Tipuri de date abstracte (TDA)

- daca vrem sa scriem un cod corect, trebuie sa stim si modul in care sunt implementate structurile de date folosite
- List (ArrayList, LinkedList): cum ne dam seama ca insert, get sau alte operatii sunt corecte?
- inainte de a implementa o structura de date, vom defini cateva axiome ce trebuie sa fie indeplinite de oricare implementarea structurii respective de date
- ce face insert si ce face get, orice implementare de get si insert va face ceea ce ne dorim/ se va comporta corect

Ce contine un TDA ?

1. **CONSTRUCTORI** - arata din ce este format ADT-ul nostru
   Ex: Lista cu elemente de tip E => avem nevoie de:
   a) o metoda de a avea o lista goala
      empty : → List
   b) o metoda de a crea o lista prin adaugarea unui element
      cons : E x List → List
   [] = empty
   [4] = cons(4, empty)
   [1, 2, 3, 4] = cons(1, cons(2, cons(3, cons(4, empty)))) = 1 : 2 : 3 : 4 : []
   Notatie: cons(a, b) <=> a : b
2. **OPERATORI** - operatii/functionalitati realizate pe baza TDA-ului
   Ex: pentru List: reverse, concat, isempty, head, tail
   head([1, 2, 3, 4]) = 1
   tail([1, 2, 3, 4]) = [2, 3, 4]
3. **AXIOME** - cum se realizeaza operatiile → set de reguli care descriu comportamentul unui operator, fara a il implementa

In multe cazuri, veti vedea la PP ca implementarea unui operator se face aproape identic cu axioma, in mult cazuri, in limbaje functionale precum Haskell si Scala.

Axiomele si proprietatile operatorilor (propozitii logice derivate din axiome) sunt identice dpdv sintactic (ambele sunt propozitii logice) si le putem folosi in alte axiome.

Constructori

- TDA-urile au 3 tipuri de constructorii
- Presupune ca am definit un TDA numit T cu elemente de tip E:
  1. **CONSTRUCTORI NULARI** → nu primesc date de intrare/parametri
     C_n : → T
     Ex: empty
  2. **CONSTRUCTORI EXTERNI** → primesc date din exterior care nu sunt de tipul TDA-ului definit
     C_e : E → T
     C_e : E x E x E → T
     Ex: Leaf : E → Tree

3. **CONSTRUCTORI INTERNI** → primesc cel putin un paramteru de tipul TDA-ului construit
   C_i : ... x T x ... → T
   Ex: cons : E x List → List

Ce este o axioma?

- o propozitie logica adevarata, care descrie corect ce vrem sa faca operatorul
- axiomele trebuie sa fie SUFICIENTE, adica sa ia in considerare toate formele pe care le poate lua un TDA (avem nevoie de axiome pentru situatia in care lista este empty, dar si cand este cons(e, l))
  isempty(empty) = True
  isempty(cons(e, l)) = False
- ideal e sa folosim constructorii de baza ai unui ADT, nu alti operatori cand definim un set de axiome, dar nu e mereu necesar
  reverse(reverse(list)) = list
- Este o axioma inutila deoarece nu ofera informatii despre functionalitatea operatorului revers, insa poate fi folosita ca proprietate
- pe baza axiomelor putem sa obtinem proprietati ale opratorilor.
  reverse(reverse(list)) = list
- pentru a folosi o proprietate, ea trebuie demonstrata

Ex: axiome pentru operatorul ce exprima lungimea unei liste
length : List → Integer
length(empty) = 0
length(cons(x, xs)) = 1 + length (xs)

Exercitii

1. Definiți axiome pentru următorii operatori pe tipul `List`:

- `append` (concatenarea unei liste la o alta; `append(cons(8, cons(12, empty)), cons(3, cons(6, empty))) = cons(8, cons(12, cons(3, cons(6, empty))))`)
- `reverse` (inversează elementele dintr-o listă)

