbývajících prvcích hledat minimum stejným způsobem. Tím dostaneme časovou složitost $\mathcal{O}(kn)$.[2] Můžeme najít následující nejmenší prvky rychleji?

Zajímavý nápad je rozdělit si zadaná čísla na dvě části o $n_1$ a $n_2$ prvcích. Na $n_1-1$ porovnání najdeme minimum v první části a na $n_2-1$ porovnání ve druhé. Porovnáním minim z obou částí najdeme minimum ze všech $n$ čísel. Celkem jsme potřebovali $n_1-1+n_2-1+1 = n-1$ porovnání. Jak teď najít druhý nejmenší prvek? V jedné části minimum známe, ve druhé ho budeme muset znova spočítat. Pokud jsou části stejně velké, tak už druhé nejmenší číslo najdeme na nejvýše $\lceil n/2 \rceil + 1$ porovnání. Celkem jsme dvě nejmenší čísla nalezli na nejvýše $\lceil 3n/2 \rceil$ porovnání.

Můžete navrhnout, že celou myšlenku můžeme zopakovat i v každé části. Rozdělením obou částí dostaneme čtyři nové části. Ty můžeme zase dále dělit a to tak dlouho, dokud části nejsou jednoprvkové. Tím dostaneme datovou strukturu, které se říká halda. Při práci s haldou používáme i další užitečné operace: přidávání nových prvků, mazání prvků a podobně.

Ještě než přistoupíme k samotné definici, tak jeden problém pro vás. Dostanete $n$ prvků. Jak rychle najdete $\sqrt{n}$ nejmenších z nich? Zvládli byste to v čase $\mathcal{O}(\sqrt{n})$? (viz. cvičení)

**Definice:** *Stromová datová struktura* je reprezentace stromu v počítači (viz definice na straně **??**). V každém vrcholu $v$ si pamatujeme hodnotu $x(v)$, které se říká klíč (key).

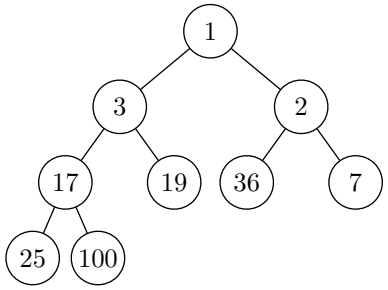
*Halda* je stromová datová struktura splňující vlastnost haldy. Zakořeněný strom

[1] Na místo smazaného prvku překopírujeme poslední prvek a zkrátíme pole o jedna.

[2] Někdo jiný by mohl navrhnout, ať nejprve všechny prvky setřídíme. Potom najdeme $k$ nejmenších prvků v čase $\mathcal{O}(n \log n + k)$. To je ale opět velký čas v případech, kdy je $k$ výrazně menší než $\log n$.

má *vlastnost haldy* právě tehdy, když pro každý uzel $v$ a pro každého jeho syna $w$ platí $x(v) \leq x(w)$. Díky této vlastnosti bude kořen stromu obsahovat nejmenší klíč z celé haldy.

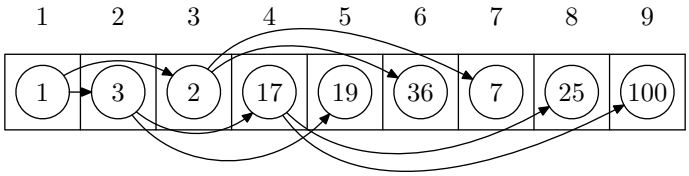
*Binární halda* je úplný binární strom s vlastností haldy. Strom je *binární*, pokud má každý vrchol nejvýše dva potomky. Binární strom je *úplný*, pokud jsou všechny jeho hladiny kromě poslední úplně zaplněny a v poslední hladině leží listy co nejvíce vlevo. Úplnost binárního stromu zaručuje hezký vyvážený tvar stromu a to nám garantuje výšku stromu nejvýše $\lceil \log n \rceil$. Příklad binární haldy je na obrázku vpravo.[3]




**Operace:** s haldou běžně provádíme následující operace:

- `MIN` – vrátí nejmenší klíč v haldě
- `DELETE_MIN` – vymaže uzel s nejmenším klíčem
- `INSERT`($h$) – přidá nový uzel s hodnotou $h$
- `DELETE`($v$) – smaže uzel $v$
- `DECREASE_KEY`($v$, *okolik*) – uzlu $v$ zmenší hodnotu klíče o *okolik*
- `INCRESE_KEY`($v$, *okolik*) – uzlu $v$ zvětší hodnotu klíče o *okolik*
- `MAKE` – dostane pole $n$ prvků a vytvoří z nich haldu.

