HP35.

Mattias Sandberg

2 Sep 2022

Introduktion

Denna uppgift går ut på att implementera en miniräknare som kan räkna matematiska uttryck med hjälp av ”reverse Polish notation”. Detta ska vi göra för att få en förståelse för hur stacken fungerar.

Det vi ska göra och vad reverse Polish notation går ut på är att skriva matematiska uttryck med operanden sist. tex 6+4 skrivs 64+. Fördelen med detta är att man kan komma undan paranteser.

Vi kommer att använda en stack vilket är en datastruktur som tillåter två grundlägande operationer vilket är pop och push. En pop operation kommer ta bort och lägga till föremålet högst i stacken, dvs om stacken inte är tom. Vi kan även kolla om stacken är tom och peka på olika obijekt i stacken med hjälp av pekare.

Miniräknaren

I Miniräknare klassen i min java kod så implementerar jag funktioner som används som metamatiska operander som finns i klassen Item/enum. Dvs det matemetiska operanderna ADD, SUB, MUL, DIV och VALUE (addition, subtraktion, multiplikation, division och värde). I stack.pdf filen som vi fick fanns det rutinen för ADD operationen. Jag gjorde likande för det resterande matematiska operationer dvs att jag popar x och y värderna och sen pushar tex (x + y) för att sist använda break funktionen för att stoppa loopen. Att tilläga är att dessa kodstycken körs i ”public void step”. Nedan kommer koden för följnade operationer:

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

                   }
```

Jag använder också en item class för att ge våra föremål som kommer implementeras en typ och ett föremål. En del av koden (den mest väsentliga) som skrevs i item class representeras nedan:

```
        public Item(int value) {
        this.value = value;
        this.type = ItemType.VALUE;
    }

    public Item(ItemType itemType) {
        this.value = 0;
        this.type = itemType;
    }
```

Stacken

Vi ska använda oss av en stack. Det finns både statisk stack och dynamisk stack. En statisk stack har en fixerat storlek (utrymme). En dynamisk stack är mer komplex och har förmågan att växa i förhållande till item. Frågar att besvara för den statiska stacken.

fråga1:"Does the pointer point to the location above the top of the stack or does it point to the top of the stack?".

Svar:pekaren pekar alltid på det senase värdet i stacken dvs i toppen av stacken.

fråga2:"What is the value of the pointer when the stack is empty?". Svar: I en tom stack pekar pekaren på den nästa fria platsen i stacken.

fråga3:"What should you do when a program tries to push a value on a full stack (stack overflow)?". Svar: Du får antingen utöka (hårdkodat) på den statiska stacken, och den dynamiska stacken ordnar det aoutomatiskt.

fråga4:"What should happen when someone pops an item from an empty stack?". Svar: Om jag anropar en pop() metod på en tom stack så får jag meddelandet "EmptyStackException"

Då den dynamiska stacken var den mest komplexa stacken att konstruera så jag kommer jag visa delar av den koden nedan (pop och push metoderna):

```
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;
        }
```

I det flesta fall är tidskomplexiteten O(1) om den inte behöver kopieras över till en större array. Om den behöver göra det så blir tidskomplexiteten O(n). Detta gäller för den dynamiska stacken.

Jag använde en Nanotime-funktion för varje loop (operation) för att beräkna hur mycket tid som förloras mellan den dynamiska och den statiska stacken när operationer utförs. Enligt mina uträkningar tar det cirka $20-30$ procent längre tid för den dynamiska stacken beroende på inputs. Tex: 6000ns för den statiska och 7500ns för den dynamiska stacken.

Räkna ut din sista siffra

För att räkna ut sin sista siffra i sitt personnummer så måste jag först multiplicera det första 9 siffrorna med 2,1,2,1 osv, och sen addera ihop resultatet. Sedan tas summan med modulo 10 och det subtraheras med 10.

10((y1 2+y2+m1 2+m2...)mod10), För att vi ska kunna göra detta med hjälp av våran miniräknare program måste vi implementera en MOD10 (modulo 10) operation. Denna operation läggs i funktionen är en case i public void step som ligger i public class Calculator. Själva modulo 10 skrivs i stack.push som (x - (x/y)*y).

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

Mitt personnummer är 200211061650 och om jag stoppar in värdena i min kalkylator så får jag att min sista siffra ska vara 1. Jag får samma om jag räknar för hand. om jag dock tar mitt ”resultat” och tar 9 minus det istället för 10 minus det så får jag rätt. Många av mina kamrater har gjort samma sak och får då rätt svar. men jag förstår den grundläggande principen.