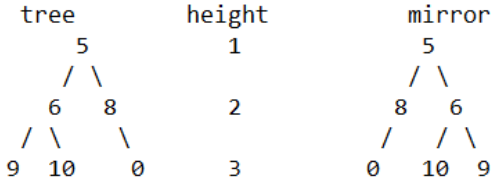
Axiome pentru append:
Append primeste 2 liste.
Vom defini axiomele acestui operator in functie de toti constructorii uneia dintre acestea.
append(empty, l) = l
append(cons(e, l_1), l_2) = cons(e, append(l_1, l_2))

Axiome pentru reverse:
reverse empty = empty ⇔ reverse [] = []
reverse cons(e, l) = $reverse_2$(empty, cons(e,l)) ⇔ reverse e:l = $reverse_2$ ([ ], e:l)
Unde definim axiome pentru operatorul auxiliar $reverse_2$:
$reverse_2$ (l_2, cons(e, l_1)) = $reverse_2$ (cons(e, l_2), l_1)
$reverse_2$ (l_2, empty) = l_2
In lista l_2 se construieste inversul, pana se consuma lista din dreapta:
l_2 lista initiala
[] 1:2:3:4:[]

1:[] 2:3:4:[]
2:1:[] 3:4:[]
3:2:1:[] 4:[]
4:3:2:1:[] []

2. Definiți axiome pentru următorii operatori pe tipul `BTree`:

- `mirror : BTree → BTree` (arborele oglindit pe verticală, i.e. pentru orice nod, copilul stâng devine copilul drept și vice-versa)
- `flatten : BTree → List` (lista cu toate elementele din arbore; observați că există mai multe ordini posibile)



Intai trebuie sa definim cine este TDA-ul Btree:

emptyTree : → BTree [nular]
Node : Integer x BTree x BTree → BTree [intern]

Node(5,
  Node(6, Node(9, emptyTree, emptyTree), Node(10, emptyTree, emptyTree)),
  Node(8, emptyTree, Node(0, emptyTree, emptyTree))
)
Apoi definim axiome pentru operatori pe baza constructorilor acestui TDA.

size : BTree → Integer
size(emptyTree) = 0
size(Node(e, s, d)) = 1 + size(s) + size(d)
Observatie: regulile/axiomele cele mai generale se pun mai jos, sus se pun cele mai particulare

height : BTree -> Integer
height emptyTree = 0
height Node(e, s, d) = max(height(s), height(d)) + 1
Observatie: Orice operator introdus trebuie sa fie definit, DAR consideram ca functiile matematice deja sunt definite.

mirror : BTree → BTree
mirror(emptyTree) = emptyTree
mirror(Node(e, s, d)) = Node(e, mirror(d), mirror(s))

flatten : BTree → List
flatten(emptyTree) = [] ← am folosit scrierea prescurtata pentru cons si empty
flatten(Node(e, s, d)) = e:[] ++ flatten(s) ++ flatten(d)
← aici ar mai fi nevoie sa aratam ca operatorul de concatenare este asociativ
(e:[] ++ flatten(s)) ++ flatten(d) = e:[] ++ (flatten(s) ++ flatten(d)) [facut mai jos, la ex. 3]

Vom introduce un operator nou: ++ / concat
Notatie: concat (l_1, l_2) = l_1 ++ l_2

concat : List x List → List ← vom nota prescurtat concat cu ++
concat(empty, l) = l
concat((e:l_1), l_2) = e : concat(l_1, l_2)
Ex: concat (1:2:3:[], 4:5:[]) = 1:concat(2:3:[], 4:5:[])) =
= 1:2:concat(3:[], 4:5:[])) = 1:2 :3:concat([], 4:5:[])) = 1:2:3:4:5:[]

3. Definiți axiome pentru următorii operatori pe tipul `Map`:

- `update : Map × K × V → Map` (un Map cu o nouă asociere cheie:element)
- `delete : Map × K → Map` (șterge cheia și valoarea asociată)

Vom defini TDA-ul Map cu valori de tip V si chei de tip E:
{
// chei : valori
x : 45
y : 99
z : 734568
}
<=> entry(x, 45, entry(y, 99, entry(z, ..., noEntry)))

