# Köer

Mattias Sandberg

06 Okt 2022

## Introduktion

I denna uppgift så ska vi konstruera och undersöka en mycket vanlig datastruktur vilket är köer. Vi ska implementera en rad av element som väntar i kö och om nya element läggs till i listan så behöver vi vänta tills det föreliggande föremålen har hanterats innan det ny kommande föremålen hamnar längst fram i listan. Köer kallas också för FIFO vilket står för "First in first out" vilket liknar förklaringen av en stack vilket kallas för "last in first out". Vi kommer att konstruera denna datastruktur på två olika sätt. Den första är genom att använda en array och den andra är genom att använda en länkad lista som vi har konstruerat i tidigare uppgifter. Att konstruera programmet med arrayer bör vara mest effektiv men däremot mer komplicerad och därför kan länkade listor vara fördelaktigt och det är det som jag kommer att undersöka i denna rapport. Nedan kommer en simplifierad bild över hur detta kan visualiseras från hemsidan "*https* : *//www.programiz.com/java − programming/queue*" :


Figure 1: Simpliferad bild över ett "Kö" datastruktur

### A linked list

Det mest triviala sättet att implementera ett kösystem i programmering är genom att använda en länkad lista. Kön kommer då enbart att ha en fast egendom vilket är "huvudet" och nya element adderas enbart till slutet av listan. Nedan kommer kod bitar där jag har implementerat denna datastrukturen utifrån det som gavs i PDF: en. Först och främst så deklarerar jag klassen och dess nod för att sedan implementera funktionen "enqueue" som är väldigt simpel med enbart är några kodrader:

```
public class Queue()<P>
{
  Node queue;
  private class Node
  {
    P foremal;
    Node nasta;

    private Node(P fore, Node nast)
    {
      this.foremal = fore
      this.nasta = nast
    }
  }
  public enqueue(P fore)
{
  this.queue = Node(fore, queue)
}
}
```

Tidskomplexiteten för att ta bort ett element och lägga till ett element är konstant dvs Ordo O(1). Vi kan dock utveckla detta program genom att implementera en pekare till det första elementet (huvudet) och en annan pekare till det sista elementen i listan. Fördelen med detta är att vi nu kan direkt gå till det sista elementet när ett nytt föremål ska läggas till i listan. Eftersom programmet inte behöver gå igenom hela listan från början för att lägga till ett nytt element så ska detta vara mindre kostsamt. Nedan kommer bitar av koden där jag implementerar denna funktion:

```
public P dequeue()
{
  P fore = null;

  if (this.fram != null)
  {
    Node nast = this.fram;
    this.fram = this.fram.nasta;

    if (this.bak == nast)
        this.bak = null;
        fore = nast.foremal
  }
  return fore;
 }
 public boolean empty()
```

```
{
  return (fram == null);
}
```

Denna implementation gör så att programmet håller koll på det element som kommer in sist i kön och tidskomplexiteten för att lägga till och ta bort element är även i detta fallet Ordo O(1), dvs konstant tid. Eftersom tiden slås ut över alla operationer så kallas det amorterad kostnad.

### Breath first traversal

I denna uppgift så ska vi använda oss av implementationer från ”Binary trees” och använda oss av ett kösystem när vi konstruerar vår iterator. Vi kommer att ha en nod som är den nästa noden i sekvensen men nu när vi ska hoppa till nästa nod så kommer vi att addera både den högra och den vänstra grenen till kön innan vi tar bort den nästa noden från kön (detta är förutsatt att den ej är null). Vi kommer att ha två olika typer av noder, den första typen utgör trädstrukturen och den andra typen utgör den länkade listan i kösystemet. Det är viktigt att noderna i kön refererar till noderna i trädstrukturen. Processen som sker kallas för ”Breath first” eftersom vi går över alla noder en gång innan vi går vidare till nästa nivå som är ett steg djupare. Jag har lärt mig att ”Breath first” är fördelaktigt att implementera när vi använder mer generella datastrukturer där till exempel vissa vägar är cirkulära. Detta är på grund av att en ”Depth first” strategi kan resultera i en oändlig loop tillskillnad från en ”Breath first” operation. Nedan kommer min kod för programmet jag har beskrivit ovan:

```
import java.util.*;
public class BreathfirsttraversalQueue
{
    public static void main(String args[])
    {
        // Skapar en tom prioritetkösystem
        PriorityQueue<Integer> ko = new PriorityQueue<Integer>();
        // Jag använder en add() metod för att addera element till kön
        ko.add(17);
        ko.add(20);
        ko.add(30);
        ko.add(50);
        ko.add(40);
        ko.add(15);
        ko.add(10);
        // Skriver ut prioritetkösystem
        System.out.println("Prioritetkosystem: " + ko);
        // Jag skapar iteratorn
```

```
        Iterator val = ko.iterator();
        // Skriver ut värderna efter att det har itererats genom kön
        System.out.println("Iterator värderna är följande: ");
        while (val.hasNext()) {
        System.out.println(val.next());
        }
    }
}
```

