## Залік

## Допоміжна інформація

*Екземпляри класів типів*

Визначення класу типів Num, як воно задано в модулі Prelude.

*-- | Basic numeric class.*

***class*** Num a ***where***

{-# MINIMAL (+), (\*), abs, signum, fromInteger, (negate | (-)) #-}

(+), (-), (\*) :: a -> a -> a

*-- | Unary negation.*

negate :: a -> a

*-- | Absolute value.*

abs :: a -> a

*-- | Sign of a number.*

signum :: a -> a

*-- | Conversion from an 'Integer'.*

fromInteger :: Integer -> a

x - y = x + negate y

negate x = 0 - x

*-- The functions 'abs' and 'signum' should satisfy the law:*

*--*

*-- > abs x \* signum x == x*

*--*

Тип AbstractInteger один з варіантів представлення цілих чисел, він має три конструктора: Zero (константа) , Succ AbstractInteger (наступник абстрактного цілого) і Pred AbstractInteger(попередник абстрактного цілого).

***data*** AbstractInteger

= Zero

| Succ AbstractInteger

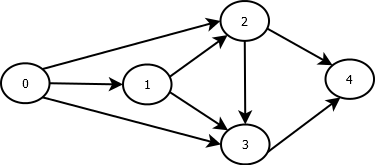
| Pred AbstractInteger

***deriving*** (Show, Eq)

Всі операції над AbstractInteger повинні підтримувати інваріант

*Дане типа AbstractInteger або Zero або результат використання тільки одного з конструкторів Succ / Pred, а не обох. Так Pred (Pred Zero) – коректне дане типа AbstractInteger, але Pred (Succ Zero) – ні, тому що його можна нормалізувати до Zero.*

*Орієнтовані графи*

Орієнтований граф G = (V, E) складається зі скінченної множини вершин V = {v0, v1, …,vm} та скінченної множини ребер E = {e1, e2, …, en}. Кожне ребро – це пара вершин (v,w). По ребру (v,w) можна пройти від вершини v до вершини w. Орієнтований граф може мати петлі, ребра виду (v,v), і не являється мультиграфом, тобто з одної вершини до іншої можна перейти не більше ніж по одному ребру.

*Мал.1. Орієнтований ациклічний граф.*

*Маршрут* (*шлях*) в графі – це список ребер [(v0, v1), (v1, v2), (v2, v3), …, (vk-1, vk)], що веде з початкової вершини v0 в кінцеву vk, в подальшому позначається як список вершин [v0, v1, …, vk]. *Довжина шляху* – кількість реберг k. *Ланцюг* - маршрут, у якого ребра попарно різні. *Простий ланцюг* - маршрут, у якого вершини попарно різні. *Цикл* - замкнений маршрут v0==vk. *Простий цикл (контур)* - замкнений простий маршрут. Контур, що містить всі вершини графу, *- гамільтоновий*.

Граф ациклічний або безконтурний, якщо він немає жодного циклу. Ациклічний граф не може мати петлі.

Топологічне сортування графа G = (V, E), це розташування всіх його вершин в такому порядку [v0, v1, …, vm], що якщо (vi,vj) ϵ E, то вершина vi в списку лівіше вершини vj , тобто [...,vi,…,vj …]. Список [0,1,2,3,4] – топологічне сортування графа на Мал.1. Топологічне сортування можливо лише для ациклічного графу.

Розглядаються орієнтовані графи, вершини яких цілі числа від 0 до m - 0..m.

Для представлення графа (дані типу Graph) використовуються списки суміжності. Граф з (m+1) вершиною – це список довжини (m+1), i-ий список (0≤i≤ m) – це список всіх вершин графа, суміжних з вершиною i.

***type*** Graph = [[Int]]

Список gr1 задає орієнтований граф (Мал.1) gr1 = [[1,2,3], [2,3], [3,4], [4],[]] .