Binární haldu si můžeme snadno reprezentovat v poli $\mathtt{x}[\,\cdot\,]$. Využijeme při tom úplnosti binárního stromu. Uzly stromu očíslujeme po hladinách počínaje od jedničky. Těmito čísly budeme uzly označovat. Uzel $i$ uložíme do $\mathtt{x}[i]$. Levý syn uzlu $k$ bude uložen na pozici $2k$ a pravý syn uzlu $k$ na pozici $2k+1$. Naopak otec vrcholu $k$ se bude nacházet na pozici $\lfloor k/2 \rfloor$. Na následujícím obrázku jsme takto do pole poskládali binární strom z předchozího obrázku.




Aby se nám s haldou lépe pracovalo, zavedeme si dvě pomocné funkce `BUBBLE_UP`($v$) a `BUBBLE_DOWN`($v$). Bublání funguje stejně jako v třídění pomocí bublinkového algoritmu, akorát místo průchodu pole probubláváme podél cesty ve stromě vedoucí z uzlu $v$ do kořene nebo do listu. `BUBBLE_UP` zajistí probublání lehkých prvků směrem nahoru ke kořeni. `BUBBLE_DOWN` naopak zajistí propad těžkých prvků dolů směrem k listům. Nebudeme probublávat podél celé cesty, ale jen dokud v aktuálním vrcholu

[3] **Pohádka:** (Jak si představit fungování haldy?) Zajdeme na kouzelnickou párty, kde si prohlédneme kouzelnickou šíšu. Kouzelnická šíša vypadá jako obrázek haldy. Ten znárodňuje kouzelné baňky, které jsou propojeny velmi úzkými trubičkami. Každá baňka je naplněna jiným plynem – podle toho, co který kouzelník donese. Vrchní baňka je opatřena hadičkou s ventilem, pomocí které lze plyn z horní baňky odpouštět a ochutnávat.

Lehčí plyny se snaží stoupat vzhůru a těžší naopak klesají. Podívejme se na dvě sousední baňky, které jsou spojeny trubičkou. Pokud spodní baňka obsahuje lehčí plyn než horní baňka, tak začne lehčí plyn probublávat do horní baňky. Těžší plyn začne naopak klesat a probublávat dolů. Jsou to kouzelná baňky. Pokud má baňka na výběr, tak kouzlo šíšové seance pohlídá, aby do baňky probublal jen ten lehčí ze dvou plynů.

Tato kouzelná šíša zajišťuje, aby se při kouzelnických dýcháncích mohli odpouštět plyny ve správném pořadí. Jinak by z toho kouzelními mohli mít střevní potíže (plyny by probublávali ve střevech stejně, jako to mohou dělat mezi baňkami).

není splněna vlastnost haldy. Pomocí `swap(`$i$`, `$j$`)` značíme prohození dvou prvků na pozicích $i$ a $j$ v poli `x`$[\,\cdot\,]$.

```
BUBBLE_UP(k):
   while k > 1 and x[⌊k/2⌋] > x[k] do
      swap(k, ⌊k/2⌋)
      k := ⌊k/2⌋

BUBBLE_DOWN(k):
   while k ≤ ⌊n/2⌋ do
      min := 2k {min bude pozice syna s menším klíčem }
      if 2k + 1 ≤ n and x[2k] > x[2k + 1] then
         min := 2k + 1
      if x[min] < x[k] then
         swap(k, min)
       else
         break
      k := min
```

