Köer

Viktor Björkén

October 2022

1 Introduktion

I denna uppgift kommer vi att behadla köer, vi kommer med hjälp av olika metoder att studera och testa olika sätt att bygga dessa köer. Detta för att få en bild över hur vi på bästa sätt kan inplementera dessa köer.

Skillnaden mellan köer och andra datastrukturer så som stackar är att dessa precis som i verkliga köer går efter ett system som kallas, FIFO. Detta betyder då "first in, first out" vilket menas på som en "vanlig" kö att det element som varit i kön längst kommer att komma ut först. Jämförelsevis med en stack där det senaste element som läggs in kommer att vara det första som kommer ut.

De huvudsakliga metoderna vi kommer att anävda oss av när vi konstruera dessa köer kommer vara först och främst länkade listor, som är något vi lärt oss innan och bör vara relativt bekväma med. Vi kommer sedan att bygga vidare på detta och då använda oss utav arrayer istället. Detta kommer istället säkerligen vara lite kluriga.

När vi löst dessa olika implmetioner kommer vi att jämföra tidskomplexiteten mellan de för att se vilken som är den mest tidseffektiva.

2 Länkad lista

2.1 Endast en pekare

I denna första del är uppgiften att använda länkade listor för att bygga våran "kö". Vi kommer göra detta på ett liknande sätt till hur vi har jobbat med länkade listor förut.

Vi kommer att göra detta på 2 olika sätt vi kommer först att endast använda 1 pekar och sedan se hur skillnaden blir när vi istälet använder två, en i vid "huvudet" och en i slutet.

För att implementera denna struktur börjar vi att anväda det vi tidigare har lärt oss med noder och bygger våran kod. När vi ska addera element i våran kö, måste vi gå igenom hela listan från våran "huvud" pekare och sedan gå bakåt för att hitta "svansen"/slutet. När vi istället ska ta bort ett värde ur kön kommer vi redan ha våran pekare på första elementet som är det element som ska plockas ut.

Med denna kod samt förståelsen av den kan vi konstatera att tidskomplexiteten för att lägga till ett värde i kön blir O(n) eftersom vi måste gå igenom hela listan på n element. För att ta ur ett värde blir den istället O(1).

2.2 Dubbla pekare

När vi istället har 2 pekare, både vid ”huvudet” samt ”svanesen” kan vi stället göra på följande. Vi börjar med att initisera båda pekarna till null, därefter när vi ska stoppa in ett element där båda pekarna pekar på detta element (längs fram och bak). När vi sedan implementerar flera element gör vi liknande men låter ”svans” pekaren flytas till det nya elementet.

När vi ska ta ut det värde som ligger längst fram i kön gör vi följande. Eftersom vi har en pekare på det första element låter vi denna peka på head.next som sedan gör att vi pekar på värdet ”bakom” det som tidigare var ”huvudet”. Vi returnar sedan det som vi hade först och kollar så att inte listan har blivit tom.

Tidskomplexiteten för både att lägg till i listan samt ta bort ur listan med dubbla pekare kommer att våra O(1).

3 Bredd först sökning

I en Bred först sökning fungerar algoritmen anorlunda från den föregående uppgiften. istället för gå ner i varje gren från vänster till höger som i ”djup-först” sökning gå denna via tvären. Med detta menas att v börjar som tidigare i rooten, men sedan går ner i vänstra grenen och därefter går till nästa gren till höger fast på samma nivå. Detta kan illustrera med föjande bild.

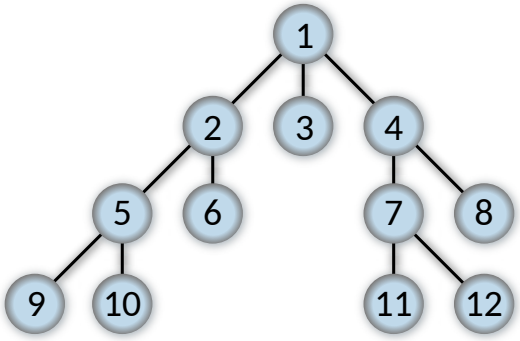

Figure 1: Bredd först sökning

Vi ska med denna information för att bygga en kö på liknande sätt som förra uppgiften ”binära sök träd”. Som i den tidigare uppgiften kommer våra vi använda en nod-uppbyggnad där vi kommer ha 2 typer av noder. En som är till för självaste trädet samt en som används för listan.