Pas 1: definim constructorii
noEntry : → Map [nular]
entry : E x V x Map → Map [intern]

Pas 2: definim axiome pentru operatori
get : E x Map \ {noEntry} → V
get(k, entry(k, v, m)) = v
get(k, entry(c, v, m)) = get(k, m) (c != k)

exists : E x Map → Bool
exists(k, noEntry) = False
exists(k, entry(k, v, m)) = True
exists(k, entry(c, v, m)) = exists(k, m) (c != k)

update : E x V x Map → Map
update(k, w, noEntry) = entry(k, w, noEntry)
update(k, w, entry(K, v, m)) = entry(k, w, m)
update(k, w, entry(c, v, m)) = entry(c, v, update(k, w, m))

ex: update(1, ‘a’, entry(2, ‘b’, entry(3, ‘d’, entry(1, ‘d’, noEntry)))) =
= entry(2, ‘b’, update(1, ‘a’, entry(3, ‘d’, entry(1, ‘d’, noEntry)))) =
= entry(2, ‘b’, entry(3, ‘d’, update(1, ‘a’, entry(1, ‘d’, noEntry)))) =
= entry(2, ‘b’, entry(3, ‘d’, entry(1, ‘a’, noEntry)))

delete : E x Map → Map
delete(k, noEntry) = noEntry
delete(k, entry(k, v, m)) = m
delete(k, entry(c, v, m)) = entry(c, v, delete(k, m))

Inductia structurala

- folosita pentru a demonstra proprietati ale TDA-urilor ce deriva din axiome
- **CAZ DE BAZA**
  → aratam ca proprietatea tine pentru constructorii NULARI si EXTERNI
- **IPOTEZA DE INDUCTIE**
  → ce avem de demonstrat
- **PAS DE INDUCTIE**
  → aratam ca relatia din ipoteza de inductie ramane adevarata si cand folosim oricare din constructorii sai INTERNI

Exercitii:

4. Demonstrați următoarele propoziții, folosind inducție structurală:

a) $\forall$ t $\in$ BTree => size(t) = size(mirror(t))

CB: aplicat doar pentru constructorii nulari (si externi)

size(emptyTree) = 0

size(mirror(emptyTree)) = size(emptyTree) = 0

=> OK

II: Pp. ca size(t) = size(mirror(t)) este adevarata pentru orice lista

PI: Demontram ca II tine pentru orice constructor intern. Adica vrem sa demonstram:

size (Node(e, s, d)) = size(mirror(Node (e, s, d)))

size(Node (e, s, d)) = 1 + size(s) + size(d)

size(mirror(Node (e, s, d))) = size(Node(e, mirror(d), mirror(s))) =

= 1 + size(mirror (s)) + size(mirror(d)) = [conform II]

= 1 + size(s) + size(d)

=> Inductie OK

b) $\forall$ t $\in$ BTree => size(t) = length(flatten(t))

CB:

size (emptyTree) = 0

length(flatten(emptyTree)) = length(empty) = 0

II:

size(t) = length(flatten(t))

PI: Aratam ca: size(Node(e, s, d)) = length(flatten(Node(e, s, d)))

size (Node(e, s, d)) = 1 + size(s) + size(d)

length(flatten(Node(e, s, d))) =

= length(append(append(cons(e, empty), flatten(s))), flatten(d))

= ... = [vezi mai jos] = 1 + size(s) + size(d)

Aici avem de aratat o proprietate in plus, adica

length(append(a, b)) = length(a) + length(b) $\forall$ a, b $\in$ List

CB:

length(append(empty, l)) = length(l)

length(l) + length(empty) = length(l) + 0 = length(l)

II:

length(append(a, b)) = length(a) + length(b)

PI:

length(append(cons(x, l_1), l_2)) = length(cons(x, append(l_1, l_2))) =

= 1 + length(append(l_1, l_2)) = 1 + length(a) + length(b)
length(cons (x, l_1)) + length(l_2) = 1 + length (l1) + length(l2)
=> Inductie OK

