En kalkylator

Hiba Qutbuddin Habib

Spring Fall 2022

Introduktion

I denna uppgiften kommer vi att studera ”stack” närmare samt skapa djupare förståelse för hur en ”stack” egentligen fungerar. För kunna studera en ”stack” närmare så kommer vi att implementera en kalkylator som kan beräkna matematisk uttryck som i sin tur beskrivs med omvänd ”polish notation”. Vi implementerar en ”static stack” samt en ”dynamic stack”.

An expression

En klass av typen Item skapas. Varje Item beskrivs med en ”Type” samt ”Value”. Konstruktorn och metoderna för objektet defineras enligt följande:

```
  public Item(int value, ItemType type) {
        this.type = type;
        this.value = value;
    }
    public int value()
    {
        return value;
    }
    public ItemType Type()
    {
        return type;
    }
}
```

The calculator

En objekt av typen ”Calculator” skapas. ”Calculator” definras av ett uttryck, pekare samt en stack. De metoderna som definerar en ”Calculator” är bland annat metoden ”run” som beskriver hur programmet ska köras, samt metoden”step” som beskriver steg för varje type. Metoden ”step” kompletteras enligt följande:

```
            case SUB: {
            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);
            if (y == 0) {
         System.out.println("Undefin");}
            break;}
        case MUL: {
            int y = stack.pop();
            int x = stack.pop();
            stack.push(x * y);
            break;}
        case VALUE:{
            stack.push(nxt.value());}
```

Implementera stack

Static stack

I denna delen skapas en stack alltså en objekt av typ stack. Denna stacken ska vara static dvs att den ska ha en begränsad storlek. Efter att stacken skapas så implementeras den i kalkylator.

För att kunna göra en objekt av typen stack så behövs det en pointer. Pointer är en variabel som pekar på toppen av stack. Då i detta fall är stacken en vektor så pekar denna pointer till en tom vektor som vi beskriver med värdet (-1).

```
int top;
top =-1
```

De två viktigaste funktioner i en stack är ”push” and ”pop”. När funktonen ”push” användes så innebär det att vi lägger en värde på toppen av stacken, då den värde som ligger redan på top gå ner en index och denna nya värdet vi pushar kommer att ligga på toppen, alltså först i stacken. När stacken är full så kan vi inte pusha in något mer, då det handlar om en ”static stack” då kan användare meddelas med en undantag enligt följande:

```
public int push(int n){
       if (isFull()) {
            throw new java.lang.ArrayIndexOutOfBoundsException("Stack is full"); }
    return stack[++top]=n;}
```

Den andra viktiga funktionen för stack är ”pop”, detta innebär att värdet som ligger på toppen tas bort och returneras så att den värde som är under blir på topp. Om stacken är redan tom så bör användare meddelas om det med ett meddelande enligt nedan:

```
  public int pop(){
     if(isEmpty()){
```

```
        throw new java.lang.NullPointerException("Satck is Empty");}
 return stack[top--]; }
```

Dynamic stack

Skillnaden mellan dynamic och static stack är att det dynamic stack kan expandera när stacken är full till skillnad från static. Dynamic stack kan också kontrahera när den blir tom. För att kunna implementera en sådan stack så behövds två ny metoden, expand och kont, dessa implementerades enligt nedan:

```
public void expand() {
      int newStack[] = new int[maxsize * 2];
      for (int i = 0; i < maxsize; i++) {
          if (isFull()) {
              newStack[i] = stack[i];}
          stack = newStack;
          maxsize *= 2;}}

public void kont () {
          int newstack1[] = new int[maxsize / 2];
          for (int i = 0; i < maxsize; i++) {
              if (isEmpty()) {
                  newstack1[i] = stack[i]; }
              stack = newstack1;
              maxsize = maxsize / 2;}}
```

Benchmarks

När vi testar dessa två implementation , static stack and dynmaic stack så noterar vi att det finns skillnad i exekveringstid. Då när static stack är full så krävs det inget stort arbete utan kommer programmet att bara kasta en undantag och därför går exekvering snabbare. När det gäller dynmaic stack så krävs det ytterligare arbete när stack är full, för de behövs en stack expandation, koppiering från den gamla stacket, ändring av maxsize till det nya maxsize sen utföring av operationer, allt detta gör att vid användning av dynamic stack så kommer exekvering att ta längre tid när stacken är full och expandation krävs. Däremot så dynamic stack är mer praktiskt att använde än static, då man behöver inte tänka på maxkapacitet då den kan expandera såsom kontrahera. Dock med static stack så är det viktigt att hålla koll på antal data man matar in till förhållande till stackets storlek, detta är inte så praktiskt vid arbete med en stor mängd data.

Försök	**Dynamic**	**Staic**
1	3.7	1
2	4.4	1.3
3	2.4	1.4
4	4.9	1.3
5	2.6	1.2

Table 1: Tid för dynamic och static stack vid 5 mätningar i millisekund

Mätningar i tabeln ovan visar att den dynamiska stacken kräver längre tid än den statiska. Mätningar som jag utförde gav en väligt lite skillnad mellan exekverings tid för både stackarna. Detta kan bero på att ökningen av mängd data vara inte för stor, vilket gjorde att vid expandation av den dynamiska stacken, mängd data som kopierades över var liten. I tabellen över kan vi också se att tid för den statiska stacken är konstant, det tyder på att den tillhör mängden O(1). När det gäller den dynamsiska stacken så vet jag att tiden kommer att tillhöra mångden O(log(n)), dock detta kan jag inte se tydligt i mina mätningar vilket kan bero på mängd data jag använde mig av.

Calculate last digit

För att beräkna den sista siffran i personnummer så används denna formel $10-((y1*'2+y2+m1*'2+m2...)mod_{10}))$ Formel ovan kan skrivas om i omvänd polish notation, vilket jag använde för att beräkna min sista siffran. Resultat blev följande: $1092*91*+02*+81*+22*+21*+82*+31*+42*+10\%-$.

Den speciella tecken $*'$ innebär att om vi har ett tal x och tal y så först beräknas $x*y$, om resultatet blir x_1y_1 så adderas x_1+y_1. Detta kan tillämpas i omvänd polish notation genom att först skriva $xy*$ sen läggs en $+$.

Implementering av $*'$ samt $mod_{10}\%$ kan ske enligt följande:

```
case MULADD:{
              int y=stack.pop();
              int x= stack.pop();
              stack.push(x*y);
              int m=stack.pop();
              if (m>10){
                  stack.push(m %10);
                  stack.push(m=m/10);
                  int x1 =stack.pop();
                  int x2=stack.pop();
                  stack.push(x2+x1);}
              if (m<10){
                  stack.push(m); }
              break; }

 case modulos:{
              int y=stack.pop();
              if(y>10){
                  int n = y / 10;
                  int x=n*10;
                  stack.push(y-x); }
              if(y<10){
                  stack.push(0); }
              break;}
```