Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Uge(r)seddel 7 – individuel opgave

Torben Mogensen

Deadline 27. oktober

I denne periode skal I arbejde individuelt. Formålet er at arbejde videre med lister, mængder, afbildninger og rekursive datastrukturer, altså HR kapitel 4–6.

Opgaverne i denne uge er delt i øve- og afleveringsopgaver.

Øveopgaverne er:

Ø7.1. HR opg. 4.9, 4.11, 6.4.

Ø7.2. HR afsnit 6.4 beskriver binære træer `BinTree`, der svarer til typen `nodeTree`, der blev præsenteret ved forelæsningen 6. oktober. Underafsnittet "Search trees" viser, hvordan man kan bruge disse træer til at definere *søgetræer*: Træer, hvor alle værdier til venstre for rodknuden er mindre end værdien i rodknuden, og hvor alle værdier til højre for rodknuden er større end værdien i rodknuden. Afsnittet definerer funktioner til indsættelse af værdier i et søgetræ (`add`) samt til søgning i et søgetræ (`contains`). Bemærk, at hver værdi kun kan forekomme en gang i et søgetræ, så det implementerer en mængde af værdier.

Definer følgende mængdeoperationer:

i En funktion `minElement : BinTree<'a> -> 'a`, som returnerer det mindste element i mængden. Hvis der ikke er nogen værdier i træet, skal en passende fejlmeddelelse gives.

ii En funktion `maxElement : BinTree<'a> -> 'a`, som returnerer det største element i mængden. Hvis der ikke er nogen værdier i træet, skal en passende fejlmeddelelse gives.

iii En funktion `removeMin : BinTree<'a> -> BinTree<'a>`, der fjerner det mindste element fra en mængde. Hvis mængden er tom, returneres den uændret.

iv En funktion `removeMax : BinTree<'a> -> BinTree<'a>`, der fjerner det største element fra en mængde. Hvis mængden er tom, returneres den uændret.

v En funktion `remove : 'a -> BinTree<'a> -> BinTree<'a>`, der fjerner et element fra en mængde, svarende til funktionen `Set.remove`.

Ø7.3. Et binært træ kaldes *balanceret*, hvis de to undertræer til enhver knude i træet er *næsten* lige store, dvs. at størrelserne på undertræerne højest må have en forskel på 1.

i Diskuter om træerne i Figur 6.11 i HR er balancerede.

ii Tegn et søgetræ svarende til samme mængde som `t4` i Figur 6.11, men som er balanceret.

iii Lav en funktion `isBalanced : BinTree<'a> -> bool`, der afgør om et træ er balanceret. **Vink:** Lav en hjælpefunktion, der returnerer størrelsen s af et træ som en værdi `Some` s, hvis træet er balanceret, og ellers `None`.

Afleveringsopgaven er:

Vi ser blandt andet på træer af typen `ListTree<'a>`, som er defineret i HR afsnit 6.6.

A7.1. Lav en funktion `size : ListTree<'a> -> int`, der finder størrelsen af et træ målt som antal knuder. Brug så vidt muligt kun eksplicit rekursion over træet, brug funktioner fra HR Figur 5.1 til at arbejde med lister, hvor det er muligt.

A7.2 Lav en funktion `span : int list -> int`, der givet en liste af tal finder forskellen mellem det største og det mindste tal i listen. Hvis listen er tom, er resultatet 0. Brug ikke eksplicit rekursion.

A7.3 Vi definerer at et træ af typen `ListTree<'a>` er *balanceret*, hvis størrelsen af enhver knudes undertræer højest afviger med 1 fra hinanden. Træet i HR Figur 6.14 er f.eks. ikke balanceret, da midterste undertræ til roden har størrelse 1 mens højre undertræ har størrelse 3. Men hvis der under knuden i midten tilføjes præcis en knude, vil alle undertræer til roden have størrelse 2 eller 3, og da undertræerne åbentlyst er balancerede, bliver træet dermed balanceret.

Skriv en funktion `isBalanced: ListTree<'a> -> bool`, der afgør om et træ er balanceret. **Vink:** Lav en hjælpefunktion, der returnerer størrelsen s af et træ som en værdi `Some` s, hvis træet er balanceret, og ellers `None`.

Undgå så vidt muligt at bruge eksplicit rekursion undtagen på træstrukturen. Lister bør så vidt muligt behandles med funktioner fra HR Figur 5.1.