Časová složitost obou probublávacích funkcí je nejvýše tolik, kolik je výška úplného binárního stromu a to je $\mathcal{O}(\log n)$. Pomocí pomocných funkcí už snadno naimplementuje ostatní operace haldy.

```
MIN:
   return x[1]

DELETE_MIN:
   x[1] := x[n]
   n := n − 1
   BUBBLE_DOWN(1)

INSERT(h):
   n := n + 1
   x[n] := h
   BUBBLE_UP(n)

DELETE(k):
   val := x[k]
   x[k] := x[n]
   n := n − 1
   if val ≤ x[k] then
      BUBBLE_DOWN(k)
   else
      BUBBLE_UP(k)

DECREASE_KEY(k,okolik):
   x[k] := x[k] − okolik
   BUBBLE_UP(k)

INCRESE_KEY(k,okolik):
   x[k] := x[k] + okolik
   BUBBLE_DOWN(k)
```

Časová složitost `MIN` je konstantní. Časová složitost ostatních výše uvedených operací je stejná jako časová složitost `BUBBLE_UP` nebo `BUBBLE_DOWN` a to je $\mathcal{O}(\log n)$.

Podívejme se na to, jak z $n$ prvků na vstupu postavit haldu. Jednoduše bychom mohli začít s prázdnou haldou a $n$-krát zavolat operaci `INSERT`. To by mělo časovou složitost $\mathcal{O}(n \log n)$. My si ale ukážeme, jak postavit haldu z $n$ prvků v poli v lineárním čase. Prvky necháme v poli $\mathtt{x}[\,\cdot\,]$ tak, jak jsme je dostali na vstupu a nad polem si představíme binární strom. Naším cílem je přeuspořádat prvky tak, aby splňovaly vlastnost haldy. Dosáhneme toho postupným voláním `BUBBLE_DOWN`.

```
MAKE:
  for i := ⌊n/2⌋ downto 1 do
    BUBBLE_DOWN(i)
```

Proč po skončení `MAKE` splňuje každý vrchol vlastnost haldy? Platí invariant, že v každém kroku algoritmu splňují všechny vrcholy $j$, pro $i \leq j \leq n$, vlastnost haldy. V následujícím kroku necháme klíč ve vrcholu $i-1$ probublat dolů, takže také vrchol $i-1$ bude splňovat vlastnost haldy. U vrcholů $j$, pro $i \leq j \leq n$, se tím vlastnost haldy neporuší.

Nyní dokážeme, že postavení haldy pomocí `MAKE` bude trvat jen $\mathcal{O}(n)$. Nechť $h = \lceil \log n \rceil$ je výška úplného binárního stromu na $n$ vrcholech. Operace `BUBBLE_DOWN` zavolaná na uzel v $k$-té hladině od zdola bude trvat čas přímo úměrný $k$. Z vlastností binárního stromu víme, že v následující hladině stromu je dvojnásobek prvků. Proto je v $k$-té hladině od zdola $2^{h-k}$ prvků. Označíme-li časovou složitost operace `MAKE` pomocí $X$, dostaneme

$$X = \sum_{k=1}^{h} 2^{h-k} \cdot k.$$

Sumu lze spočítat fintou tak, že ji vynásobíme dvěma a odečteme od ní tu samou sumu. Koeficienty před stejnou mocninou dvojky se krásně odečtou a zbude nám jednoduchá suma. Konkrétně pro člen $2^{h-k}$ máme $2 \cdot k2^{h-k} - (k-1)2^{h-k+1} = 2^{h-k}$. Celkem dostaneme

$$X = 2X - X = 2^h + \sum_{k=1}^{h-1} 2^{h-k} - 2^0 h = \sum_{j=0}^{h} 2^j - 1 - h = 2n - \lceil \log n \rceil = \mathcal{O}(n).$$

**Poznámka k implementaci:** Operace `BUBBLE_UP`$(k)$ a `BUBBLE_DOWN`$(k)$ můžeme implementovat lépe. Místo prohazování dvou sousedních prvků na procházené cestě $P$ ve stromě si zapamatujeme počáteční hodnotu $val := \mathtt{x}[k]$, na cestě $P$ budeme procházené prvky posouvat o jedna a hodnotu $val$ uložíme až na konečnou pozici.