*Різні*

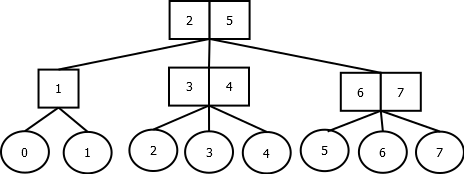
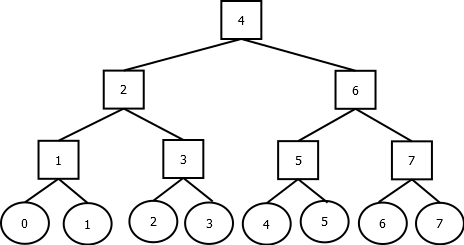
Список цілих чисел зростаючий , якщо кожний попередній елемент менше (<) за наступного. Наприклад [4, 8, 23, 89].

Довільне ціле додатне числу m можна представити в n-ковій системі числення (n <= 16) у вигляді рядка (String). Для кодування цифр більших за ‘9’ використовуються символи ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’. Наприклад представлення числа 21 в різних системах має вид: “10101” - двійкова, “41” - пятіркова, “21” - десяткова, “15” - шістнадцяткова.

Побудова за заданим n-значним десятковим числом виразу і його обчислення. .

* Проміж цифр числа можна розставити (n-1) знаків арифметичних операцій з множини {+, -, \*}, отримуючи цілий арифметичний вираз. Вираз обчислюється зліва направо без урахування пріоритету операцій. Наприклад для числа 427 можна отримати 4+2+7=13, 4\*2-7=1, 4-2-7=-5, 4+2\*7= 42 і т.д.
* Використавши (n-1) знаків арифметичних операцій з множини {+, -, \*} та 2\*(n-1) круглих дужок можна отримати цілий арифметичний вираз в повному записі з дужками, де кожна пара дужок вказує порядок виконання одної з операцій. Вираз обраховується згідно порядку вказаному дужками. Наприклад для числа 427 можна отримати ((4\*2)-7)=1, (4\*(2-7))=-20, ((4-2)-7)=-5, (4-(2-7))=9 і т.д..

*2-3-дерева*

*2-3-деревом* (збалансоване дерево пошука) називається дерево, у якого кожний вузол, що не являється листком, має двох або трьох синів, а довжини всіх шляхів з кореня в листки однакові. Порожнє дерево і дерево з одним листком також являються 2-3-деревами. Вважається, що вся інформація - записи з ключами зберігаються в листках 2-3-дерева, а внутрішні вузли містять ключі для ефективності роботи з 2-3-деревом. Для спрощення, ми вважаємо, в подальшому, що запис складається лише з одного ключа.

*Мал.2. Два еквівалентні 2-3-дерева tr1 і tr2, що задають множину {0,1,2,3,4,5,6,7}*

Означення 2-3 дерева, у якого запис містить лише ключ – значення типу a:

***data*** Tree23 a = Leaf a

| Fork2 (Tree23 a) a (Tree23 a)

| Fork3 (Tree23 a) a (Tree23 a) a (Tree23 a)

| Null23 -- порожнє 2-3-дерево!!!

***deriving*** (Eq, Show)

2-3-дерево задовольняє властивостям:

* 2-3 дерево має три різні види вершин
  + *Leaf k* : листок, що містить одне значення *k*
  + *Fork2 tl x tr* : 2-розгалуження, що містить ліве *tl* і праве *tr* піддерева та значення *x*.
    - Для всіх значень *v* з піддерева tl *v<= x.*
    - Значення *x* - мінімальне в піддереві *tr*
  + *Fork3 tl x tm y tr* : 3-розгалуження, що містить ліве tl, середнє tm і праве tr піддерева та значення x і y.
    - Для всіх значеннь *v* з піддерева tl *v<= x.*
    - Значення *x* - мінімальне в піддереві *tm*
    - Для всіх значеннь *v* з піддерева tm *v<= y*.
    - Значення *y* - мінімальне в піддереві *tr*
* Всі листки знаходяться на одному рівні

