Assignment 6 - Queue with array

Jesper Hesselgren

Oktober 5, 2024

Introduktion

I denna rapport ska vi fortsätta undersöka datastrukturen köer. Om du inte har läst rapporten *Assignment 6 - Queues* rekommenderas det att göra det för en introduktion till vad en kö är som datastruktur och hur den kan implementeras med hjälp av en länkad lista. Istället för att implementera en kö med en länkad lista, ska vi nu undersöka en alternativ metod där kön implementeras med en array. Vi kommer att gå igenom hur implementationen ser ut och lyfta dess specialfall som behöver hanteras i `enqueue` och `dequeue` metoden.

Implementation

I denna implementation av en kö ska vi använda en array för att representera kön. I denna implementation kommer kön innehålla heltal, som exempelvis kan representera ID:n för objekten som står i kön. När vi implenterar en kö med en array krävs det mer än bara referenser till det första och sista objektet, som vi såg i implementation med en länkad lista.

Arrayen i implementationen sätts till typen Integer. Anledningen till att vi använder Integer istället för int är att vi vill kunna sätta en position till null när ett objekt tas bort från kön, vilket markerar att platsen är tom och kan återanvändas. Detta skulle inte vara möjligt med datatypen int, eftersom den inte kan innehålla null.

Utöver arrayen behöver vi några andra attribut för att vår kö ska fungera som önskat. Vi behöver en "räknare" som håller reda på hur många objekt som för närvarande finns i kön. Vi behöver också en "pekare" till det första objektet i kön samt en "pekare" till nästa lediga plats i arrayen. Dessa pekare gör det möjligt att effektivt både lägga till och ta bort objekt från kön.

Utöver detta har jag valt att inkludera ett attribut som håller reda på arrayens maximala storlek. Även om det inte är strikt nödvändigt, underlättar

det implementationen genom att ge en tydlig översikt över hur lång kön kan vara. Det gör det också enklare att hantera situationer då kön blir full, samt när pekarna når slutet av arrayen och måste "slå runt" till början igen, vilket är en del av wrap-around delen i implmentationen. Nedan kan vi se de attribut som används i implementationen:

```
Integer[] queue; // Array som lagrar köelementen
int size; // Köns maximala storlek
int n; // Håller koll på antalet objekt i kön
int point_first; // Pekar på det första objektet i kön
int point_last; // Pekar på första lediga position i kön
```

Enqueue() metoden

Enqueue-metoden används, precis som implementation med en länkad lista för att lägga till objekt i kön. Skillnaden med att använda en array istället för en länkad lista är att en länkad lista är dynamisk och kan växa efter behov utan begränsningar (så länge datorns minne tillåter det), vilket innebär att vi inte behöver oroa oss för utrymme för nya objekt. Med en array uppstår däremot problemet att arrayen har en fast storlek och inte automatiskt kan utökas. Därför måste vi kontrollera att det finns tillräckligt med utrymme i kön innan vi lägger till ett nytt objekt.

Ett ytterligare problem uppstår när `point_last` når den sista positionen i arrayen, men det fortfarande finns lediga platser på tidigare positioner som frigjorts genom att `dequeue`-metoden har tagit bort objekt från kön. För att lösa detta använder vi en teknik som kallas `wrap-around`, där vi låter `point_last` börja om från index 0. Denna `wrap-around`-teknik fungerar på så sätt att varje gång vi ska öka våra pekare, använder vi modulo-operationen. Istället för att bara öka pekaren med 1, ökar vi den med 1 och använder `mod size` för att se till att pekaren slår om när den når slutet av arrayen. På så sätt kan vi utnyttja det lediga utrymmet i arrayen och därmed effektivisera minneshanteringen i kön. När vi använder denna `wrap-around`-teknik innebär det att kön endast är full när `point_last` och `point_first` är lika och antalet objekt i kön är lika med den maximala storleken av arrayen.

