# En HP35 Miniräknare

Jonathan Nilsson Cullgert

Hösten 2023

## Introduktion

Denna uppgift gick ut på att skapa en HP35 miniräknare som kan beräkna matematik med hjälp av omvänd polsk notation. Anledningen till att just detta exempel används i denna uppgift är för att det är ett perfekt exempel att tillämpa stacken. Stacken använder en sist in, först ut princip och då omvänd polsk notation är som att lägga allt på en hög, en stack, och sedan göra beräkningar på de värden som ligger högst upp på stacken.

## Stacken

Stacken är ett av flertalet alternativ att lagra data på, där data som placeras på toppen av stacken hämtas först. Detta är ett väldigt effektivt sätt att lagra data på om man vill hämta och använda data ofta. Till exempel om man ska implementera en funkton i ett program som kan gå fram- och tillbaka genom olika tillstånd, som att använda ctrl + z och ctrl + y.

I denna uppgift är det huvudsakligen två metoder som används med stacken, dock finns det fler. Push metoden används som pushar, lägger på, ett element på stacken. Pop metoden används som poppar, hämtar, ett element från stacken. Utöver det finns det andra metoder som exempelvis top, vilken kollar på det översta elementet i stacken utan att hämta det.

När man använder stacken kan man dock stöta på problem om stacken har en bestämd storlek. Om man försöker lägga in mer element i stacken än vad den tillåter kommer man få överflöde på stacken och förlora data. I denna uppgift finns detta problem i den statiska stacken men inte i den dynamiska stacken. Man kan även stöta på problem åt andra hållet, om man försöker få ut data ur stacken när den är tom kan man råka hämta data som ligger närliggande i minnet av stacken men som **INTE** ligger på stacken. Det blir ett underflöde i stacken.

## Statisk

Den första delen av denna uppgift var att implementera en statisk stack som som kunde beräkna matematiken från ett bestämt antal element genom att pusha och poppa på stacken. Jag skapade en abstrakt klass för stacken som innehåller push och pop metoderna samt en pekare på stacken. Sedan skapade jag en statisk klass som ärver från stack klassen. I den statiska klassen kommer metoderna implementeras sådan att de faktiskt fungerar och inte bara existerar.

I den statiska stacken finns det som sagt risken att det blir underflöde eller överflöde i stacken, det vill säga pekaren hamnar utanför stacken på någon av ändarna. För att lösa detta problem sattes två gränser.

```
if(pointer >= stack.length){//ger en exception om man får stack overflow
        throw new IndexOutOfBoundsException("stack overflow");
    }
```

Denna kod finns i push metoden och ger en exception när stacken har blivit full och på så sätt kommer man inte råka pusha på en full stack. En likvärdig rad kod har skrivits för när stacken är tom, då går det inte att poppa mer om stacken är tom.

```
if(pointer == 0){
        throw new IndexOutOfBoundsException("No more elements in stack");
    }
```

När stacken är full kommer pekaren att peka på det elementet som ligger ett index efter det sista elementet. Anledningen till att pekaren kommer peka på just det elementet är för att när push metoden körs indexeras pekaren framåt upp ett steg efter man har lagt till ett värde på stacken. Det vill säga att om man pushar det sista elementet på stacken kommer pekaren indexeras upp ett steg och peka på nästa element trots att det inte går att lägga till ett nytt element.

Om Stacken är tom kommer pekaren ha värdet 0 och det beror på att stacken i denna uppgift är byggd på arrayer i java och i java är arrayer noll indexerade, vilket ger värdet. Anledningen till att pekaren mot förmodan inte blir negativ är just för att kodsträngen i pop metoden som hindrar den att poppa när stacken är tom också tvingar pekaren att stanna på 0.

## Dynamisk

Nästa del av uppgiften var att göra en ny stack som tillskillnad från den statiska kan ändra storlek beroende på hur många element som finns i stacken. Denna typ av stack skulle inte problem med överflöde på stacken då den bara blir större ifall fler element behövs läggas till på stacken.

