Uppgift 2

Simon Moradbakti

HT 2024

1 Introduktion

I den här uppgiften har skribenten implementerat en räknare som amvänder sig av *"Reverse Polish"* notationen. En enkel förklaring för vad det innebär är att alla matematiska uttrycks operand är sist i uttrycket. Ett sådant exempel är (2+2) som skulle uttryckas som (2 2 +), i *Reverse Polish* notation. Många gamla räknare fungerade på det sättet, ett sådant exempel är HP35 från 70-talet. För att det här systemet ska fungera måste nånting som kallas *"Stack"* finnas på plats. En *Stack* är en typ av datastruktur och kan lite förenklat jämföras med en hög. En *Stack* har 2 typer av operationer, *"Push"* och *"Pop"*. *"Push"* operationen, precis som det låter pushar eller lägger upp nånting, (i vårt fall oftast en operand eller siffra) på vår *Stack* eller "hög". Där förvaras den. *"Pop"* o andra sidan tar bort elementet högst upp på vår stack eller "hög" och returnerar det. En *Stack* bygger på principen "sist in, först ut". I den här rapporten kommer vi beröra både *statisk* och *dynamisk Stack*.

2 Statisk Stack

En statisk stack är en stack som inte kan alokera mer plats för element än vad den initialt har tilldelats. Lite förenklat kan det uttryckas som att en "hög" ska få plats i en låda. En oändligt stor hög får alltså inte plats utan endast en fixerad storlek. För en dynamisk stack saknas en "låda" det finns alltså inga restriktioner.Den tilldelade koden utgjorde en bra start punkt trots det tillkom vissa svårigheter. Den största var att inse att man måste initialisera sin variabel (int top;) till -1 för att visa att en stack är tom. Detta då den fungerar som stackpekare som måste peka över det översta elementet stacken. Rent kodmässigt gick hela balletten till som följande:

```
public class StaticStack {
    int[] stack;
    int top;

    public StaticStack(int size) {
        stack = new int[size]; // Initializera
            stackens array med given storlek
```

```
        top = -1; // Initializera "Top" till -1,
            stacken  r  tom!
    }

    public void push(int val) {
        if (top == stack.length - 1) { // Checka stack
            overflow.
            System.out.println("Stack overflow. gick
                inte pusha " + val);
        } else {
            stack[++top] = val; // Inkrementera push
                och pop
        }
    }

    public int pop() {
        if (top == -1) { // Checka f r stack
            underflow
            System.out.println("Stack underflow. Inget
                element att poppa.");
            return -1; // Returnera v rde (bevisa att
                stacken  r  tom)
        } else {
            return stack[top--]; // Returnera
                elementet h gst upp och dekrementera
                en g ng
        }
    }
```

En annan var att inse hur vi skulle loopa våra villkorstaser och hur det rent konkret skulle se ut. Det gick att lösa med hjälp av en del felsökning och ”try and error”.

3 Dynamisk Stack

Som nämnt tidigare har en dynamisk stack inte samma restriktioner som en statisk stack. Detta innebär att den kan hantera ”overflow” då en dynamisk stack kan skapa en ny större array och flytta över elementen till den. Detta betyder dock att stacken måste minska när uttrymme inte längre behövs, för att spara minne. Detta sparar också såklart tid. Mycket kod från den statiska stacken kunde mer eller mindre kopieras över när den dynamiska stacken skulle göras. En skillnad som behövdes göras var att implementera en funktion som skapar en ny större array för situationer som nämndes ovan. Detta kan bäst göras genom att dubblera storleken på arrayen, någonting som diskuterades i en föregående föreläsning. Varför just en dubblering valdes var för att en annan

lösning, till exempel att kvadrera en array väldigt snabbt går över styr och det som var tanken att vi skulle vinna (tid och minne) upphör snabbt att ske. Per automatik betyder det att vi måste ta höjd för det när vi gör pop, speciellt när vi minns vad definitionen för en sådan funktion är. Vi kontrollerar om antalet element är 1/4 av arrayens längd. Är det så samtidigt som storleken på arrayen är större än 4 minskas längden till att bli hälften så stor. Konsekvensen blir att stacken använder mindre minne, vilket var precis vår intention. Kod mässigt ser de berörda kodrader ut som följande:

```
    public void push(int value) {
        if (top == size) { // Checka storleken
            resize(size * 2); // Dubblera storleken
        }
        items[top++] = value; // inkrementera pointer!
    }

    // poppa element fr n stack
    public int pop() {
        if (top == 0) { // kontrollera om tom
            throw new StackUnderflowException("Stack
                underflow. inga element att poppa.");


        // kontrollera om stacken  r  1/4 full, g
            ner
        if (top > 0 && top == size / 4 && size > 4) {
            resize(size / 2); // halvera storlek p
                stack
        }

    }


    private void resize(int newSize) {
        int[] newItems = new int[newSize];
        for (int i = 0; i < top; i++) {
            newItems[i] = items[i]; array
        }
        items = newItems;
        size = newSize;
    }
```

4 Räknare - i HP35 stil!

För att kunna bygga en räknare behövde vi använda oss av en av våra stackar som vi redan byggt. Där efter fylla på med kod för att stacken ska kunna användas som en räknare. Lyckligtvis så har en skelett kod tilldelats så att man inte behöver börja från 0. Man fick välja mellan att göra en räknare baserad på en statisk stack eller en dynamisk stack. Skribenten av den här rapporten valde att implementera en dynamisk stack till sin räknare, då den har en hel del fördelar. En ny fil skapades i *Visual Studio Code*, där skelett koden lades in. Där efter beslutades om att endast integer (heltal, utan komma) bara kommer kunna behandlas. Dessa fick variablerna ”a” och ”b” och användes för beräkningar som sker under koden för respektive operand. Push och pull behövde också integreras med den dynamiska stacken. En funktion som också behövde integreras var att se till att räknaren inte kan användas på fel sätt. Dvs att om operand kommer först eller mellan 2 integers ”a” och ”b” måste ett fel meddelande komma upp. En annan var att det endast är tillåtet att stoppa in nummer och operationer i räknaren. Alltså innebär det att symboler som: ”!”, ”?” och ”@” bland annat inte är tillåtna. Divition med 0 ska också inte häller vara tillåtet och ger därför ut felmeddelande när det sker. Delar av koden ser därför ut som följande:

```
switch (input) {
                case "+":
                    if (stack.size() < 2) {
                        System.out.println("Error:
                            inte tillr ckligt m nga
                            operander f r addition");
                        break;
                    }
                    int b = stack.pop();
                    int a = stack.pop();
                    stack.push(a + b);
                    break;
                    }
                    b = stack.pop();
                    a = stack.pop();
                    if (b == 0) {
                        System.out.println("Error: aja
                            baja dela inte med 0!");
                        stack.push(a); // pusha
                            tillbaka tal d  divition
                            inte h nde
                        stack.push(b);
                        break;
                    }
```

Med denna räknare kan vi enkelt ta reda på vad 4 2 3 * 4 + 4 * + 2 - blir! Det ger 42.