Calculator

Adalet Adiljan

2 September 2023

Introduction

Till denna uppgift hade vi i uppdrag att implementera två olika typer av stackar av statisk och dynamisk typ från grund utan importering av diverse Java bibliotek eller Arraylist. I denna rapport kommer vi att upplysa tillvägagångssätten för den statiska och dynamiska datatrukturen och vår slutgliga resultat.

Stackar

Stack är en linjärtyp av datastruktur som används för att lagra och hantera element i en ordnad sekvens, utifrån en princip som kallas LIFO (Last in, first out) där det sista elementet som läggs till på stacken är den första elementet som tas bort. De viktiga operationerna som har använts till detta syfte är så kallade Push och Pop funktionerna. Push-funktionen har i syfte att lägga till ett element på toppen av stacken, där den nya elementet blir det översta elementet medan Pop funktionen har i syfte att ta bort och returnera det översta elementet.

Statisk

Statisk stack har en bestämd storlek som bestäms när stacken skapas, och kan inte växa eller krympa bortom denna initiala storlek. Vid varje push-anropning anordnas det nya element till den enda allokerade arrayen av den givna storleken.

Dynamisk

I kontrast till den statiska versionen kan den dynamiska stacket växa eller krympa beroende på antalet element den innehåller utan att vara bunden till någon storlek. Istället för att den allokerade arrayen av den givna storleken skulle leda till stackoverflow, utökar push-funktionen storleken på stacket till en större. Vad gäller pekaren

Metod

Innan implementationen av stacken har vi börjat att först definiera och komplettera de nödvändiga funktionaliteterna för vår respektive kalkylator-, Item- och Itemtype klass. Då ingen Java bibliotek av den dynamiska respektive statiska klassen tilläts att importeras behövde vi definiera de som egna publika klasser. Lättast var att skapa en egen abstrakt klass med de nödvändiga abstrakta metoderna push och pop och låta stack- och dynamiska versionen utdvidga den. En abstrakt klass kan inte instansieras direkt, utan används som ”mall” för andra klasser. På så sätt definierar abstrakta klasserna ett gränssnitt för vad subklasserna behöver innehålla.

Implementering

I den statiska klassen börjar vi med att ge en fixerad storlek på arrayen när stacken skapas. Stacken ska allokera en array av denna storlek och hålla en stackpekare (index-positionen vid varje push- eller pop operation. Vad gäller stackpekaren för push-operationen såg jag till att först inkrementera startvärdet av top=-1 med 1 enligt följande:

```
top = top + 1;
```

På så sätt låter vi stackpekaren peka på en ny position innan vi refererar index-positionen med det nya elementet tills pekarens längd inte längre är mindre än stackens längd. När användaren har överskridit uttrymmet, har vi skapat ett villkor som exekverar en output funktion som meddelar till användaren ett felmeddelande att stacken är full. Vid tillämpningen behövde vi även tänka på att ge den startvärdet - 1 för att indikera en tom stack.

Då den dynamiska stacket ska kunna växa och krympa efter behov och inte ska vara bunden till en fast storlek, har vi anpassat både i vår push- och pop-operation till att utvidga och minska längden av stacken när särskilda villkor vid repsektive villkor var uppfyllda, enligt koden nedan.

```
public void push(int value){
if(top == length - 1) {
int newCapacity = length*2;
int[] localArray = new int [newCapacity];
}
```

En väsentlig del i att implementera vår pop-funktion för den dynamiska stacket har varit att inte utöka eller minska längden av arrayen med ett extra uttrymme då detta kan påverka prestandan och minnesanvändningen för vår kalkylator. Om stacken ofta växer och krymper med en storlek kan detta leda till onödig resursanvändning eftersom att man kommer göra

många små reallokeringar, vilket kan vara dyrt när vi talar om tiden och minnesanvändningen som nyttjas. Till push-funktionen har jag istället ökat storleken med dubbla enbart när den givna villkoret är givet.

```
else {
int poppedValue = newArray[top];
top--;
if ( top < newArray.length / 4 ) {
int newLength = newArray.length/2};
:
}
```

Vad gäller pop-funktionen enligt ovan har jag skapat en if-else villkor, där if-villkoret printar ut ett felmeddelande då pekaren inte pekar på någon index-position än för att indikera att arrayen är tom och inte har någon pekare att peka på. Med else-villkoret poppar vi den sista elementet i den befintliga arrayen så länge stacket inte är tom.

