Lano

Typické použití textového editoru spočívá v otevření existujícího souboru a úpravě jeho obsahu vložením nových znaků na libovolné místo doprostřed načteného textu, či naopak mazáním znaků zprostřed textu. Ukládat obsah otevřeného souboru do běžného řetězce by však vedlo k nepřijatelně vysoké latenci takového editoru. Běžný typ `String` je pro tyto účely zvlášť nevhodný, neboť jeho indexování vyžaduje projít seznam od začátku. Ovšem ani typická imperativní implementace řetězců pomocí pole znaků nestačí, jelikož vkládání doprostřed či mazání zprostřed takového pole je v nejhorším případě opět lineární vůči délce řetězce.

Chceme-li nějaký řetězec často upravovat, je vhodné jej reprezentovat pomocí datové struktury *lano* (rope), kterou bude vaším úkolem naprogramovat. Lano je binární strom, ve kterém:

- Každý list uchovává fragment řetězce a jeho délku.
- Každý vnitřní uzel obsahuje odkaz na levého a pravého syna a též *váhu*, což je celková délka řetězce uloženého v jeho *levém* podstromě. Řetězec reprezentovaný vnitřními uzly lze získat jako zřetězení řetězce reprezentovaného levým synem a řetězce reprezentovaného pravým synem.

Odevzdávat budete jeden soubor s modulem `Rope`, který exportuje entity popsané dále.

Reprezentace

Definujte typový alias

```
type StringRep = Array Int Char
```

kde `Array` je typ poskytovaný modulem `Data.Array`. Tento typ poskytuje funkcionální rozhraní k paměťově souvislým polím, které je možné indexovat efektivně (tedy v konstantním čase). `StringRep` je tedy pole znaků indexované celými čísly.

Dále definujte rekurzivní typ reprezentující lano

```
data Rope
  = Leaf StringRep    -- ^ Leaf, contains a single string
  | Concat            -- ^ Inner node, represents a concatenation
          Int         -- ^ Weight
          Rope        -- ^ Left child
          Rope        -- ^ Right child
  deriving (Show)
```

Chcete-li, můžete hodnotové konstruktory tohoto typu definovat jako *záznamy*. Také můžete dle libosti přidat instance typových tříd (např. pomocí `deriving`), neměli byste to však potřebovat. Modul `Rope` exportuje pouze typ `Rope`, a to zapouzdřeně, tedy bez exportování konstruktorů.

Požadované funkce

Definujte a exportujte následující funkce pro práci s lany:

```
emptyRope :: Rope
stringToRope :: String -> Rope
ropeToString :: Rope -> String
indexRope :: Rope -> Int -> Maybe Char
concatRopes :: Rope -> Rope -> Rope
splitRope :: Rope -> Int -> (Rope, Rope)
updateRope :: Rope -> Int -> Char -> Rope
insertIntoRope :: Rope -> Int -> String -> Rope
```

Funkce `emptyRope` vytvoří lano reprezentující prázdný řetězec. Funkce `stringToRope` vytvoří lano se zadaným řetězcem jako svým jediným listem. Naopak, `ropeToString` vytvoří ze zadaného lana standardní Haskellový řetězec.

Ostatní funkce vyžadují implementaci komplikovanějších algoritmů:

`indexRope` dostane lano a index a vrací znak na dané pozici v řetězci reprezentovaném lanem nebo `Nothing`, pokud pozice není validní. Validními pozicemi se myslí indexy z rozsahu `0` až `n - 1`, kde `n` je délka řetězce

lana. Tato funkce nesmí procházet celý strom: počet rekurzivních zanoření musí (asymptoticky) odpovídat výšce stromu.

`concatRopes` dostane dvě lana reprezentující řetězce `A` a `B` a vrátí lano reprezentující jejich spojení (konkatenaci), tedy řetězec `AB`. Na to budete potřebovat vypočítat váhu nově vzniklého uzlu (tedy délku řetězce uloženého v laně `A`). Uvědomte si, že toto je však možné provést bez procházení celého lana `A` sečtením vah jedné konkrétní cesty v tomto laně. Funkce opět musí být v tomto smyslu efektivní.

`splitRope` dostane lano `A` reprezentující řetězec $a_0a_1 \dots a_{n-1}$ a index `i` a vrátí dvojici s lanem `B` reprezentujícím řetězec $a_0a_1 \dots a_{i-1}$ a lanem `C` reprezentujícím $a_ia_{i+1} \dots a_{n-1}$. Tato funkce funguje následovně:

- Pokud má rozdělit list, vrátí dva listy s odpovídajícími částmi řetězce.
- Jinak se rozhoduje podle váhy `w` současného vnitřního uzlu:
  - Pokud `i = w`, pak je vrácena dvojice podstromů současného uzlu.
  - Pokud `i < w`, pak se rekurzivně zavolá na levém podstromě s výsledkem `(l, r)`, přičemž výsledek celé operace je `(l, r zřetězeno s pravým podstromem)`.
  - Pokud `i > w`, pak se rekurzivně zavolá na pravém podstromě s indexem `i' = i - w` s výsledkem `(l, r)` a výsledek celé operace je `(levý podstrom zřetězený s l, r)`.

Součástí zadání je samozřejmě implementovat tento algoritmus pěkně.

`updateRope` dostane lano `A`, index `i` a znak `c` a vrací lano, které reprezentuje stejný řetězec jako `A` až na index `i`, kde je znak `c`. Pokud je index `i` mimo rozsah, funkce zadané lano vrátí bez úpravy. Opět musíte funkci implementovat dostatečně efektivně, podobně jako funkci `indexRope`.