**Poznámka:** Existují i jiné implementace haldy a haldových operací.

- (pole) Nejjednodušší realizace haldových operací je v poli. Prvky necháme v poli tak, jak jsme je dostali. Nalezení minima realizujeme průchodem celého pole v čase $\mathcal{O}(n)$. Všechny ostatní operace realizujeme přímým přístupem do pole v čase $\mathcal{O}(1)$. Přidávané prvky vložíme na konec pole a zvětšíme velikost pole o 1. Mazání prvku provedeme tak, že mazaný prvek nahradíme posledním prvkem a pole o jedna zkrátíme.

- ($d$-regulární halda) Zobecněním binární haldy je *$d$-regulární halda.* Od binární se liší pouze tím, že každý vrchol má nejvýše $d$ synů. Podmínka na úplnost stromu (zaplněnost hladin) zůstává. Více se o $d$-regulární haldě dozvíte ve cvičeních.

- (Fibonacciho halda) Fibonacciho halda pochází od Fredmanna a Tarjana. Fibonacciho halda pro změnu slevuje z úplnosti stromu, ale stále vyžaduje rozumnou zaplněnost hladin ("košatost" binárních stromů). Fibonacciho halda je množina stromů splňujících vlastnost haldy. Stromy v haldě jsou různého stupně 0, 1, 2,..., $k$. Stupeň stromu zhruba odpovídá stupni grafu v kořeni. S rostoucí vzdáleností od kořene klesá i maximální povolený stupeň ve vrcholech stromu. Dá se ukázat, že strom stupně $d$ obsahuje alespoň $\Phi^d$ vrcholů, kde $\Phi = (1+\sqrt{5})/2$.

  Nejhorší případ stromu stupně $d$, tj. nejmenší a tedy i nejméně košatý strom, jaký je povolen, se konstruuje složením nejmenších stromů stupně $d-1$ a $d-2$. Počet vrcholů v nejmenším stromu stupně $d$ je $F_d = F_{d-1} + F_{d-2}$. Formulka je stejná jako při výpočtu Fibonacciho čísel, proto se této haldě říká Fibonacciho halda.

  Fibonacciho halda realizuje operace `MIN`, `INSERT`, `DECREASE_KEY`, `INCRESE_KEY` v amortizovaném čase $\mathcal{O}(1)$ a operace `DELETE_MIN` a `DELETE` v amortizovaném čase $\mathcal{O}(\log n)$. Implementace této haldy je podstatně složitější a konstanty před časovými složitostmi jednotlivých operací jsou poměrně vysoké. Na druhou stranu použitím Fibonacciho haldy můžeme dosáhnout podstatně lepších asymptotických časových složitostí.

Následující tabulka shrnuje časové složitosti jednotlivých operací při různých reprezentacích.

| | `DELETE_MIN` | `DELETE` | `INSERT` `DECREASE_KEY` | `INCRESE_KEY` |
|---|---|---|---|---|
| pole | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| binární halda | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| $d$-regulární halda | $\mathcal{O}(\frac{d \log n}{\log d})$ | $\mathcal{O}(\frac{d \log n}{\log d})$ | $\mathcal{O}(\frac{\log n}{\log d})$ | $\mathcal{O}(\frac{d \log n}{\log d})$ |
| Fibonacciho halda | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

Pozor, u Fibonacciho haldy je všude uvedena amortizovaná časová složitost.

Všimněme si, jak parametr $d$ u $d$-regulární haldy interpoluje mezi polem a binární haldou (pro $d = 2$ dostaneme binární haldu a pro $d = n$ realizaci v poli). Optimální hodnota parametru $d$ se hledá tak, aby celková časová složitost algoritmu, ve kterém haldu používáme, byla co nejmenší.

