Queues

Adalet Adiljan

Hösten 2023

Introduktion

Kö är en linjär datastruktur vars funktion och struktur efterliknar en situation vi stöter på dagligen - att stå i kö, som följer den enkla principen: ” Först in, Först ut”, ofta förkortat som FIFO. I det här rapporten fokuseras det på att implementera en kö med två metoder: en med hjälp av en array, och en annan med hjälp av en länkad lista. Vi kommer att avslöja hur båda implementationerna fungerar internt, belysa deras styrkor och kompromisser och ge en omfattande förståelse för köer i teorin och praktiken.

Metod

Länkad lista

Den linjära datastrukturen kan som enkelt implementeras med en länkad lista. I detta fall har strukturen enbart en egenskap, nämligen huvudet (head), och nya element läggs helt enkelt i änden av listan. Nackdelen med implementationen infinnes med remove-metoden som kräver att man traverserar hela listan för att hitta det nästkommande elementet. Tidsdkomplexiteten för metoden kommer att ligga på

$$O(n)$$

där n är antalet element i kön. Förslag till förbättring som i sin tur kan öka prestandan av både add- och remove-metoden vore om vi håller en pekare till det första elementet (head) och det sista elementet (tail) i kön. Det nya elementet läggs till direkt i slutet av listan där tail-pekaren refereras till, istället för att traversera igenom listan.

```
(:)
Node newNode = new Node(item, null);
      if(head==null) {
               head = newNode;
               tail = newNode;
```

```
} else {
        tail.next = newNode;
        tail = newNode;
        }
```

Givet i koden ovan kommer add- respektive remove-funktionen att bidra till en komplexitet på den konstanta tiden

$$O(1)$$

Svanspekaren bifogar effektivt det nya elementet som nästa nod och huvudpekaren på remove-metoden uppdateras varje gång till att peka på nästa nod i listan när den tar bort det första elementet, vilket också medför en konstant tid. Om kön är tom tilldelar vi både head- och tail till den nya noden. Detta utförs oberoende av köns storlek och kräver konstant tid. Om kön inte är tom, lägger vi till den nya noden i slutet av kön genom att ändra tail.next att peka på den nya noden och uppdaterar sedan tail till att vara den nya noden, vilket likaså är oberoende av köns storlek och tar konstant tid.

Med hjälp av vår enkla queue-implementation av en länkad lista kan detta användas till att effektivt traversera igenom vårt binära träd. Tillskillnad från depth-first verisonen följer breadth-first ”bredden” av trädet innan man går ”djupare” ner i trädets grenar. När vi använder oss av kö-strukturen lägger man till (enqueue) rotnoden i kön och fortsätter sedan lägga till alla dess förgrenade noder (barn) i kön, i ordningen de förekommer. Genom att ta bort (dequeue) noder från kön i den ordning de läggs till kommer man att utforska alla noder på en viss nivå innan man går vidare till nästa nivå. Denna typ av traversal kan vara av ett lämpligare alternativ för att implementera search-funktionen för ett obalanserad binärt trädet eftersom den garanterar att alla noder på nivå 1 utforskas innan man går djupare ner till nivå 2. Det specifika värdet som eftersöks hittas snabbare genom att hitta det närmaste förekomsten av värdet snarare än att gå ner i djupet först. Dessutom är denna strategi anpassad för eventuella circkulära vägar, där depth-first strategin kan resultera i en oändlig loop medan breadth-first avslutar sökningen direkt efter att den eftersökta elementet hittats.

Array

Man kan även implementera kö-strukturen med en enkel array-struktur. Skillnaden från en länkad lista ligger då i den underliggande datatypen som används för att lagra elementen. Medan en länkad lista dynamiskt allokerar minne för varje element och använder pekare för att koppla elementen tillsammans använder en array en fördefinierad storlek och allokerar minnesuttrymmet i förväg. Grundläggande strukturen består utav av att köstorleken får en fixerad längd givet n, där det första elementet i kön är på

index-positionen 0, och det sista elementet är på position k-1 (där k är den första lediga platsen). Då en länkad listas struktur bidrar till att vara mer dynamisk, är vår uppgift även att se till att array:en har den egenskapen.

