SI-möte #11, Programkonstruktion och Datastrukturer

Elias Castegren & Kristiina Ausmees
elca7381@student.uu.se ‖ krau6498@student.uu.se

9 februari 2012

Begrepp

i) Hur fungerar referenser i ML? Vad är skillnaden mellan en referens och en bindning?

ii) Vad menas med imperativ programmering och hur skiljer det sig från funktionell programmering?

Övningar

1.
Betrakta följande ML-kod:

```
val a = 42;
val b = ref 42;
val c = a;
val d = b;
val f1 = (fn x => x + a);
val f2 = (fn x => x + !b);
f1 2;   →
f2 2;   →
val a = 13;
b := 99;
d := 23;
f1 c;   →
f2(!d);   →
```

Vilket värde beräknas vid de markerade raderna? Vad har `a`, `b`, `c` och `d` för värden i slutet?

Fler begrepp

i) Vad är en *hashtabell*? Vad gör en hashfunktion? Vad är hashfunktionens definitions- och värdemängd?

ii) Hur definieras en hashtabells *loadfaktor*? Vad innebär *rehashing*? Vad är tidskomplexiteten för rehashing?

iii) Vad innebär *chaining* när man pratar om hashtabeller?

iv) Vad är *öppen adressering* (*open addressing*)? Hur fungerar *linjär* och *kvadratisk probing*?

v) Hur många element kan man lagra i en hashtabell med m platser om man använder chaining respektive open addressing? Vilka värden kan loadfaktorn anta med de olika teknikerna?

2.
Antag att du har en hashtabell med 10 celler som använder chaining. Tabellens hashfunktion är

$$h(k) = k \; mod \; m$$

där m är antalet celler i hashtabellen. Sätt in följande element:

$$113, 19, 155, 16, 129, 43, 99, 26, 125, 73$$

Ta sedan bort elementen

$$19, 26, 129, 73$$

Vad händer om man försöker leta upp elementet 33? När vet man att man kan sluta leta?

3.
Sätt in och ta bort samma element som i uppgift 1, fast i två hashtabeller som använder öppen adressering med linjär respektive kvadratisk probing (utan rehashing) istället för chaining. Använd samma hashfunktion som ovan. För den kvadratiska probingen kan du använda probfunktionen $f(i) = i^2$. Räkna det totala antalet prober som behövs när man sätter in och tar bort elementen från de olika tabellerna.

4.
Funktionen `killDuplicates(l)` tar bort alla dubletter ur en lista `l`. Den kan implementeras genom att man för varje element e söker igenom resten av listan och tar bort alla förekomster av e. Algoritmen har då den genomsnittliga tidskomplexiteten $\Theta(|l|^2)$.

Använd hashtabeller för att skriva en version av `killDuplicates` som genomsnittligt har linjär tidskomplexitet! Här är en påminnelse om Poly/MLs hashbibliotek:

- `HashArray.hash(i)` - En ny hashtabell med `i` celler
- `HashArray.sub(h, s)` - `SOME` värdet med nyckeln `s` i `h` om det finns, annars `NONE`
- `HashArray.update(h, s, v)` - Uppdaterar `h` så att cellen med nyckeln `s` innehåller `v`

Lycka till!