Heap och prioriteringkö

Viktor Björkén

October 2022

1 Introduktion

I denna rapport kommer vi att fortsätta på spåret kring olika köstrukturer. Vi kommer nu att utveckla det vi tidigare diskuterat och kommer bland annat implemnetera en heap samt prioriteringsköer. En heap och en prioritetskö är inte exakt samma sak men kan under vissa omständigheter bli samma.

Vi kommer att hantera hur olika implementationer av dessa prioriteringsköer skiller sig åt samt metoder som runt om kring dessa köer. Vi kommer som i tidigare rapport testa och gemföra olika implementationers tidskomplexitet för att avgöra vilken lösning som är minst kostsam.

2 Prioritets Kö

I denna första del ska vi implementera en prioritetskö med hjälp av en lista.

Vi kommer att börja med att implementera två olika metoder, en där vi addera in element i listan samt en där vi tar bort element. Dessa ska vi implementera på 2 olika vis. Vi kommer börja med att skapa en additions metod med O(1) och ”ta bort” metoden med O(n).

Det vi sedan gör är att byter roller där additionen istället är O(n), respektive O(1). När vi implementerar denna får vi följande graf för additions metoden med tidskomplexiteten O(n).

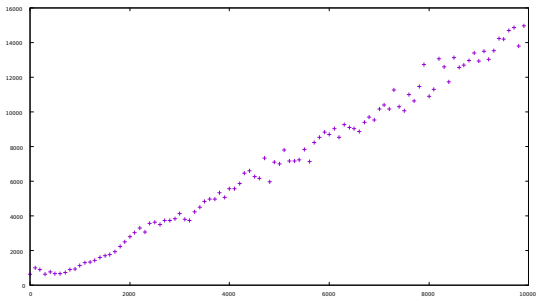

Figure 1: Add O(n)

3 Användning av träd

För att göra förbättringar kan vi istället använda oss av en trädstruktur. Detta kommer att göra det möjligt för oss att få en mindre kostlig tidskomplexitet än i den förregående deluppgiften.

Istället för en O(n) kan vi göra detta med O(log(n)), detta kommer att vara betydligt bättre speciellt desto större storlekar vi har. Metoden vi kommer att anväda oss utav för att göra detta kommer att fungera på följande.

För att göra om våran kod med denna gång en tidskomplexitet på O(log(n)) kommer vi behöva att hitta det minsta elemenet genom att kolla längst till vänster i vårat träd. Denna operation kommer då ha en tidskomplexitet på O(log(n)).

4 Heap

Det finns 2 olika sortes "heaps" en vars element i sitt träd blir större och större ju längre ifrån roten de kommer, detta kallas en "min-heap". Sedan finns också motsatsen som då är när roten är det största elementet i trädet där sedan alla grenar blir mindre och mindre.

I denna uppgift kommer vi att implmentera en heap genom att använda oss utav ett länkat binärt träd. Detta innebär som ordet binärt kommer i från att vi endast kommer att ha två barn som går vidare från en föreldär nod. Denna heap kommer att vara blanserad då med hjälp av denna länkaade binära träd implementation kan hålla kolla på antalet totala element i varje "sub-träd", detta gör att vi enklare kan addera in nästa element i den grenen med minst element för att därför göra trädet så balanserat som möjligt.

4.1 Addera

För att bygga denna metod för att addera element kommer vi att konstruera våran kod på följande sätt.

Vi kommer hålla reda på vilken våran nuvarande node är samt värde för att sedan jobba oss ner från rooten och jämföra för att hitta rätt. Om vårat nuvarande värde är mindre än på nuvärande noden så mycket vi plats på de. Vi kommer också hålla trädet balanserat som vi tidgare nämnt, detta gör vi genom att kolla om någon av antingne höger eller väntra noderna är tomma och isåfall lägger den där.

Denna implementaion kommer ge oss en tiddskomplexitet på O(log(n)).

4.2 Ta bort

När vi ska ta bort ett elementet från heapen finns det en hel del mer att tänka på och kan därför vara lite klurigare. Vi kommer börja med att göra liknande som additions metoder fast stället minska storleken med 1. Vi kan implementera detta med följande kod.

```
private Node remove() {
      if (this.left == null && this.right == null) {
        return null;
      } else if (this.left == null) {
          return this.right;
      } else if (this.right == null) {
          return this left;
      } else if (this.left.priority <  this.right.priority) {
          this.priority = this.left.priority;
          this.left = this.left.remove();
      } else {
          this.priority = this.right.priority;
          this.right = this.right.remove();
      }
      this.size -= 1;
      return this;
    }
```

Ett problem med detta är att det inte är säker att våran träd kommer att hållas balnserat när vi tar bort element på detta sätt.

