HP35 kalkylator och stackar

Viktor Björkén

September 2022

1 Introduktion

I denna rapport kommer vi att arbete runt hur en HP35 kalkylator fungerar. Detta kan beskrivas som en ”reverse polish notation” typ, av räknar. Vi kommer att med hjälp av kodsnippar vi får i uppgiften bygga vidare på räknaren. De kommer dessutom behöva anväda oss 2 egengjorda stackar. Dessa stackar kommer att vara en statsik samt 1 dynamisk. Vi kommer sedan att jämförahur dessa stackar står mot varandra genom våran HP35 räknare.

2 Statisk stack

Första uppgift är att stoppa in den kod vi har fått för att sedan addera funktioner som får vårat program att hitta vilken typ vi får, dvs ett värde eller en typ där vi använder oss av enums.

```
        public enum ItemType {
        ADD,
        SUB,
        MUL,
        DIV,
        VALUE,
        MOD
    }
```

Dessa använder vi för att göra de operationer vi vill med de beräkningar som önskas. För att de ska fungerar måste vi dessutom använda ”switch cases” där vi bestämmer hur dessa ska se ut.

Våran switch case är uppbyggd genom följande step metod, vi bygger vidare dessa steg för alla operationer, så SUB, MUL, DIV osv.

```
    public void step() {
        Item nxt = expr[ip++];
        switch(nxt.type()) {
            case ADD:
            {
```

```
                int y = StaticStack.pop();
                int x = StaticStack.pop();
                StaticStack.push(x + y);
                break;


            }
        }
    }
```

Det vi sedan gör är att vi bygger våran statiska stack. Denna stack ska ha tillgång till att pusha och popa från stacken, vi gör detta med hjälp av dessa rader kod.

```
public void push(int item) {
        if (isFull()) {
            throw new RuntimeException("Stack overflow");
        }
        // insert element on top of stack
        System.out.println("Inserting " + item);
        array[++top] = item;
    }

       public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        // pop element from top of stack
        return array[top--];
    }
```

När vi gör en becnhmark på denna stack får jag att en operation med 14 värden varav 6 är operationer. Får jag ett medelvärde på att varje steg tog 6 microsekunder var.

```
Inserting 10
 time: 5700ns
Inserting 4
 time: 5700ns
.
.
.
```

Detta ger oss ett tidsvärde det tar för varje operation att genomföras, kör vi denna kod ett antal gånger kan vi anta ett medelvärde för den tiden det tar för en operation.

2.1 Slutsats

I vårat fall peckar våran peckare alltid på den senaste värdet. För att motverka stack overflows samt popande från en tom stack har jag i min stack använt mig utav 2 funktioner som kollar så att detta inte inträffar. Om så är fallet kastar vi ett ”runtime exception”

```
 public boolean isEmpty() {
    return top == - 1;
}
public boolean isFull() {
    return top == capacity - 1; //capacity = size
}
```

Vi använder dessa funktioner i våra push och pop metoder med hjälp av if satser.

3 Dynamsik stack

I våran dynamiska stack kommer våra pop och push funktioner se ut på ett lite annorlunda sätt, detta då en dynamisk stack fungerar annorlunda.

I en dynamisk stack kommer stacken att växa eller minska beroende på hur många element som finns i stacken.

Därför kommer våra funktioner se ut på följande.

```
int pop(){
    if(pointer < (stack.length/2 - 1)) {
        int[] newStack = new int[(stack.length / 2)];
        for(int i=0;i<stack.length/2;i++) {
            newStack[i] = stack[i];
        }
    stack = newStack;
    }
    return stack[pointer--];


}

void push(int i){
    if(pointer == stack.length) {
        int[] newStack = new int[stack.length * 2];
        for(i=0;i<stack.length;i++) {
            newStack[i] = stack[i];
        }
    stack = newStack;
    }
```

```
        stack[++pointer] = i;

    }
```

När vi kör våran benchmark på den dynamiska stacken istället för den statiska inser vi att vi får högre tidsvärden.

```
Inserting 10
 time: 7200ns
Inserting 4
 time: 7200ns
.
.
.
```

Vi kan däför dra slutsatsen att den dynamiska stacken är ”långsammare” än den statiska, detta kan bero på att den dynamiska är med ”advancerad” där den man bland annat kan ”växa”per antalet pushar och pops man använder till skilland från den statiska som växer konstant.

3.1 Slutsats

Det vi kan se när det gäller tidskomplexiteten är den varier beroende på vart i stacken vi är. Eftersom stacken ökar dynamiskt så kommer stacken behöva kopieras över för att fortsätta öka. Detta ändrar tidskomplexiteten, Därför får vi tillslut en helhets komplexitet på O(n) för den dynamiska stacken.

4 Sista siffra i personnummer

I denna del ska vi med hjälp av vårat personnummer göra en beräkning där vi räknar ut våran sista siffra med hjälp av de föregående.

För att göra denna beräknning kommer vi behöva en ny opertion. Denna är modolu. Vi implementerar denna på samma sätt som tidigare operationer till exempel addition och division.

```
case MOD: {
    int y = stack.pop();
    int x = stack.pop();
    stack.push(x -(x/y)*y);
    break;
}
```

För att göra denna beräkning kommer vi att ta vårat personnummer där vi tar varanat tal multiplicerat med 2 och sedan summan av detta modulo 10. Sedan 10 minus det tal vi får.