Jag skapade den dynamiska stacken genom att dubbla längden av den tidigare storleken. När man skapar den dynamiska stacken märker man väldigt snabbt att den kommer vara långsammare än den statiska ifall man måste lägga till fler element än vad som ursprungligen fick plats. Detta är då man behöver kopiera över element två gånger varje gång stacken utökas. Först skapas en ny temporär stack med den nya önskade storleken, alla element kopieras över i en for-loop, den gamla stacken får den nya storleken och sedan i en till for-loop alla element. Alltså måste alla element gås igenom $2n$ gånger. Den temporära stacken frigörs sedan efter all kopiering så att inte onödigt mycket minne tas upp.

Den dynamiska stacken ska inte bara kunna bli större om det behövs utan den ska även kunna bli mindre ifall det finns mycket utrymme på stacken som inte används. Jag gjorde att stacken minskas till 1/4 storlek ifall det finns 1/4 så många element som totalt finns plats i stacken.

```
if(pointer == stack.length/4){
            int[] tempstack = new int[stack.length/4];
            :
            :
```

Det problemet som kan uppstå när stacken ökar och minskar på detta sätt är att man kan behöva kopiera onödigt mycket. Exempelvis om original storleken är 16, pekaren går ner till 4, man pushar ett nytt element och storleken blir då 8. Ett sätt att lösa detta är genom att exempelvis halvera storleken av stacken vid 1/4 längd. Anledningen till att jag inte valde att göra det är dels för att inte tänkte på det men också för att ifall man har poppat till 1/4 av ursprungs storleken kommer man förmodligen att fortsätta poppa. Att ändra på dessa variabler är inte svårt. I detta fall är det inte lika tidskrävande som om man exempelvis halverade och dubblerade storleken, som om man går från 16 till 4 till, 8 istället för 16 till 8 till 16 och så vidare.

När man minskar på stacken görs även det med en temporär stack som sedan frigörs, och detta görs med 2 stycken for-loopar, alltså $2n$ i tid. Den dynamiska stacken kommer inte ha överflöde, men den kan fortfarande få ett underflöde. Alltså måste samma mekanism som fanns i den statiska stacken som förhindrar detta finnas med även här.

## Benchmark

Det viktiga med denna uppgift är att se skillnaden mellan den dynamiska och statiska stacken. Vad kostar dem olika alternativen i tid och minne. Det man förlorar i tid vinner man i minne och tvärtom. Den dynamiska stacken har bättre minneshantering än den statiska, då den alltid kommer den storleken som behövs vid just en specifik tidspunkt, och den kommer aldrig att bli för liten (om inte minnet helt tar slut).

Då vi vet att den dynamiska vinner över den statiska i minne vill man självklart ta reda på hur mycket man förlorar i tid jämfört med den statiska. Detta görs genom att först pusha 1000 element på båda stackarna och sedan poppa stackarna tills det att de blivit tomma. Detta pushande och poppande görs 10000 gånger för att få stabila siffror. Det minsta värdet av hundra av dessa körningar tas och medelvärdet av dessa hämtas.

| Antal element | Förhållande |
|---|---|
| 1000 | 2.7 |

Table 1: Tabell som visar förhållandet mellan statisk och dynamisk stack

Som vi kan se i 1 är den statiska stacken 2.7 gånger snabbare än den dynamiska när man ska poppa och pusha 1000 element. Alltså vinner man en del tid på att inte behöva utöka och minska storleken av stacken allt eftersom. Denna skillnad på tid kan vara kritisk beroende på vad man använder denna stack till. Om man bygger något som behöver vara väldigt snabbt, exempelvis bromsen på en bil så kan denna tidsskillnad vara liv eller död. Det är därför väldigt svårt att säga om det man vinner i minne och möjligheten att utöka stacken är mindre värt än det man förlorar i tid. Generellt är det bättre med snabbare saker då tid är begränsat men det är väldigt enkelt att skaffa mer minne.

## Slutsats

Som slutsats kan man säga att den dynamiska och statiska stacken har olika applicering beroende på vad det är man vill göra. Vill man vara snabb bör man använda den statiska stacken och vill man ha möjlighet att ändra på storleken av stacken bör man använda en dynamisk. Det jag lärde mig av denna uppgift var den logik som används för att skapa en stack så att inte man råkar få fel, eller bli av med data. Det blev också uppenbart hur mycket tid man förlorar av att faktiskt kopiera om arrayer flertalet gånger, något man inte hade fått bra förståelse om, om man hade programmerat i exempelvis python. Striden mellan tid/minne och att man ständigt slåss om vad som är viktigast i just sitt program var viktigt att lära sig om.