### An array implementation

Det går även självklart bra att implementera ett kösystem med hjälp av en array. För mig var det dock inte helt trivialt hur implementationen skulle se ut då det var rätt så många special fall. Innan jag började programmera så ritade jag upp en bild över hur datastrukturen skulle se ut och dessa är mina huvudsakliga operationer som jag vill utföra för att få ett fungerande kösystem med arrayer. Steg nummer ett är att kolla om kön är full. Steg nummer två är att lägga till elementet i slutet av arrayen (dvs om arrayen inte är full). Steg nummer tre är att kolla om kön är tom igen för att kunna avgöra om programmet ska skicka ut ett meddelande där den säger att arrayen är tom eller om den ska flytta alla element från index två till den sista med ett steg till höger för att sedan döpa den sista indexpositionen till noll då den är tom, och sedan förminska arrayen med 1 position då elementet längs fram i kön nu behandlas. Där efter implementerar jag några ”print” funktioner för det olika fallen och skapar en main där jag testar mitt program med värden där jag försöker att hamna i dessa specialfall. Då rapporten inte ska bli för lång så väljer jag ut det bitar från min kod som jag värderar som mest elementära till denna datastruktur:

```
// funktion som lägger till ett element på slutet av kön.
    static void LaggaTillElement(int data){
        // Kollar om kön är full eller inte
        if (kapacitet == sist){
            System.out.printf("\nQueue är full\n");
            return;
        }
        // Lägger till elementet längst bak i kön
        else{
            ko[sist] = data;
            sist++;
        }
        return;
    }
    // funktion för att ta radera element från positionen i kön längst fram
```

```
static void TaBortElement(){
    // if ko is empty
    if (forst == sist) {
        System.out.printf("\nQueue är tom\n");
        return;
    }
    // Shiftar alla elementen från index två till slutet av arrayen
    // till höger med en position
    else {
        for (int i = 0; i < sist - 1; i++) {
            ko[i] = ko[i + 1];
        }
        // Sparar värdet noll på slutet av arreyn, det är ej ett element
        if (sist < kapacitet)
            ko[sist] = 0;
        // tar bort den sista positionen i kön
        sist--;
    }
    return;
}
```

### dynamic queue

Till sist så kommer jag att förklara hur jag med hjälp av det dynamiska kösystemet kan hantera situationen då kön blir full. Implementationen för detta program består för det mesta av dessa steg. Steg ett är att skapa en ny array som är dubbelt så stor som den nuvarande. Steg två är att kopiera de befinnande värdena till den ny arrayen. Steg tre är att konvertera vår array till ett kösystem. Till sist så återställer vi det första och sista värdet i vår nya array. Nedan kommer mitt program då jag har skrivit dessa stegen som jag förklarar ovan i språket java:

```
private void utokakapacitet(){
    // Skapar en ny array som är dubbelt så stor som den nuvarande
    int nykapacitet = this.koArray.length * 2;
    int[] nyArray = new int[nykapacitet];
    // Kopiera värdena till den ny arrayen
    int temporarForst = forst;
    int i = -1;
        while(true){
        nyArray[++i] = this.koArray[temporarForst];
            temporarForst++;
        if(temporarForst == this.koArray.length){
            temporarForst = 0;
```

```
            }
            if(curStorlek == i+1){
                  break;
            }
      }
      // Konvertera våran array till ett kösystem
      this.koArray = nyArray;
      System.out.println("Nya kapaciteten för arrayen är : " + this.koArray.length);
      // Återställer det första och sista värdet i våran nya array
      this.sist = i;
      this.forst = 0;
}
```