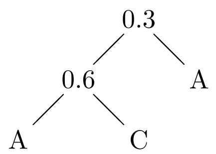
Afleveringsopgaven skal afleveres som både LaTeX, den genererede PDF, samt en fsx fil med løsningen for hver delopgave, navngivet efter opgaven (f.eks. `A7-1.fsx`), som kan oversættes med fsharpc, og hvis resultat kan køres med mono. Det hele samles i en zip-fil med sædvanlig navnekonvention (se tidligere ugesedler).

God fornøjelse

Ugens nød 4

Vi vil i udvalgte uger stille særligt udfordrende og sjove opgaver, som interesserede kan løse. Det er helt frivilligt at lave disse opgaver, som vi kalder "Ugens nød", men der vil blive givet en mindre præmie til den bedste løsning, der afleveres i Absalon.

Denne uges opgave omhandler *sandsynlighedstræer*. Et sandsynlighedstræ er et træ med sandsynligheder (tal mellem 0.0 og 1.0) i knuderne og værdier (af vilkårlig type) i bladene. Betydningen er, at en knude med sandsynlighedsværdien p repræsenterer et tilfældigt valg: Med sandsynlighed p vælges det venstre undertræ, og med sandsynlighed $1 - p$ vælges det højre undertræ. Et blad angiver den valgte værdi. For eksempel vil træet



Betyde, at man med sandsynlighed 0.3 vælger venstre gren, og derefter med sandsynlighed $1.0 - 0.6 = 0.4$ vælger den højre gren, og dermed får værdien C. Sandsynligheden for at få C er altså $0.3 \cdot 0.4 = 0.12$. Læg mærke til, at samme værdi kan optræde flere steder i træet. I det viste træ kan A f.eks. nås ved at tage venstre gren to gange (sandsynlighed $0.3 \cdot 0.6$) eller ved at tage højre gren fra starten (sandsynlighed $1.0 - 0.3$), så den samlede sandsynlighed for at vælge A er $0.3 \cdot 0.6 + 1.0 - 0.3 = 0.88$.

Vi repræsenterer sandsynlighedstræer med typen:

```
type 'a pT = Pick of 'a | Choose of float * 'a pT * 'a pT
```

Det viste træ kan med denne struktur repræsenteres som værdien

```
let st = Choose (0.3, Choose (0.7, Pick 'A', Pick 'C'), Pick 'A)
```

Nød 4.1. Lav en funktion `checkPT : 'a pT -> bool`, der undersøger, om alle sandsynlighederne i træet er mellem 0.0 og 1.0 (ikke inklusive endepunkterne).

Nød 4.2. Lav en funktion `makeMap : 'a pT -> Map<'a,float>`, der givet et træ laver en afbildning fra værdier til sandsynlighederne for at disse værdier bliver valgt. For det viste træ skal `'A'` afbildes til 0.88 og `'C'` til 0.12. Afbildningen er udefineret for alle andre værdier.

Nød 4.3. Lav en funktion `product : 'a pT -> 'b pT -> ('a * 'b) pT`, som givet to sandsynlighedstræer for typerne `'a` og `'b` laver et sandsynlighedstræ for typen `'a * 'b`, sådan at hvis sandsynligheden for at vælge x i det første træ er p og sandsynligheden for at vælge y i det andet træ er q, så er sandsynligheden for at vælge (x, y) i det resulterende træ lig med pq.

Nød 4.4. Et sandsynlighedstræ er *normaliseret*, hvis der ikke er to ens blade. Lav en funktion `normalize : 'a pT -> 'a pT`, sådan at det resulterende træ repræsenterer den samme sandsynlighedsfordeling som argumentet (dvs. at `makeMap t = makeMap(normalize t)`), men så det resulterende træ er normaliseret. `normalize st` (hvor `st` er det herover viste træ) bør derfor give enten `Choose (0.88, Pick 'A', Pick 'C')` eller `Choose (0.12, Pick 'C', Pick 'A')`

Nød 4.5. Vi måler størrelsen af et sandsynlighedstræ som antallet af `Choose`-knuder, og bruger samme kriterium for balancerede træer som i øvelsesopgaverne. Bemærk, at træet `st` er et balanceret træ af størrelse 2.

Lav en funktion `balance : 'a pT -> 'a pT`, sådan at det resulterende træ repræsenterer den samme sandsynlighedsfordeling som argumentet (dvs. at `makeMap t = makeMap(balance t)`), men så det resulterende træ er både normaliseret og balanceret.

Der skal uploades både en LaTeX-fil, der beskriver fremgangsmåden, samt en fsx fil, der indeholder definitionerne. Navngivningen af filerne er ikke vigtig.