`insertIntoRope` dostane lano `A` reprezentující řetězec $a_0a_1 \dots a_{m-1}$, index `i` a řetězec `str` $= s_0s_1 \dots s_{n-1}$. Vrací lano reprezentující řetězec $a_0a_1 \dots a_{i-1}s_0s_1 \dots s_{n-1}a_ia_{i+1} \dots a_{m-1}$. Funkce tedy vloží zadaný řetězec před znak na indexu `i`. Je-li `i` menší než nula, funkce se chová, jako by bylo rovno nule, a pokud je větší než počet znaků v řetězci, funkce se chová, jako by bylo rovné právě počtu znaků v řetězci, tedy vkládá na konec. Dejte si zde pozor, abyste neduplikovali již napsaný kód.

Příklady

```
ropeToString emptyRope
~> ""

stringToRope "Hello World!"
~> Leaf (array (0,11) [(0,'H'),(1,'e'),(2,'l'),(3,'l'),(4,'o'),(5,' '),(6,'W'),
   (7,'o'),(8,'r'),(9,'l'),(10,'d'),(11,'!')])

smallTestRope :: Rope
smallTestRope =
  Concat 6
    (Concat 2
      (stringToRope "I ")
      (stringToRope "love"))
    (stringToRope " Haskell!")

stringToRope smallTestRope
~> "I love Haskell!"

indexRope smallTestRope 0
~> Just 'I'
indexRope smallTestRope 12
~> Just 'l'
indexRope smallTestRope (-4)
~> Nothing

updateRope smallTestRope 5 'X'
~>
Concat 6
```

```
    (Concat 2
        (Leaf (array (0,1) [(0,'I'),(1,' ')]))
        (Leaf (array (0,3) [(0,'l'),(1,'o'),(2,'v'),(3,'X')])))
    (Leaf (array (0,8) [(0,' '),(1,'H'),(2,'a'),(3,'s'),(4,'k'),(5,'e'),(6,'l'),
    (7,'l'),(8,'!')]))

ropeToString $ updateRope smallTestRope 5 'X'
~> "I lovX Haskell!"

rope2 = concatRopes smallTestRope (stringToRope " And you too!")

rope2
~>
Concat 15
    (Concat 6
        (Concat 2
            (Leaf (array (0,1) [(0,'I'),(1,' ')]))
            (Leaf (array (0,3) [(0,'l'),(1,'o'),(2,'v'),(3,'e')])))
        (Leaf (array (0,8)
            [(0,' '),(1,'H'),(2,'a'),(3,'s'),(4,'k'),(5,'e'), (6,'l'),(7,'l'),
            (8,'!')])))
    (Leaf (array (0,12)
        [(0,' '),(1,'A'),(2,'n'),(3,'d'),(4,' '),(5,'y'),(6,'o'),(7,'u'),
        (8,' '),(9,'t'),(10,'o'),(11,'o'),(12,'!')]))

ropeToString rope2
~> "I love Haskell! And you too!"

-- https://upload.wikimedia.org/wikipedia/commons/f/fd/Vector_Rope_split.svg
wikiTestRope :: Rope
wikiTestRope = b
  where
    b = concatRopes c d
    c = concatRopes e f
    d = concatRopes g h
    e = stringToRope "Hello "
    f = stringToRope  "my "
    g = concatRopes j k
    h = concatRopes m n
    j = stringToRope "na"
    k = stringToRope "me i"
    m = stringToRope "s"
    n = stringToRope " Simon"

ropeToString wikiTestRope
~> "Hello my name is Simon"

fst $ splitRope wikiTestRope 0
~> Leaf (array (0,-1) [])

(l, r) = splitRope wikiTestRope 5

ropeToString l
~> "Hello"

ropeToString r
~> " my name is Simon"
```

```
(l', r') = splitRope wikiTestRope 12

ropeToString l'
~> "Hello my nam"

ropeToString r'
~> "e is Simon"

ropeToString $ insertIntoRope wikiTestRope 9 "first "
~> "Hello my first name is Simon"

ropeToString $ insertIntoRope wikiTestRope (-5) "Oh no! "
~> "Oh no! Hello my name is Simon"
```

Rady na závěr

Váš modul musí exportovat specifikované funkce, ale nic vám nebrání vytvářet si (neexportované) pomocné funkce. Naopak, je to vřele doporučeno. Nebojte se využívat vzorů a pokročilé syntaxe jazyka Haskell, jako jsou například stráže. Též si můžete usnadnit práci napsáním jednotkových testů či dokonce QuickCheckové specifikace.

Nastudujte si rozhraní modulu `Data.Array`, není vůbec komplikované na použití. Všimněte si, ve kterých typových třídách datový typ `Array` je.

Neduplikujte kód. Mějte na paměti, jakou funkcionalitu jste již naprogramovali. U řešení, kde je zjevná velká duplikace kódu, nepředpokládáme hodnocení lepší než jedním bodem. Stejně tak dbejte na efektivní implementaci zejména těch funkcí, kde to zadání přímo vyžaduje. Bude-li nějaká taková funkce zjevně neefektivní (například tedy procházet celý strom, přestože zadání požaduje průchod pouze jedné jeho větve), je dvoubodové hodnocení opět velmi nepravděpodobné.

Nebojte se zeptat, pokud vám něco nebude jasné, například na diskuzním fóru. Nedoporučujeme vám vyhledávat informace o této datové struktuře online, neboť internet je zaplevelen nekvalitním materiálem, což v tomto případě bohužel platí i o článku na anglické Wikipedii. Navíc, některé zdroje implementují poněkud jinou variantu lana s váhou rovnou celkové délce řetězce v obou podstromech dohromady, což pak vede na trochu jiné algoritmy.