Skillnaden och anledningar till varför man istället skulle anväda sig utav en ”Bredd först Sökning” istället för en ”djup först sökning” beror lute på. En fördel med denna variant är att man förebygger att hamna i oändlliga loopar om det visar sig at 1 gren gå ihop med någon annan i en loop. En annnan fördel är att det i vissa situtioner kan vara enklare att hitta det man söker om det visar sig att någon gren innan är orimligt lång.

4 Kö implementation med array

För att använda en array till att göra en kö kommer vi först att behöva skapa en array med ett bestämt antal platser, och sedan ta in 2 värden, vårat huvud samt våran svans. För att skapa våra funktioner för att lägga till och ta bort element ur kön kan vi skriva en kod på följande vis.

För att lägga till ett värde så måste vi först kolla så att kön inte är full vet vi att det finns plats lägger vi elementet på sista platsen (svansen), och sedan ökar denna.

För att ta ut ett värde kollar vi först och främst att det finns ett värde att plocka. Därefter kan vi ta ut elementet som är på längst fram (huvudet), sedan flyttar vi hela arrayen för att på så sätt få fram det nästa element till huvudpositionen.

Detta kan se ut på följande sätt.

```
 // addQueue()
{
     if (storlek == tail) {
         System.out.printf("Queue is full");
         return;
     }

     else {
         queue[tail] = data;
         tail++;
     }
     return;
 }

//delQueue()
     if (head == tail) {
         System.out.printf("Queue is empty");
         return;
     }
     else {
         for (int i = 0; i < tail - 1; i++) {
             queue[i] = queue[i + 1];
         }
```

```
            if (tail < storlek)
                queue[tail] = 0;


            tail--;
        }
        return;
}
```

Denna implementation ger oss en tidskomlexitet på O(N).

5 vira runt

Lite problematik som vi måste ta hänsyn till är bland annat när till exempel k är lika med n, detta sker när våran kö blivit full. För att överkomma dessa problem kan vi bland annat skapa en kö som ”virar” runt sig själv, detta kan var aningen komplicerat att kunstruera men med hjälp av papper och penna kan man enklare visualisera hur detta skulle fungera.

För att lösa detta kan vi som ett exempel vara att vi ”virar runt” sig själv. Med detta menar jag att den kan visualliseras som en cirkel där slutet av arrayen sitter ihop i början. Detta kommer göra det möjligt att lägga till element efter vi redan fyllt upp den fast då återigen på plats noll. Det vi gör när vi implementera detta är att vi behöva öka k med ett varje gång vi adderar ett nytt element. Sedan sätta k till noll om då fallet att k är på sista positionen i arrayen.

En annan lösning är att öka storleken på arrayen, om vi sätter ett konstant värde kommer vi dock inte kunna med säkerhet säga att den inte kommer fyllas när fler och fler elements läggs till.

En dynamisk kö är ett exempel på en lösning för att förebygga att en kö blir full. Detta kommer vi disuktera i nästa stycke.

6 Dynamiska kön

I denna sista deluppgift kommer vi att hantera hur vi kan med tidigre kunskap av dynmiska stackar bygga ett dynamiskt kö system.

Detta kommer att göra det möjligt för oss att fylla våran kö utan att behöva tänka på storleken på arrayen. Detta då arryen i detta fall kommer att skapas en ny array som är dubbelt så stor. Sedan kopieras alla värden över från den tidigare arrayen till den nya större arrayen.

```
//Dynamisk kö
        int storlek = this.array.length*2;
        int[] nyArray = new int[storlek];
```

```
int tempHead = head;
int index = -1;
while(true){
    nyArray[++index] = this.array[tempHead];
    tempHead++;
    if(tempHead == this.array.length){
        tempHead = 0;
    }
    if(currentSize == index+1){
        break;
    }
}

this.array = nyArray;

this.head = 0;
this.tail = index;
```

7 Refernser

Figur 1 - jun 02, 22 https://en.wikipedia.org/wiki/Breadth-first$_s$earch