# Övning 4 Minneshantering

Fråga 4-1:
*Stacken* fungerar enligt principerna LIFO (Last In First Out) och FILO(First In Last Out), dvs som en trave böcker ovanpå varandra. Programmet allokerar och kommer åt variabler och metoder i stacken med hjälp av direkt minnesåtkomst. För accesstiden har det ingen påverkan var de ligger. Metoder som är av value-typ lagras på stacken. Om metod A körs lagras den på stacken och om den i sin tur anropar en metod B, kommer metod B att läggas ovanpå metod A på stacken. När metod B är klar kommer den minnespositionen att raderas först och därefter raderas metod A när den är klar. Stacken har snabb åtkomst och automatisk minnesrensning. Vid rekursiva metoder kan man få en stack overflow eftersom det är endast vid basfallet i rekursionen som stacken börjar rensas. Då det är den sista metoden in som ska rensas ut först(LIFO).

```
1 int a = 4;
2 int b = a;
3 int A(int a, int b) { return B(a, b); }
4 int B(int a, int b) { return a + b; }
5 Console.WriteLine(A(a,b));        //A anropar B, B raderas från stacken först då den
6                                   //anropas sist
```

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | B(a,b) | B raderas | |
| | | | A(a,b) | A(a,b) | A(a,b) | A raderas |
| | | b = 3 | b = 3 | b = 3 | b = 3 | b = 3 |
| | a = 4 | a = 4 | a = 4 | a = 4 | a = 4 | a = 4 |
| ***Steg*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** |

*Tabell 1: Stack*

| Stack | Heap |
|---|---|
| `int a = 4; //Stack`<br>`int b = a; //Stack`<br>`Console.WriteLine(ReferenceEquals(a, b));`<br>`Output: False` | `string c = $"{a} {b}";  //Heap`<br>`string d = c;  //Heap`<br>`Console.WriteLine(ReferenceEquals(c, d));`<br>`Output: True` |

*Tabell 2: Stack vs Heap*

På *heapen* allokeras reference-typer som till exempel strängar, objekt, klasser, delegater mm, dessa fungerar enligt principen att en pointer lagras i stacken, pointern referer sedan till en plats på heapen där själva objektet allokeras. Heapen har ingen automatisk minnesrensning utan hanteras av GC(Garbage Collector) när objektet inte behövs.

Fråga 4-2. Skillnad är hur de allokeras och hur de rensas. Value type på Stacken och Reference type på Heapen.

| | | **Stack** | | **Heap** |
|---|---|---|---|---|
| `int a = 4;` | -> | a = 4 | | |
| `int b = a;` | -> | b = 4 | | |
| `string c = $"{a} {b}";` | -> | c(ref) | -> | c(obj) |

*Tabell 3: Value Type vs Reference Type*

Fråga 4-3. ReturnValue() är en Value Type då den endast innehåller value types och ReturnValue2() är en Reference Type då den innehåller en klass MyInt() som är en Reference Type. När man sätter y=x blir y.MyValue = 3 eftersom de är då samma objekt, dvs både objekt x och objekt y refererar till samma minnesutrymme. Ändrar vi sedan y.MyValue = 4 ändras dämed även x.MyValue till 4 då x referar till samma minnesplats som y.

Fråga 4.1-2 Listans kapacitet ökar vid 1, 5, 9, 17, 33 element osv, dvs när den underliggande arrayen är full vid 4, 8, 16, 32 osv. och ytterligare kapacitet behövs.

Fråga 4.1.3 Den dubbleras.

Fråga 4.1-4 På grund av att den använder en underliggande array, vilka har en fast kapacitet när de är initialiserade. En ny underliggande array skapas när den är full. Hur stor ökningen av kapaciteten är bestäms av definitionen för List med hänsyn till att så effektivitet och prestanda.

Fråga 4.1-5 Nej.

Fråga 4.1-6 När man har behov av stora listor och frekventa förändringar av antal element i listan.

4.2-1 ExamineQueue
Simulering av ICA-kö
a ICA öppnar, tom kö

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |

b Kalle ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Kalle | | | | | | |

c Greta ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Kalle | Greta | | | | | |

d Kalle blir expedierad och lämnar kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Greta | | | | | | |

e Stina ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Greta | Stina | | | | | |

f Greta blir expedierad och lämnar kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Stina | | | | | | |

g Olle ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Stina | Olle | | | | | |

4.3-1 ExamineStack
Simulering av ICA-kö
a ICA öppnar, tom kö

| | | | | | | |
|---|---|---|---|---|---|---|

b Kalle ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Kalle | | | | | | |

c Greta ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Kalle | Greta | | | | | |

d Kalle blir expedierad

| | | | | | | |
|---|---|---|---|---|---|---|
| Kalle | Greta | | | | | |

e Stina ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Kalle | Greta | Stina | | | | |

f Greta blir expedierad

| | | | | | | |
|---|---|---|---|---|---|---|
| Kalle | Greta | Stina | | | | |

g Olle ställer sig i kön

| | | | | | | |
|---|---|---|---|---|---|---|
| Stina | Olle | Stina | Olle | | | |

Fråga 1 Om kön fylls på kan ingen lämna kön.

4.4 Fråga 1: Jag använder stack. Man kan då loopa igenom strängen och lagra alla öppna parantestecken. När loopen kommer till ett stängt parantestecken kan vi jämföra med den senaste öppna parantestecknet som nu är överst i stacken. På det sättet kan vi kontrollera om varje stängande parantes har en öppnande parantes.