Revenim la ce aveam inainte:
length(flatten(Node (e, s, d))) =
= length(append(append(cons(e, empty), flatten(s))), flatten(d))
= length(append(cons(e, empty), flatten(s))) + length(flatten(d)) =
= length(cons (e, empty)) + length(flatten(s)) + length(flatten(d)) =
= 1 + length(empty) + length(flatten (s)) + length(flatten (d)) =
= 1 + 0 + length(flatten(s)) + length(flatten (d)) =
= 1 + length(flatten(s)) + length(flatten(d)) =
= 1 + size(s) + size(d)

=> Inductie OK

c) $\forall$ l $\in$ List => append(l, empty) = l

CB:

l = empty
append(empty, empty) = empty

II:

append (l, empty) = l

PI:

l = cons(x, xs)
Demonstram ca
append(cons(x, xs), empty) = cons(x, xs)
append(cons(x, xs), empty) = cons(x, append(xs, empty)) = cons(x, xs)

=> Inductie OK

d) $\forall$ l1, l2, l3 $\in$ List => append(l_1, append(l_2, l_3)) = append(append(l_1, l_2), l_3)

adica: l_1 ++ (l_2 ++ l_3) = (l_1 ++ l_2) ++ l_3

CB:

l1 = empty
empty ++ (l_2 ++ l_3) = l_2 ++ l_3
(empty ++ l_2) ++ l_3 = l_2 ++ l_3

II:

l_1 ++ (l_2 ++ l_3) = (l_1 ++ l_2) ++ l_3

PI: demontram ca:

cons (x, l_1) ++ (l_2 ++ l_3) = (cons (x, l_1) ++ l_2) ++ l_3
cons(x, l_1) ++ (l_2 ++ l_3) = cons(x, l_1 ++ (l_2 ++ l_3))
= (cons (x, l_1) ++ l_2) ++ l_3 = cons(x, l_1 ++ l_2) ++ l_3 =
= cons(x, (l_1 ++ l_2) ++ l_3) = cons(x, l_1 ++ (l_2 ++ l_3))

=> Inductie OK

e) $\forall$ l_1, l_2 $\in$ List => length(append(l_1, l_2)) = length(append(l_2, l_1))

CB:

length(append(empty, l_2)) = length(l_2)
length(append(l_2, empty)) = length(l_2) deja anterior demonstrat

II:

length(append(l_1, l_2)) = length(append(l_2, l_1))

PI: demonstram ca:

length(append(cons(x, l_1), l_2)) = length(append(l_2, cons (x, l_1)))

length(append(cons (x, l_1), l_2)) =

= length(cons(x, append(l_1, l_2))) =

= 1 + length(append(l_1, l_2)) =

= 1 + length(l_1) + length(l_2)

length (append (l_2, cons(x, l_1))) =

= length(l_2) + length(cons(x, l_1))) =

= length(l_2) + 1 + length(l_1) =

= 1 + length(l_1) + length(l_2)

=> Inductie OK

f) ∀ l1, l2 ∈ List -------> reverse (append (l1, l2)) = append(reverse(l2), reverse (l1))

CB:

reverse(append(empty, l_2)) = reverse (l_2)

append(reverse(l_2), reverse(empty)) =

= append(reverse(l_2), empty) = reverse(l_2)

II:

reverse(append(l_1, l_2)) = append(reverse(l_2), reverse(l_1))

PI: Demonstram ca:

reverse(append(cons(x, l_1), l_2)) =

= append(reverse(l_2), reverse(cons(x, l_1)))

reverse(append(cons(x, l_1), l_2)) =

= reverse(cons(x, append(l_1, l_2)))

= append(reverse (append(l_1, l_2)), cons(x, empty)) =

= append(append (reverse(l_2), reverse(l_1)), cons(x, empty)) =

= (reverse(l_2) ++ reverse(l_1)) ++ cons(x, empty)

append(reverse(l_2), reverse(cons(x, l1))) =

= reverse(l_2) ++ reverse(cons(x, l_1)) =

= reverse(l_2) ++ (reverse(l_1) ++ cons(x, empty)) =

= (reverse(l2) ++ reverse(l1)) ++ cons(x, empty)

=> Inductie OK