Nedan kan vi se implementationen av `enqueue`-metoden och hur vi hanterar dessa special fall. Först kontrollerar vi om kön är full genom att anropa metoden `full()`, som avgör om `point_last == point_first` och `size == n`. Om dessa villkor är uppfyllda, är kön full och vi måste utöka dess kapacitet. Vi skapar då en ny array med dubbla storleken och kopierar över de befintliga elementen till den nya arrayen med hjälp av en `for`-loop. Därefter uppdaterar vi referensen så att den nya arrayen används som kö, och juste-

rar våra pekare `point_first` och `point_last` till rätt positioner.Slutligen utför vi det vanliga fallet där vi lägger till det nya värdet i kön, ökar pekaren `point_last` med hjälp av modulo-operationen (wrap-around tekniken), och uppdaterar antalet objekt i kön.

```
public void enqueue(int value) {

    if (full()) {
        System.out.println("Utökar kön");

        Integer[] largerQueue = new Integer[size * 2];

        for (int i = 0; i < n; i++) {
            largerQueue[i] = queue[(point_first + i) % size];
        }

        queue = largerQueue;
        size = largerQueue.length;
        point_first = 0;
        point_last = n;
    }

    queue[point_last] = value;
    point_last = (point_last + 1) % size;
    n++;
}
```

Dequeue()-metoden

Dequeue-metoden fungerar på samma sätt som i en implementation med en länkad lista för att ta bort objekt från kön. Precis som med `enqueue`-metoden vi precis har tittat på, finns det några specialfall som måste hanteras. Ett sådant fall är när vi tar bort det sista objektet från kön och kön därmed blir tom. Om detta fall inte hanteras korrekt, kan vi få en kö som bara växer och aldrig frigör minne som inte längre används. För att optimera minnesanvändningen och undvika onödig minnesförbrukning, bör vi därför minska storleken på kön när den blir tom.

Ett annat fall är när pekaren `point_first`, som pekar på det första objektet i kön, når slutet av arrayen, men det fortfarande finns fler objekt i kön som ligger i början av arrayen. Detta händer när nya objekt har lagts till efter att `point_last` har nått slutet och slagit omtill början av arrayen genom wrap-around-tekniken. Vi måste då hantera pekarna så att dessa objekt kan nås korrekt.

Nedan kan vi se implementationen av `dequeue`-metoden och hur vi hanterar dessa specialfall. Vi börjar med att kontrollera om kön är tom genom att anropa funktionen `empty()`, som undersöker om `point_last == point_first` och om antalet objekt i kön är noll (`n == 0`). Om så är fallet, är kön tom, och vi kontrollerar om arrayens storlek är större än 10 innan vi halverar dess storlek. Om arrayen redan är liten, finns det ingen anledning att minska dess storlek ytterligare. Denna gräns på 10 kan självklart justeras utifrån behov.Efter detta återställer vi pekarna och avslutar metoden genom att returnera tidigt.

Om kön inte är tom, utför vi standardfallet där vi tar bort det första objektet, nollställer dess plats i arrayen, minskar antalet objekt och uppdaterar pekaren `point_first` med hjälp av *wrap-around*-funktionaliteten. Denna funktionalitet hanterar även specialfallet där `point_first` når slutet av arrayen, men det fortfarande finns objekt kvar i kön i de tidigare positionerna i arrayen. I detta fall slår"`point_first` om och flyttas till början av arrayen, vilket gör att vi kan fortsätta hantera objekten korrekt.

```
public Integer dequeue() {

    if (empty()) {
        System.out.println("Kön är tom");

        if (size > 10) {
            Integer[] newQueue = halfArray(queue);
            queue = newQueue;
        }
        point_first = 0;
        point_last = 0;
        n = 0;

        return null;
    } else {
        int x = queue[point_first];
        queue[point_first] = null;
        n--;
        point_first = (point_first + 1) % size;
        return x;
    }
}
```