Assignment 6 - Queues

Jesper Hesselgren

Oktober 1, 2024

Introduktion

Queues, eller köer, är ett vanligt förekommande fenomen i vardagen som vi ofta stöter på, från kassaköer till trafik. På samma sätt är köer en viktig datastruktur inom programmering, där de används för att hantera och bearbeta data i en specifik ordning, likt hur de fungerar i verkligheten. I den här rapporten kommer vi att undersöka datastrukturen närmare och implementera en enkel kö med hjälp av en länkad lista. Vi kommer även att analysera effektiviteten hos kön och genomföra en förbättring för att se hur det påverkar prestandan.

Köer

En kö som datastruktur representerar en samling objekt där nya objekt läggs till längst bak och tas bort längst fram, enligt principen FIFO (First In, First Out).

Den första kön vi har implementerat består av en nod "head" som representerar det första elementet i kön. Varje nod i sig innehåller data och en referens till nästa nod i kön. Det är dessa referenser som bygger våran köstruktur.

För att ta bort ett element från kön, använder vi metoden dequeue. Eftersom våran kö följer FIFO-principen (First In, First Out) tar vi alltid bort det första elementet. Detta görs genom att flytta pekaren "head" från det första elementet till dess referens. Fallet där kön skulle vara tom hanteras genom att returnera ett tomt värde null.

```
public Integer dequeue() {
      if (head == null) {
          return null;
      } else {
          Integer num = head.item;
```

```
            head = head.next;
            return num;
        }
    }
```

För att lägga till ett nytt objekt i kön måste vi traversera hela listan tills vi når det sista objektet och låta det referera till det nya objektet. Om kön är tom, hanteras detta genom att kontrollera om head refererar till null. Om så är fallet, sätts head till det nya objektet.

```
public void enqueue(Integer item) {
        // Om kön är tom
        if (head == null) {
            head = new Node(item, null);
        } else {
            Node ref = head;
            while (ref.next != null){
                ref = current.next;
            }
            ref.next = new Node(item, null);
        }
    }
```

Operationen att ta bort ett element från kön, som hanteras av metoden dequeue(), har en tidskomplexitet på O(1). Oavsett hur lång kön är, tar det alltid lika lång tid att koppla bort det första elementet. Däremot har operationen att lägga till ett objekt i kön, som hanteras av metoden enqueue(), en tidskomplexitet på O(n). Detta beror på att vi måste traversera hela listan, som består av n objekt, innan vi når det sista elementet och kan fästa det nya objektet.

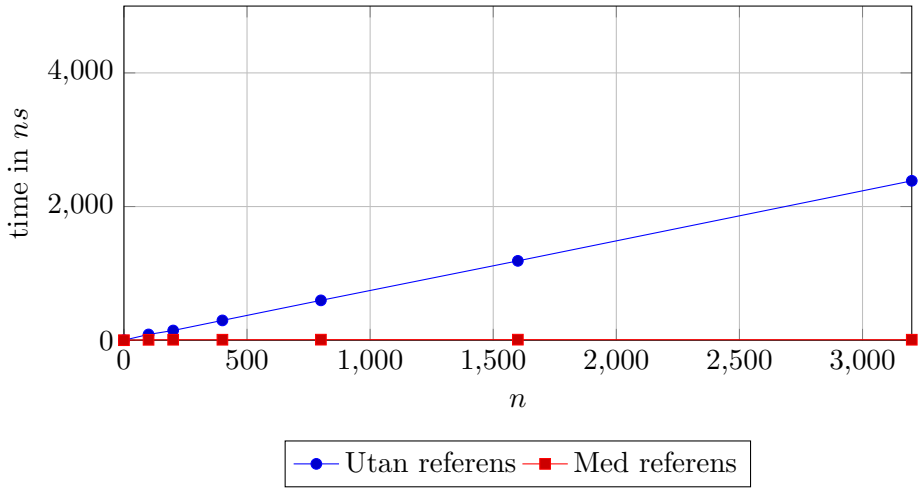
Genom att lägga till en egenskap "last" i vår ködatastruktur som pekar på det sista objektet i kön kan vi optimera operationen "enqueue" så att den får samma tidskomplexitet, O(1), som "dequeue". Istället för att behöva traversera hela kön för att hitta det sista objektet kan vi använda referensen som pekar på det sista objektet och låta det referera till det nya objektet som ska läggas till. Därefter uppdaterar vi "last"-referensen till att peka på det nya objektet.

```
    public void enqueue(Integer item) {
        // Om kön är tom
        if (head == null) {
            head = new Node(item, null);
            last = head;
        } else {
```

```
            Node ref = last;
            ref.next = new Node(item, null);
            last = ref.next;
        }
    }
```

Mätningar dequeue implementationer

I figur 1 och tabell 1 nedan visas data från mätningar av två olika implementationer av metoden "Enqueue". Utifrån figur 1 framgår det tydligt att implementationen utan en referens till det sista objektet växer linjärt, medan implementationen med en referens till det sista objektet har konstant tidskomplexitet.



Figur 1: Enqueue operationens effektivitet med referens till sista objektet och utan referens

Antalet element	**Utan referens**(ns)	**Med referens**(ns)
100	86	6
200	145	7
400	296	7
800	596	8
1600	1187	7
3200	2385	7

Tabell 1: Enqueue operationens mätningar vid det olika implementationer

Slutsats/diskussion

I denna rapport har vi undersökt den dynamiska datastrukturen “köer” och visat hur man enkelt kan implementera en kö med hjälp av en länkad lista. Vi undersökte även en optimering av vår “enqueue”-metod genom att lägga till en pekare som refererar till det sista elementet i listan. Detta resulterade i att tidskomplexiteten för vår “enqueue”-metod förbättrades från O(n) till O(1).

Köer är en struktur vi möter både i vardagen och inom datorvärlden, även om vi inte alltid är medvetna om det. De används t.ex flitigt i allt från operativsystemet till nätverksflöden. Vi har sett att en relativt liten optimering av vår datastruktur kan leda till stora förbättringar i effektivitet och prestanda, vilket är särskilt viktigt i situationer där hantering av stora datamängder kan förekomma.