Till vår generiska klass QueueArray har vi deklarerat en kö-array av objekt typ kallad Object[]arr, som används för att lagra ködata. Instansvariablerna size, least och peak används som hjälpvariabler till kön. Size används för att hålla reda på antalet element i kön, dess storlek. Index-pekaren list används för att hålla reda på det första elementet i kön, medan peak används för det sista. I vår konstruktor har vi sedan instansilerat var och en av dessa sina värden.

Dynamisk kö

```
public void enqueue(T item) {
    if(isFull()) {
            resize(capacity*2);
       }
            arr[peak] = item;
            peak = (peak+1)%capacity;
            size++;
    }
```

Dynamiska kön implementerades enligt följande: Vid en ny instans av QueueArray - klassen initieras den med en standardkapacitet på heltalsvärdet 10, och andra variabler kallad size, least och peak för att hantera kön. Innan ett nytt element läggs till, kontrollerar enqueue-metoden om kön är full. I sådana fall dubblar vi den nuvarande längden vi har med hjälp av resize-metoden.

```
private void resize (int newCapacity) {
            Object[] newArray = new Object[newCapacity];
            int j = 0;
            for(int i = least; i < least + size; i++) {
                    newArray[j++] = arr[i%capacity];
            }
            arr = newArray;
            least = 0;
            peak = size;
            capacity = newCapacity;
    }
```

Denna metod används för att ändra storleken på den interna arrayen som används för att hantera kön. Vi allokerar en ny array med den nya

storleken givet i argumentet, som ersätts till vår befintliga array. Lokala variabel-pekaren j används som räknare för att indexera den nya arrayen med hjälp av vår for-loop för att kopiera element från den befintliga- till den nya arrayen. Kopieringen sker i loopen, där loopens startpunkt motsvarar least (index för den första elementet i befintliga kön) och går uppt till least+size (index för det sista elementet i kön). När kopieringen är klar, återställs least-variabeln effektivt till 0 (eftersom den första positionen i den nya arrayen börjar från index 0). I samma fall uppdaterar vi sedan sista index-pekaren peak till size och sist till den nya längden av arrayen, för att reflektera längden (capacity) av kön.

```
public T dequeue() {
            (:)

            T item = (T) arr[least];
            arr[least] = null;
            least = (least + 1)%capacity;
            size--;

            if(size < capacity / 4 && capacity > CAPACITY) {
                    resize(capacity/2);
            }
            return item;
    }
```

Dequeue-metoden i den dynamiska versionen används för at ta bort och returnera element som finns längst fram i kön, vilket är den allra första elementet som lades till i början. Koden inleds med att kontrollera om kön är tom, annars meddelar den användaren att det inte finns något att ta bort. Därefter sparar den elementet från första positionen i variabeln Item, och sätter positionen till null. Efter att elementet har tagits bort uppdateras least-indexet för att peka på nästa element i kön. Då kön implementerar liknar en cirkulär struktur (wrap-around), bör det kontrolleras att indexet hålls inom kapaciteten med hjälp av modulo-operatorn. Slutlingsvis kontrollerar vi om storleken är mindre än 25 procent av den befintliga längden och om den är större än den ursprungliga standardkapaciteten. Om så är fallet, anropas resize för att allokera en ny array vars längd är till hälften.

Sammanfattning

Vi har nu i denna rapport fått tagit del av köer som en linjär datastruktur med fokus på implementeringar som följer FIFO-principen, vilket används i många vardagssituationer. Genom att tillämpa strukturen på två olika

sätt har vi fått en närmare insikt i kö-strukturen, vilket ger oss chansen att diskutera deras fördelar och kompromisser.

I länkade listans implementation diskuterar vi hur vi kan förbättra prestandan genom att använda huvud- och tailpekaren samt dess användning för att effektivt utforska binära träd med breadth-first strategin. Denna är alltså användbar för att hitta eftersökta noden i träd och är anpassad för att hantera cykliska strukturer.

Med en array utforskar vi dynamiskt ändringsbara köar och diskuterar hur vi kan optimera minnesanvändningen. Genom att ändra storleken på den interna arrayen kan vi hantera en växande mängd data och undvika att allokera onödigt med minnesuttrymme.