## 1.2 Prioritní fronta

**Motivace:** V počítači je fronta úloh čekajících na zpracování. Čekající úlohy „sedí" v paměti v dlouhé řadě židlí, které běžně říkáme pole. Pokud přibude nová úloha, tak se posadí na židli na konci fronty. Pan Procesor úlohy postupně zpracovává. Když řekne „další pán na holení", tak si vyzvedne úlohu na začátku fronty. Takhle funguje obyčejná fronta. Na její realizaci není nic složitého.[4]

Co když se ale přižene systémová úloha, která začne tvrdit, že je důležitější než ostatní úlohy, a začne předbíhat ve frontě? Pokud budeme chtít důležitější úlohy upřednostnit, tak každé úloze přiřadíme její prioritu. Představme si ji jako číselnou hodnotu. Úloha s vyšší prioritou může předběhnout všechny úlohy nižší prioritou.

[4]Pokud budeme mít dostatečně velké pole, tak nemusíme úlohy přesazovat. Vyzvednutí úlohy, která je na řadě, i posazení nově příchozí úlohy nám zabere jednotkový čas. Pokud bychom chtěli šetřit místem, tak můžeme řadu židlí stočit do kruhu. Díky tomu se budou uvolněné židle automaticky „recyklovat" tím, že se jakoby přesunou na konec řady. Ve skutečnosti nic nepřesunujeme, jen si posuneme ukazovátko určující začátek fronty. Díky tomu bude vyzvednutí úlohy, která je na řadě, i posazení nově příchozí úlohy trvat jednotkový čas. (Nikoho nemusíme přesazovat.)

Jak ale teď bude fungovat vyzvedávání úlohy, která je na řadě (má nejvyšší prioritu)? A kam posadíme nově příchozí úlohy? Frontu můžeme v poli udržovat setříděnou podle priorit. Odebrání úlohy, která je na řadě, se nebude lišit od obyčejné fronty. Odebereme první prvek pole. Pro přidání nové úlohy musíme nejprve přesadit úlohy s nižší prioritou o jedno místo zpátky a tím si vytvořit volnou židli pro nově příchozí úlohu. To ovšem vyžaduje až $\mathcal{O}(n)$ přesazování.

Navrhněte systém fungování prioritní fronty tak, aby přidání nové úlohy a i odebrání úlohy s nejvyšší prioritou, fungovalo co nejefektivněji. To je abychom museli přesadit co nejméně úloh. Není nutné, aby byly úlohy v poli seřazeny podle priority.

**Úkol:** Obyčejná fronta je seznam prvků seřazený podle času příchodu. Prioritní fronta je seznam prvků seřazený podle priorit. Každý prvek má svojí hodnotu, tzv. prioritu. Prvky s vyšší prioritou mohou ve frontě předběhnout prvky s nižší prioritou. Pokud mají dva prvky stejnou prioritu, tak jsou seřazeny podle času příchodu.

Pro práci s prioritní frontou potřebujeme umět odebrat prvek, který je na řadě (ten s nejvyšší prioritou), přidávat a mazat prvky a také u některých prvků měnit prioritu. A to vše co nejefektivněji.

Pro jednoduchost předpokládejme, že vyšší priorita odpovídá nižší číselné hodnotě. Prvky s nejvyšší prioritou jsou tedy ty s nejnižší číselnou hodnotou.[5]

**Řešení pomocí pole:** V tomto řešení je $\mathtt{x}[\,\cdot\,]$ neuspořádané pole. Nemusíme nic vytvářet. Operace nalezení minima spočívá v průchodu pole a trvá čas $\mathcal{O}(n)$. Ostatní operace lze realizovat v konstantním čase.

Použitím setříděného pole si moc nepomůžeme. Zkuste se zamyslet nad realizací a časovou složitostí jednotlivých operací. Dostaneme horší výsledek než při použití haldy.

**Řešení pomocí haldy:** Použijeme haldu, kde klíčem uzlů/úloh bude jejich priorita. Operace požadované po prioritní frontě přímo odpovídají operacím pro práci s haldou. Například odebrání prvku s nejvyšší prioritou zajistí funkce `MIN` a `DELETE_MIN`, změnu priority funkce `INCRESE_KEY` a `DECREASE_KEY` apod. Časová složitost operací je $\mathcal{O}(\log n)$.

## 1.3 Příklady

1. (nalezení prvních $\sqrt{n}$ nejmenších čísel) Dostaneme pole $n$ prvků $\mathtt{A}[1,\ldots,n]$. Najděte algoritmus, který vypíše prvních $\sqrt{n}$ nejmenších čísel. Umíte to v čase $\mathcal{O}(n)$?