5 Utöka ett värde i elementet

En annan vanlig operation vi kommer använda inom heapen är att pusha ner element längre ner i trädet. Detta kan vara användbart när vi vill utöka värdet i rooten och sedan pushar ner detta värde till någon av antingen högra eller vänstra grenen tills den hamnar på rätt plats.

Denna push funktion kommer att användas när vi söker efter ett värde med högst prioritet och sedan använda dess element. Det vi sedan gör är att vi ger denna en ny prioritet och sedan returnerar tillbaka i kön.

Anleding till varför vi behöver anväda våran push funktion i detta fall är pga att vi ökar värdet i noden när vi uför denna operation. Därför behöver vi då pusha ner till sin rätta plats.

6 Array implementation

När vi ska konstruera en heap med hjälp av en array kommer vi kunna se trädet i en array genom att roten representerar index 0. Därefter kommer index 1 och så vidare representeras av noderna per varje ”rad” uppifrån och ner. Där varje nod efter roten kan ses som postionen n, n * 2+1 samt n*2+2.

När vi implementerar en heap i en array på detta sätter kommer vi att beroende på trädets struktur få olika fyllda arrayer. När vi har flera ”blanka” grenar, till exempel en förälder nod med endast ett barn kommer vi få flela ”null” värden som fyller upp våran array, detta till skillnad från en väl fyllt träd där så gott som alla platser i arryen kommer ha olika värden.

Med hjälp av en heap kommer vi kunna konstuera en så fulländat träd som möjligt vilket är det vi vill ha när vi använder oss av array implemetationen. Om hela den nedersta raden är fylld med element för att vara fulländat träd för arrayen.

6.1 ”Bubbla”

När vi lägger till ett värde i våran array heap är det viktigt att det värde på elementet som adderas är större än dess förälder nod, detta då våran heap även kan kallas ”min-heap”. En min-heap är en heap där roten är det minsta värdet i trädet och där alla element under är större. Om vi då adderar in ett element vars värde är mindre än förelder noden kommer denna ej uppfylla kraven för en heap.

För att lösa detta kommer vi behöva ”bubbla” upp detta värde. Om värdet är mindre än förälder noden byter vi plats på de och därmed ”bubblar” upp elemenetet. Vi fortsätter denna process tills att element är på rätt plats.

Min bubbel funktion skriver jag på följande sätt.

```
public void add(int x) {
        if(size==maxSize){
            System.out.println("heapen är full");
            return;
        }
        size++;
        int index = size;
        heap[index] = x;
        bubbleUp(index);
    }

    public void bubbleUp(int pos) {
        int parentIndex = pos/2;
        int currentIndex = pos;
        while (currentIndex > 0 && heap[parentIndex] > heap[currentIndex]) {

            swap(currentIndex,parentIndex);
            currentIndex = parentIndex;
            parentIndex = parentIndex/2;
        }
    }
```

När vi adderar ett nytt värde och värdet är större än dens förälder nod är vi klara, men om den inte är det kallas bubbleUp() funktion tills den hamnar rätt. Detta kommer ge oss en tidskomplexitet på O(log(n)).

6.2 ”Sänka ner”

I denna deluppgift av array implementaionen kommer vi istället för att bubbla upp sänka ner ett element längre ner i trädet. Detta kommer vi använda ner vi istället för att addera ett element ta bort ett element. Det element vi kommer att returnera är roten, när denna försvinner byter vi ut den mot det sista elementet längt ner i trädet. Därefter kommer vi med hjälp av denna funktion att sänka det värdet till rätt plats.

Detta kommer fungera på liknande sätt som den tidigare additions metoden, fast denna gång kollar vi uppifrån och ner. Sedan jämför vi om ens nod är större än något av sina barn, om detta stämmer byter elementen plats. När det inte finns något element som är större kvar under är vi klara.

7 Skilland mellan array och länkad struktur

I detta stycke kommer vi att jämföra två olika metoder för att heap implementationer, dessa två är via en array samt ett träd. Vi kommer att göra en benchmark för dessa 2 implementationer med samma antal växande storlek (n) och sedan jämföra dess tider samt avgöra hurvida vi får olika tidskomplexiter eller ej.

storlek(n)	**array**	**träd**
160	58	75
320	76	92
640	102	140
1280	130	210

Det vi kan se är att array versionen kommer att vara snabbare än det binära träd implementationen, Detta då vi har en träd som är balanserat därmed kommer det vara enklare att byta runt värderna när vi använder oss utav våran array implementation.

Vi kan däremot få problem med arrayen när vi har för många tomma platser då vi har en konstant storlek på arrayen.

Det vi också ser på ovanstående tabell är att även om vi får bättre tid på våran array heap så får vi samma tidskomplexitet

$$O(log(n))$$