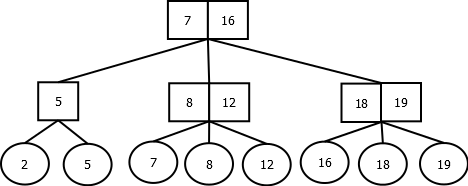
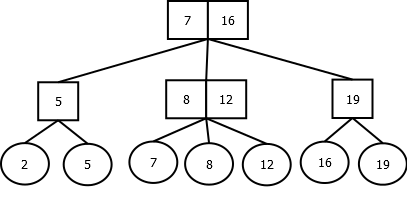
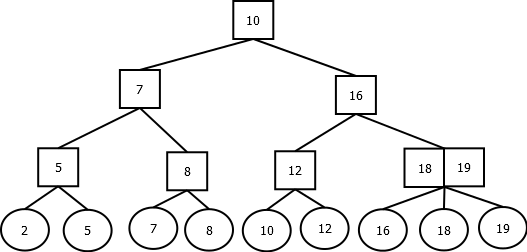
Одна і та ж множина записів може представлятися різними 2-3-деревами. На Мал. 2 показано два різні 2-3-дерева, що задають одну і туж множину натуральних чисел від 0 до 7.

Два 2-3-дерева,що задають одну і туж множину записів, називаються еквівалентними.

Зауважимо, що 2-3-дерево може мати декілька записів з однаковими ключами і вони можуть задавати мультимножини.

Щоб знайти запис з ключом v, проходять дерево від кореня до відповідного листка. Проходячи внутрішній вузол *Fork2 tl x tr* або *Fork3 tl x tm y tr*, порівнюють значення ключа *v* з ключами *x* і *y*, щоб визначити в якого сина *tl*, *tm* або *tr*, продовжити пошук. Наприклад, якщо для вузла *Fork3 tl x tm y tr* виконується умова *x <= v <y*, то пошук потрібно продовжити у сина *tm*.

Вставка нового запису з ключом проходить в три кроки:

* *Пошук.* Як при пошуку виконується прохід від кореня дерева до внутрішнього вузла сином якого повинен стати запис з ключом . (Такі внутрішні вузли, сини яких є листки дерева, називаються термінальними.)
* *Вставка.* Виконати вставку нового запису, при цьому якщо
  + Вузол має два листка, то створюється новий термінальний вузол з трьома листками, котрий повертається як результат.
  + Вузол має три листки, то створюються два нових термінальних вузла, котрі повертаються як результат і виконується коригування структури дерева при поверненні до кореня дерева.
* *Коригування.* Виконати коригування 2-3-дерева, повертаючись тим же шляхом до кореня:
  + Якщо в результаті вставки утворився додатковий вузол, то потрібно відновити баланс по висоті, що може привести до збільшення висоти 2-3-дерева.

*Мал.3. Вставка в 2-3-дерево нових записів 18 і 10*

Для реалізації операції вставки нового значення в 2-3-дерево можна використати функції:

* *isTerminal* :: (Ord a) => Tree23 a -> Bool

*isTerminal tr* – предикат , що перевіряє чи являється корінь 2-3-дерева термінальним вузлом.

* *insTerm* :: (Ord a) => a -> Tree23 a -> (Tree23 a, Maybe (a, Tree23 a))

*insTerm v tr* – додає в 2-3-дерево *tr,* корінь якого термінальний вузол, значення *v*, повертаючи:

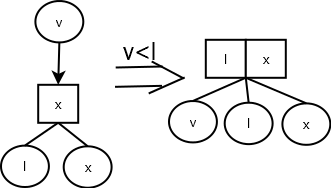
* + *(t1, Nothing) –* нове розширене дерево *t1*
  + *(t1, Just (w,t2))* – два нових дерева *t1* і *t2*, *w* - найменше значення в дереві *t2*
  + На Мал. 4 показано два варіанта (з семи) вставки в термінальний вузол
    - *insTerm v (Fork2 (Leaf l) x (Leaf x)*) =

(Fork3 (Leaf v) l (Leaf l) x (Leaf x), Nothing)

* + - *insTerm v (Fork3 (Leaf l) x (Leaf x) y (Leaf y))* =

(Fork2 (Leaf v) l (Leaf l), Just (x, Fork2 (Leaf x) y (Leaf y))

