En miljö

Kenan Dizdarevic

23 Januari 2023

Inledning

Tidigare har vi arbetat med *nyckel-värde-databaser*, även kallat ”Map”. Dessa databaser används för att hitta ett specifikt värde associerat med en unik nyckel. Vi gör inga antaganden gällande nycklarna, vi kommer endast jämföra dom med vanliga operatorer.

Vi kommer att implementera databasen med två olika datastrukturer. I den första implementeringen skall vi implementera den som en lista av tupler. Databasen kommer att implementeras som ett träd i den andra implementeringen.

Till sist skall vi jämföra våra implementeringar och se hur de jämför sig med **Elixirs** egna implementering av funktionen *Map*.

Undersökning

Lista

Vi antar att vår databas kommer vara liten, då representerar vi paren av nycklar och värden som en tupel. En lista kan representeras på följande vis:

```
list = [{:a, 1}, {:b, 2}, {:c, 3}]
```

Bokstäverna representerar nycklarna och talen är alla värden.

Listan ska ha möjligheten att utföra fyra operationer:

- `new()`: returnerar en tom map
- `add(map, key, value)`: lägger till ett nytt par, eller uppdaterar en befintlig nyckel
- `lookup(map, key)`: returnerar {`key, value`} om nyckeln finns, annars returneras `nil`
- `remove(map, key)`: returnerar en map där givna nyckeln är borttagen

Detta skall implementeras rekursivt, vi behöver tänka ett steg längre innan vi börjar implementera funktionerna. Innan vi implementerade *remove(map, key)* ställde vi upp de olika fall som kan uppstå.

- **Fall 1**: Tom map
- **Fall 2**: Nyckeln finns inte i map
- **Fall 3**: Nyckeln finns i map

Koden ser ut som följande:

```
def remove([], _) do [] end
def remove([{key, value} | tail], key) do tail end
def remove([head | tail], key) do [head | remove(tail ,key)] end
```

En mycket viktig funktion som vi använder är *mönstermatchning*. I *remove/1* behandlar vi fallet där map är tom, vi returnerar endast en tom lista. Detta är vårt basfall för rekursionen. *remove/2* tar emot en lista som inte är tom. Vi sätter första elementet i listan till `{key, value}` och resten av listan till *tail*. Vi returnerar endast *tail* från detta fall I *remove/3* sker det rekursiva anropet. Vi delar återigen upp listan i *head* och *tail*. Sedan returnerar vi hela listan men vi anropar *remove* igen på *tail*.

Detta tankesätt som innefattar rekursiva anrop gäller både för *add* och *lookup* för listan. Denna algoritm är **inte** svansrekursiv, vilket inte är till vår fördel. Det spelar ingen roll om listan är sorterad eller ej, det gynnar oss inte. Vi skall nu försöka implementera en svansrekursiv algoritm, men med en annan datastruktur.

Träd

Om vår map blir större är inte implementeringen med listan den bästa lösningen. Vi vill implementera en algoritm som är svansrekursiv. Vi skall undersöka om implementeringen av en map i form av ett träd är bättre. Trädets egenskaper skall vara likadana som listans. Vi skall implementera samma funktioner men i ett träd.

Vi väljer att återigen studera funktionen *remove({:node, key, value, left, right}, key)*. Notera att istället för att ta in en lista som argument har vi en nod. Vi har ett basfall om trädet är *nil* returnerar vi *nil*. Sedan har vi två fall där vi returnerar noden till vänster om det saknas en nod till höger och tvärt om för det andra fallet. Resten av koden för *remove* ser ut som följande:

```
def remove({:node, key, _, left, right}, key) do
{key, value, rest} = leftmost(right)
{:node, key, value, left, rest}
end
```

```
def remove({:node, k, v, left, right}, key) when key < k do
{:node, k, v, remove(left, key), right}
end
def remove({:node, k, v, left, right}, key) do
{:node, k, v, left, remove(right, key)}
end
def leftmost({:node, key, value, nil, rest}) do {key, value, rest} end
def leftmost({:node, k, v, left, right}) do
{key, value, rest} = leftmost(left)
{key, value, {:node, k, v, rest, right}}
end
```

Vi har implementerat en ny funktion som heter *leftmost*. Uppgiften som den har är att traversera över trädet för att hitta den nod längst till vänster i trädet. Denna nod kommer att ta över platsen där noden som vi skall ta bort finns.

Vi använder oss av *mönstermatchning* i det första fallet i koden ovan. Där vi sätter *key, value* & *rest* till den nod som är längst till vänster från nästa högra nod.

De två fall efter genomför de rekursiva anropen. Vi jämför vår eftersökta nyckel med den nuvarande, om vår nyckel är större rör vi oss till höger i trädet. Detta möjliggör hela algoritmen, man kan se det som att vi delar upp trädet i ett nytt och börjar leta i det nya.

Prestandajämförelse

Vi skall nu mäta implementeringarnas prestanda och jämföra dem. Elixir har en egen implementering av *Map* som är skriven i C++, vi skall se hur våra implementeringer ställer sig mot den. Vi kommer att genomföra en prestandajämförelse på de tre olika funktionerna i varje implementering. Mätvärden som vi erhåller presenteras i tabellerna nedan.

Storlek	**Lista**	**Träd**	**Map**
128	0.5	0.3	0.1
256	1.1	0.4	0.1
512	2.0	0.4	0.1
1024	4.36	0.4	0.1
2048	9.2	0.5	0.1
4096	22.7	0.5	0.1

Tabell 1: Tid i µs för *add* i de olika implementeringarna

Storlek	Lista	Träd	Map
128	0.17	0.1	0.04
256	0.35	0.15	0.04
512	0.53	0.12	0.04
1024	1.1	0.12	0.04
2048	2.3	0.17	0.03
4096	6.4	0.16	0.04

Tabell 2: Tid i µs för *lookup* i de olika implementeringarna

Storlek	Lista	Träd	Map
128	0.74	0.28	0.08
256	1.1	0.28	0.09
512	2.3	0.37	0.1
1024	4.5	0.34	0.1
2048	9.1	0.50	0.1
4096	22.1	0.46	0.1

Tabell 3: Tid i µs för *remove* i de olika implementeringarna

Resultaten är tydliga. Listan lämpar sig inte för stora datamängder. Trädet är en avsevärd förbättring från listan men *Map* är den självklara vinnaren. Den är implementerad som ett träd av hashtabeller vilket gör alla åtkomster konstanta.

Våra implementeringar visar även skillnaden mellan något som är *svansrekursivt* och inte. När vi i framtiden utvecklar algoritmer vill vi ha dem svansrekursiva. Det kräver mer eftertanke men resultatet är avsevärt bättre.

Slutsats

Sammanfattningsvis ser vi att det finns tydliga skillnader mellan något som är *svansrekursivt* och inte. Listan fungerar men det är inget bra alternativ om vi arbetar med större datamängder. Trädet är svansrekursivt och lämpar sig bättre för större datamängder.

Vi har även fått en djupare förståelse för det rekursiva tankesättet i funktionell programmering. Vi bör alltid försöka implementera svansrekursiva funktioner.