Uppgift 7 för betyg D

Simon Moradbakti

HT 2024

Introduktion

I denna uppgift var syftet att testa på ett nytt sätt att bygga en kö-algoritm. Alltså ett alternativt sätt än det sätt föregående uppgift behandlade, med länkad lista och användandet av enqueue och dequeue för att lägga till och ta bort i listan. Detta nya sätt är bättre, då vi använder oss av en array som använder sig av minne på ett mer sparsamt sätt. Alltså en så kallad *dynamisk* array. Detta gör hela algoritmen mer effektiv. Denna array kallas "Wrap Around" array och har fått sitt namn pågrund av hur den utför själva indexteringen, ritas denna upp ser det ut som en cirkulär array som pekar på sig själv (engelskans Wrap Around). Syftet med denna rapporten är att fungera som en guide för läsaren att förstå hur Wrap Around metoden fungerar.

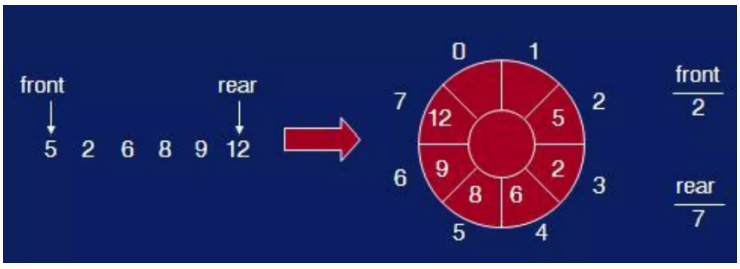

Figure 1: Illusterande bild av Wrap Around Array. OBS: ej min egen bild.

Ayyaswamy, Kathirvel. (2018). *UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES*. (Misrimal Navajee Munoth Jain Engineering College). Hämtad: 4/10-24. Länk: https://www.slideshare.net/slideshow/unit-ii-linear-data-structures-stacks-queues/10670748732

Wrap Around metoden som koncept!

Som man fick lära sig från föregående uppgift (Queues) så måste en kö alltid följa FIFO principen, dvs först in först ut. Något annat skulle per definition inte längre vara en kö. I den uppgiften användes länkad lista för att spara olika

element. Men med den mycket smartare Wrap Around metoden använder vi oss av en array för att spara element samt att vi slipper flytta dessa element.

Om man väljer att göra som i uppgiftbeskrivningen. Alltså att vi antar att vi har en array av en arbitär storlek n, samt att det första elementet finns på index 0 och det sista elementet på index x-1. Där x är den första tillgängliga positionen. Så vet vi sedan tidigare att vi måste flytta alla element om ett tas bort eller läggs till. Men om man istället väljer att hålla koll på front och backindexen, (i koden beskrivet som x och y) så kan vi välja att uppdatera indexen. Detta med hjälp av modulo operanden (i java beskrivet med procent notations symbolen). Detta är en viktig komponent för att Wrap Around ska fungera. Man kan initiera sin kö på detta sätt:

```
private T[] queue;
    private int x, y, n;
    private int size;
    public WrapAroundQueue(int capacity) {
        queue = (T[]) new Object[capacity];
        x = 0;
        y = 0;
        n = capacity;
        size = 0;
    }
```

Men det är bra mycket mer som måste finnas på plats för att detta ska fungera. Likt tidigare måste vi faktiskt ha enqueue och dequeue operationer, då detta faktiskt fortfarande är en kö. Dvs operander för att lägga till eller ta bort element.

Operander som läggs till

ett element läggs till får det position x och ökar x. Om x kommer till slutet av arrayen, alltså (n-1), återställs det till 0, med hjälp av den specifika kodraden x= (x+1) [procent symbol] n. Detta gör att vi faktiskt kan använda början av vår array flera gånger förutsatt att det finns ledigt utrymme.

Om ett element tas bort måste dess objekt returneras på position y och sätter det sedan till null. Detta för att spara minne. Vi ökar sedan y med 1 och återställer det till början med hjälp av kodraden y = (y+1)[procent symbol] n. Rent kodmessigt kan det se ut såhär:

```
    public void enqueue(T item) {
        if (size == n) {
            expandArray();
        }
        queue[x] = item;
        x = (x + 1) % n;
        size++;
    }
```

```
public T dequeue() {
    if (size == 0) {
        throw new IllegalStateException("Queue e
            tom :( ");
    }
    T item = queue[y];
    queue[y] = null;
    y = (y + 1) % n;
    size--;
    return item;
}
```

Förutom dessa måste vi också ha ett sätt att utöka vår array, då den faktiskt är dynamisk och på sätt kan välja själv när den ska växa eller krympa. Detta görs i praktiken genom att skapa en ny array som är dubbelt så stor som den föregående. Vi kopierar helt enkelt sedan över dessa till den nya med samma ordning av elementen. I praktiken kopierar vi över vår variabel och ser till att dom kopieras rätt in i vår nya array. Detta skulle kunna göras såhär:

```
private void expandArray() {
    int newCapacity = n * 2;
    T[] newQueue = (T[]) new Object[newCapacity];
    for (int z = 0; z < size; z++) {
        newQueue[z] = queue[(y + z) % n];
    }
    queue = newQueue;
    y = 0;
    x = size;
    n = newCapacity;
}
```

Specialfallen

Tyvärr är detta inte allt, utan man måste även ha specielfallen i beaktning. Dvs om kön är full eller tom. Vi kan enkelt skriva kod raden size==0. Denna gör helt enkelt att ifall man försöker ta bort ett element från en kö som är tom kommer det att kastats ett undantag för att hantera det. Vid full kö måste arrayen vara full alltså: size==n. Då utökar vi arrayen och kopierar över elementen till en ny större array. För Java gäller det att se till att använda sig av "New" när det ska vara med i koden. C har inte samma problem. När det kommer till självaste kontrollerna är det väldigt staright forward! Det räker egentligen bara att returnera vår size variabel:

```
// kontroll Queue tom?
public boolean isEmpty() {
```

```
        return size == 0;
    }
    // kontroll Queue full?
    public boolean isFull() {
        return size == n;
    }
    // hitta Queue storlek
    public int size() {
        return size;
    }
```

Sammanfattning

Den här rapporten hade som syfte att beskriva hur Wrap Around array metoden fungerar och beskriva den likt en guide. Rapporten har gått igenom dom olika delarna, så som enqueue, dequeue samt den dynamiska arrayen. Den har även gått igenom specialfallen och hur dom kontrolleras. En generell förklaring och genomgång av algoritmen har också utförts. Rapporten har även berört veckans parallela uppgift om köer och har jämförts med den för att man konkret ska se vad skillnaden mellan dom är.