   *Nápověda:* čtverec.

2. (Max-heap) Vymyslete podobnou datovou strukturu jako je halda, ale tentokrát chceme odebírat prvek s největší hodnotou.

3. (Heapsort) Zkuste vymyslet jednoduchý třídící algoritmus využívající haldy. Jaká bude jeho časová složitost v nejhorším případě? Umíte ho realizovat tak, abychom potřebovali pouze to pole, ve kterém dostaneme vstup?

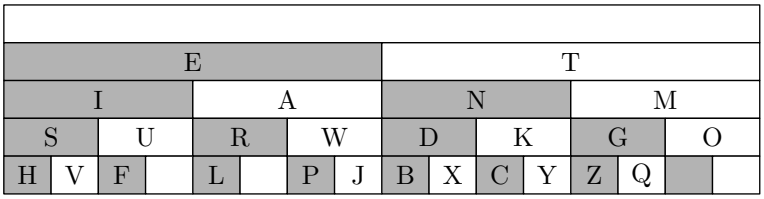
4. ($d$-regulární halda) Binární halda je halda, kde má každý vrchol nejvýše dva syny. V $d$-regulární haldě má každý vrchol nejvýše $d$ synů. Zkuste zobecnit binární haldu na $d$-regulární haldu. Jak se $d$-regulární halda uloží do pole? Na které pozici budou synové vrcholu z pozice $k$? Na jaké pozici bude jeho otec? Jak se změní implementace haldových operací? Jaká bude jejich časová složitost?

[5] Čtenář by jistě zvládl prohodit maxima za minima, ale nechme to stejné jako v zavedení haldy.

5. (Kalendář událostí) Jak byste realizovali kalendář událostí? Kalendář událostí funguje podobně jako váš diář. Průběžně dostáváme úlohy, které mají přesně stanovený čas, kdy se mají vykonat. Můžeme je dostávat v libovolném pořadí, tedy ne nutně seřazené podle času. Úlohy zpracováváme v pořadí podle času. Při zpracovávání úlohy se dozvíme, jaké další úlohy přibyly. Občas naopak zjistíme, že máme nějakou naplánovanou úlohu zrušit, případně ji přeplánovat na jiný čas.

   Při realizaci se stačí zamyslet nad následujícími funkcemi: Odebrání úlohy, která je na řadě. Přidání nové úlohy, smazání a přeplánování konkrétní úlohy.

6. (Využití reprezentace binárního stromu v poli) Napište co nejrychlejší program, který čte ze vstupu morseovku a překládá ji do anglické abecedy. Využijte při tom vyhodnocovacího stromu z následujícího obrázku.[6] Binární strom na obrázku je netradičně nakreslený jako tabulka.




   Při vyhodnocování kódu písmene v morseovce (například `.-`) začneme v prvním řádku (v kořenu stromu). Pokud je první znak tečka, přesuneme se o řádek níže doleva. Pokud je první znak čárka, přesuneme se o řádek níže doprava. Dostali jsme se do dalšího vrcholu stromu. Tento postup opakujeme, dokud nezpracujeme všechny znaky aktuálního písmene v morseovce. Skončíme ve vrcholu označeném písmenem, které odpovídá Morseovu kódu.

   Binární vyhodnocovací strom si reprezentujte v poli, například jako řetězec "␣`ETIANMSURWDKGOHVF`␣`L`␣`PJBXCYZQ`␣␣", kde ␣ představuje mezeru. Předpokládejte, že vstup obsahuje pouze znaky '`.`', '`-`', '`|`' a že je celý vstup je ukončen znakem '`$`'. Tak "`....|---|-..|-.|.|...|-|.|...|-|..`".

   *Nápověda:* Jediný cyklus + stavový automat pamatující si pozici ve vyhodnocovacím stromě.

[6] Pro zjednodušení práce jsme použili anglickou verzi morseovky, bez písmene CH. To by nám lehce komplikovalo práci tím, že se skládá ze dvou znaků.