Assignment 2 - Stackar & HP35 calculator

Jesper Hesselgren

September 4, 2024

Introduktion

I denna uppgift implementeras en enkel kalkylator som kan beräkna matematiska uttryck skrivna i "Reverse Polish Notation" (RPN). Syftet med uppgiften är att ge en förståelse för hur datastrukturen stackar fungerar och kan användas. RPN används för att uttrycka matematiska beräkningar utan behov av parenteser. Två versioner av stackar implementeras i programmeringsspråket Java: en statisk stack med fast storlek och en dynamisk stack som kan växa och minska efter behov. Dessa datastrukturer används för att bygga en fungerande kalkylator som tar emot tal och operatorer från användaren, utför beräkningarna och returnerar resultatet.

Statisk stack

Den statiska stacken implementeras med hjälp av en array stack och en pekare top, som håller reda på nästa lediga position i stacken. Stackens storlek bestäms när en instans av klassen skapas genom att ett storleksvärde skickas till konstruktorn. Vid varje operation används pekaren för att bestämma var ett nytt element ska läggas till eller varifrån ett element ska tas bort. Klassen innehåller två huvudmetoder: push och pop, som används för att lägga till respektive ta bort element från stacken.

När ett nytt element läggs till i stacken via push-metoden, kontrolleras först om stacken är full genom att jämföra pekaren top med arrayens storlek. Om stacken är full, kastas ett StackOverflowError. Om det finns plats, placeras det nya elementet på den plats som top pekar på, och pekaren ökas med ett steg för att förbereda för nästa inmatning.

Pop-metoden tar bort det översta elementet i stacken. Om stacken är tom, det vill säga om pekaren top är 0, kastas ett IllegalStateException för att signalera att det inte finns några element att ta bort. Om stacken inte är tom, minskas pekaren med ett steg och det element som tidigare låg överst returneras.

```java
public class StaticStack {

    int[] stack;
    int top = 0;

    // Constructor
    public StaticStack(int size) {
        stack = new int[size];
    }

    public void push(int num) {
        if (top == stack.length) {
            throw new StackOverflowError("Stack is full");
        } else {
            stack[top++] = num;
        }
    }

    public int pop() {

        if (top == 0) {
            throw new IllegalStateException("Stack is empty");
        }
        return stack[--top];
    }
}
```

Dynamisk stack

Den dynamiska stacken är konstruerad på ett liknande sätt som den statiska stacken och implementerad med en array och en pekare top, som håller reda på nästa lediga position i stacken. Den dynamiska stacken har dock även attributet size, som håller koll på stackens aktuella storlek. På samma sätt som den statiska stacken innehåller den dynamiska stacken de två huvudmetoderna push och pop, men den innehåller också en hjälpfunktion kallad resize, som anropas när stacken behöver utökas eller minskas.

När ett nytt element läggs till i stacken via push-metoden, kontrolleras först om stacken är full genom att jämföra pekaren top med stackens storlek (size). Om stacken är full, anropas resize-metoden för att fördubbla stackens storlek, vilket gör att nya element kan läggas till utan att överskrida ar-

rayens gränser. Därefter läggs det nya elementet till på den position som pekaren top pekar på, och pekaren ökas med ett steg.

Pop-metoden tar bort och returnerar det översta elementet i stacken. Om stacken är tom, kastas ett IllegalStateException för att signalera att det inte finns några element att ta bort. När ett element har tagits bort, kontrolleras om stackens storlek är större än fyra och om mängden använda element är en fjärdedel av stackens totala storlek. Om så är fallet, anropas resize-metoden igen, men denna gång för att halvera storleken på stacken, vilket sparar minne när få element används. Anledningen till att vi kontrollerar att stackens storlek är större än fyra är att det med stor sannolikhet inte är lönsamt att minska stacken ytterligare, eftersom vi inte skulle spara mycket minne. Dessutom skulle vi troligtvis snart behöva förstora stacken igen, vilket skulle innebära onödiga resize-operationer och ineffektiv kod.

Resize-metoden ansvarar för att skapa en ny array med den nya storleken och kopiera över alla element från den gamla stacken till den nya. Den uppdaterar också värdet för size så att stacken kan fortsätta växa eller krympa dynamiskt beroende på behov.

```
public class DynamicStack {

    int[] stack;
    int top = 0;
    int size;

    public DynamicStack(int size) {
        this.size = size;
        stack = new int[this.size];
    }

    public void push(int num) {
        if (top == size) {
            resize(size * 2);
        }
        stack[top++] = num;
    }
```

```
    public int pop() {

        if (top == 0) {
            throw new IllegalStateException("Stack is empty");
        }

        int data = stack[--top];
        if (size > 4 && top > 0 && top == size / 4) {
            resize(stack.length / 2);
        }
        return data;
    }

    private void resize(int newSize) {
        int[] newStack = new int[newSize];
        for (int i = 0; i < top; i++) {
            newStack[i] = stack[i];
        }
        stack = newStack;
        size = newSize; // Updating the size value of the stack
        System.out.println(size);
    }
}
```

Implementering av stack i calculatorn

Genom att använda stackar kan vi implementera en kalkylator som utnyttjar "Reverse Polish Notation" för att utföra addition, subtraktion och multiplikation. I kodexemplet nedan har kalkylatorn implementerats med hjälp av vår dynamisk stack. Kalkylatorn fungerar genom att användaren matar in tal och operatorer via terminalen. När användaren matar in ett tal, placeras det på stacken, och vid inmatning av en operator utförs operationen på de två översta elementen på stacken. Resultatet placeras sedan tillbaka överst i stacken och resultatet av operationen skrivs samtidigt ut i konsolen.

Först skapas en instans av den dynamiska stacken med en initial storlek på 16. Därefter initieras ett objekt av klassen BufferedReader, som används för att läsa in användarens inmatning via terminalen. Programmet går sedan in i en while-loop, där programmet kontinuerligt tar emot och bearbetar användarens input.

Programmet fortsätter köras så länge användaren matar in nya tal och operationer. När en tom rad matas in, avslutas kalkylatorn och det översta resultatet från stacken skrivs ut i konsolen. Denna konstruktion av kalkyla-

torn gör det möjligt att utföra flera på varandra följande beräkningar utan att behöva använda parenteser eller ta hänsyn till operatorprioritet, vilket är en av fördelarna med Reverse Polish Notation".

```
public class HP35 {

    public static void main(String[] args) throws IOException {

        DynamicStack stack = new DynamicStack(16);

        System.out.println("HP-35 pocket calculator");
        boolean run = true;

        BufferedReader buffer =
        new BufferedReader(new InputStreamReader(System.in));

        while (run) {
            System.out.println("> ");
            String input = buffer.readLine();

            switch (input) {
                case "+":
                    int n1 = stack.pop();
                    int n2 = stack.pop();
                    int sum = n1 + n2;
                    stack.push(sum);
                    System.out.println("The result is: " + sum);
                    break;
                case "-":
                    int n3 = stack.pop();
                    int n4 = stack.pop();
                    int diff = n3 - n4;
                    stack.push(diff);
                    System.out.println("The result is: " + diff);
                    break;
                case "*":
                    int n5 = stack.pop();
                    int n6 = stack.pop();
                    int prod = n5 * n6;
                    stack.push(prod);
                    System.out.println("The result is: " + prod);
                    break;
                case "":
```

```
                run = false;
                break;

            default:
                Integer nr = Integer.parseInt(input);
                stack.push(nr);
                break;
        }
    }
```