Зображення, що містить годинник

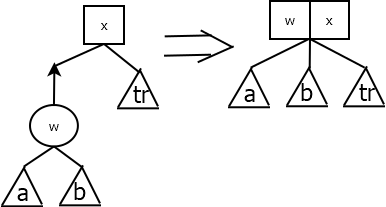
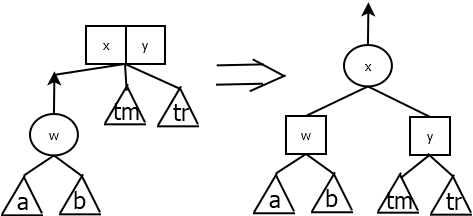
Автоматично згенерований опис

*Мал. 4. Приклади вставки значення в термінальний вузол з двома і трьома листками.*

* *insFork* :: (Ord a) => a -> Tree23 a -> (Tree23 a, Maybe (a, Tree23 a))

*insFork v t* – додає в 2-3-дерево *t,* корінь якого нетермінальний вузол, значення *v*, повертаючи:

* + *(t1, Nothing) –* нове розширене дерево *t1*
  + *(t1, Just (w,t2))* – два нових дерева *t1* і *t2*, *w* - найменше значення в дереві *t2*
  + На Мал. 5 показано два варіанта (з пяти) коригування внутрішніх вузлів 2-3-дерева.
    - *insFork v (Fork2 tl x tr) : v <x* - в результаті виконання вставки v в дерево tl повернуто (a, Just (w,b)) => ( Fork3 a w b y tr, Nothing)
    - *insFork v (Fork3 tl x tm y tr) : v <x* - в результаті виконання вставки v в дерево tl повернуто (a, Just (w,b)) => (Fork2 a w b, Just (x, Fork2 tm y tr))



*Мал. 5. Приклади коригування 2-3-дерева, коли створено два дерева при вставці.*

## Задачі

В допоміжному файлі, котрий включає визначення типів і тестові дані, необхідно надати визначення вказаних функцій і встановіть типи екземплярами необхідних класів.

*Екземпляри класів типів*

1. Встановіть *AbstractInteger* екземпляром *типів Ord*, визначивши власну функцію порівняння.
   * (Succ(Succ Zero)) > (Succ Zero) = True
2. Функція *aiToInteger ai,* котра перетворює *ai* дане типу AbstractInteger в Integer Наприклад:
   * aiToInteger Zero = 0
   * aiToInteger (Pred (Pred Zero)) = -2
3. Функція *plusAbs ai1 ai2,* котра додає *ai1* і *ai2 -* дані типу AbstractInteger, її результат дане типу AbstractInteger. *Не визначайте* її шляхом трансформування додавання для звичайних Integer. Наприклад:
   * plusAbs (Pred (Pred Zero)) (Succ (Succ Zero)) = Zero
   * plusAbs (Pred Zero) (Succ (Succ Zero)) = Succ Zero
4. Функція *timesAbs ai1 ai2t*, котра множить *ai1* і *ai2 -* дані типу AbstractInteger, її результат дане типу AbstractInteger. Наприклад:
   * timesAbs (Pred (Pred Zero)) (Pred (Pred (Pred Zero))) =

*=* Succ( Succ ( Succ (Succ (Succ (Succ Zero)))))

1. Встановіть *AbstractInteger* екземпляром класу типів *Num* шляхом визначення відповідних функцій для цього класу.
   * negate (Succ(Succ Zero) = (Pred (Pred Zero)))
   * fromInteger 3 == Succ(Succ(Succ Zero)) = True
   * abs Zero = Zero

*Орієнтовані графи*

1. Функція *gamiltonWay gr*, що шукає в орієнтований граф *gr* гамільтоновий цикл (контур). Якшо граф не має гамільтонового циклу, то функція повертає Nothing.
   * gamiltonWay gr1 = Nothing
   * gamiltonWay gr2 = Just [0,4,1,3,2,0]
2. Предикат *isAcyclic gr*, котрий перевіряє що граф *gr* не має циклів.
   * isAcyclic [[1],[2],[]] = True
   * isAcyclic gr1 = True
   * isAcyclic gr3 = False
3. Предикат *isTopolSort gr ts,*  котрий перевіряє, що список *ts* є топологічне сортування орієнтованого графа *gr*.
   * isTopolSort [[1],[2],[]] [0,2,1] = False
   * isTopolSort gr1 [0,1,2,3,4] = True
4. Функція *longWay gr a b*, котра знаходить в орієнтованому *gr* графі найдовший простий ланцюг, що з’єднує дві вершини графа *a* і *b*. Якщо в графі не існує шляху, що з’єднує вершини *a* і *b,* то функція повертає Nothing.
   * longWay gr3 4 3 = Just [4,0,1,2,3]
   * longWay gr2 0 4 = Just [0,3,2,4]
   * longWay gr1 1 0 = Nothing