Innan vi utför dessa operationer sparar vi först det befintliga värdet i en temporär variabel sedan dekrementerar pekaren. Då vi bör tänka på att optimera minnesanvändningen och effektiv hantering av resurserna, har vi suttit ett villkor i att utföra en halverad förkortning av arrayens nuvarande längd, enbart om antalet element underskrider en fjärdedel av arrayens längd.

Prestandamätning (benchmark)

Tables

repetitions	dynamic	static	cost
100	63.5	38.6	23.9
800	38.7	22.5	16.2
1400	45.8	20.8	25

Table 1: Antalet repetitioner för push- och pop-operation för den dynamiska och statiska klassen, tiderna i mikrosekunder samt den uträknade kostnaden efter att ha subtraherat klassarna till varandra.

Tidsresultatet indikerar hur effektiv de enskilda implementationerna av push- och operationerna är för de respektive klasserna. Efter 100 repitioner verkar den statiska implementationen vara mer effektiv, eftersom den inte behöver några dynamiska ändringar i array-storleken, vilket förklarar varför tiden under två av de tre tillfällen är längre hos den dynamiska. Anledningen till att tiden var kortast efter 800 repitioner för den dynamiska implementationen kan förklaras av vår storlekshantering i de fall som den behöver

utvidgas respektive förkortas. Eftersom att vi vid ökning kan förlänga respektive minska längden med en faktor, kan det vara att vi har haft fördelen av att redan ändrat storleken på den underliggande arrayen och därmed inte behövt genomföra många storleksförändringar. Detta har alltså gett oss en insikt för hur prestandan kan påverkas beroende på hur stort värde vi matar in i förhållande till vår storlekshantering. I detta testfall påvisar den statiska stacken vara snabbare än den dynamiska pga. att den inte har storleksförändringar som kräver O(n)-tid.

Tidskomplexitet

Tidskomplexitet kan vara relevant för att mäta hur snabbt en algoritm eller en operation körs som en funktion av storleken på dess input. Man får en uppfattning kring hur effektiv en algoritm eller en operation utförs när det gäller tiden det tar att köra den. Här kan det vara intressant att analysera och jämföra stack-versionernas respektive funktioner. I beskrivningarna nedan används O(1) och O(n) noteringar. O(1) innebär att tidskomplexiteten är konstant oberoende av inputens storlek, medan (n) innebär att tidskomplexiteten är proportionell på storleken av indatan. Stor input medför längre tid medan ett ganska litet värde medför en mindre tid.

Dynamiska stacken

I push-operationen har den en tidskomplexitet på O(1) i de flesta fall eftersom den huvudsakligen utför en tillsättning av ett element till en array. När arrayen är full, kommer det att ta O(n) tid då alla de befintliga element behöver kopieras till den nya arrayen. Genomsnittligt, ligger den amortiserade tidskomplexiteten på O(1).

Vad gäller pop-funktionen har den tidskomplexiteten O(1) eftersom att den översta och direkt tillgängliga elementet tas bort från stacken, vilket medför en konstant operation.

Statiska stacken

I de flesta fall ligger tidskomplexiteten för push-operationen ligger på O(1), eftersom den utför en enkel tillsättning av ett element till en array med fixerad storlek. Det behövs därmed inte att dubbla storleken eller kopiera elementen.

Likt vad gäller den dynamiska klassen, ligger tidskomplexiteten för popfunktionen på linjär ökning O(1), eftersom den finns direkt tillgänglig som överst element på stacken.

Diskussion

Trots att det är vår första uppgift, finns det en hel del saker som jag har lärt mig och som jag vet att jag kommer att ha nytta utav i framöver. Förutom att ha gett mig värdefulla praktiska erfarenheter av stack-implementering av dynamiska och statiska typ, kommer jag nog att ha mest nytta utav uppbyggandet av prestandamätningen. Denna gav mig en värdeful insikt i hur prestandamätningen kan variera och skilja sig mellan dessa olika versioner. På så sätt kan vi göra viktiga avvägningar som inte riskerar oss att hamna i någon stor bekostnad i tid.