*Різні*

1. Функція *merge*, котра бере два зростаючі списки цілих чисел і зливає їх в один зростаючий список (без повторних значень). Наприклад:

* merge [1,4,7] [4,5,7,9] = [1,4,5,7,9]

1. Функція *intToString n m*, що будує по цілому додатньому числу m його представлення в n-ковій системі числення (n <= 16).

* intToString 543 16 = “21f”
* intToString 543 6 = “2303”

1. Функція *stringToInt n xs*, що переводить рядок xs, котрий задає ціле додатне число в n-ковій системі числення (n <= 16), в ціле число. Якщо рядок xs не задає ціле число у відповідній системі числення, то результат Nothing

* stringToInt 10 “56a” = Nothing
* stringToInt 16 “21f” = Just 543

1. Функція *genExpr a b*, що генерує всі можливі розставлення (n-1) знаків арифметичних операцій в заданому n-значному числі *a* так, щоб результат отриманого виразу було число *b*. Операції вибираються з множини {+, -, \*}. Вираз обчислюється зліва направо без урахування пріоритету операцій. Наприклад

* genExpr 5432 12 = [“5+4-3\*2”]

1. Функція *genExprBracket a b*, що генерує всі можливі розставлення (n-1) знаків арифметичних операцій і дужок, що вказують порядок виконання цих операцій, в заданому n-значному числі *a* так, щоб результат отриманого повного виразу було число *b*. Операції вибираються з множини {+, -, \*}. Наприклад:

* genExprBracket 5432 12 = ["(((5+4)-3)\*2)","((5+(4-3))\*2)"]

*2-3-дерева*

1. Предикат *isTree23 tr*, котрий перевіряє чи являється об`єкт *tr* типу Tree23 2-3-де-ревом*.* Наприклад:
   * isTree23 (Fork2 (Leaf 0) 1 (Leaf 2)) =False
   * isTre23 tr3 = True
2. Предикат *eqTree23 tr1 tr2*, котрий перевіряє чи являються два 2-3-дерева *tr1* і *tr2* еквівалентними. Наприклад:
   * eqTree23 tr1 tr2 = True
3. Предикат *elemTree23 tr v,* котрий перевіряє чи містить 2-3-дерево *tr* значення *v.* Наприклад:
   * elemTree23 tr3 12 = True
   * elemTree23 tr3 13 = False
4. Функція *insTree23 tr v,* що вставляє в 2-3-дерево *tr* нове значення *v*. Можна визначити допоміжні функції *isTerminal*, *insTerm* та *insFork*. Наприклад:
   * insTree23 tr4 10 = tr5

## Типи функцій

*aiToInteger*  :: AbstractInteger -> Integer

*plusAbs*  :: AbstractInteger -> AbstractInteger -> AbstractInteger

*timesAbs*  :: AbstractInteger -> AbstractInteger -> AbstractInteger

*gamiltonWay* :: Graph -> Maybe [Int]

*isAcyclic*  :: Graph -> Bool

*isTopolSort* :: Graph -> [Int] -> Bool

*longWay* :: Graph -> Int -> Int -> Maybe [Int]

*merge* :: [Int] -> [Int] -> [Int]

*intToString* :: Int -> Int -> String

*stringToInt* :: Int -> String -> Maybe Int

*genExpr* :: Int -> Int -> [String]

*genExprBracket* :: Int -> Int -> [String]

*isTree23* :: (Ord a) => Tree23 a -> Bool

*eqTree23*  :: (Ord a) => Tree23 a -> Tree23 a -> Bool

*elemTree23*  :: (Ord a) => Tree23 a -> a -> Bool

*insTree23* :: (Ord a) => Tree23 a -> a -> Tree23 a