Heap eller prioritetskö

Hiba Qutbuddin Habib

Oktober 2022

Introduktion

Prioritetskö är en kö som används för att både lagra och hämta olika sorters data, det som skiljer en prioritetskö ifrån en vanligt kö är att elementerna i en piroritetskö plockas ut utifrån sin högre prioritet däremot i en vanlig kö plockas elementen som har vart längst i köen först. Prioritetskö algoritmen kan implmeneteras på olika sätt och med hjälp av flera andra datastrukuter såsom länkade lista, träd, samt vektorer. I denna uppgiften kommer vi att implementera prioritetskö på olika sätt syftet är då att kunna jämföra samt förstå nackdelar och fördelar med de olika implementationer.

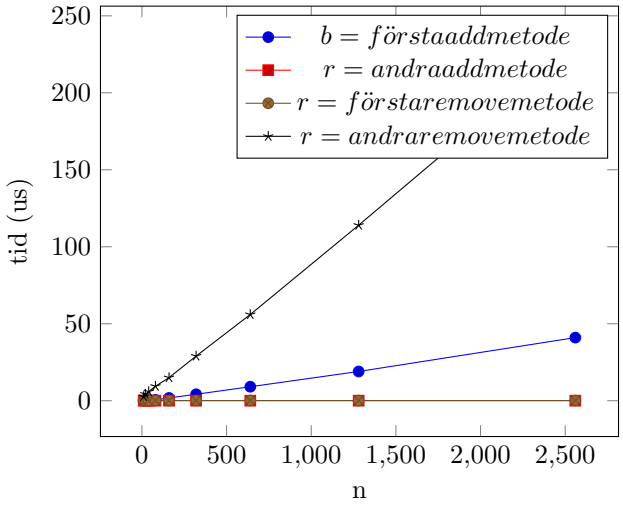
Länkade lista

En av de enklaste sätten för att implementera en kö är att använda en enkel länkade lista. Vid implementering kommer man att behöva två viktiga metoder som kallas för "enqueue" samt "dequeue". Dessa metoder har i uppgift att lägga till en elemenet i kö samt tar bort en element från kö. I denna delen kommer elementerna med minst värde att prioriteras först. De två metoderna kan implementeras på två olika sätt, i den ena sättet sorteras elementerna i kö utifrån sin prioritet när man adderar de i kö och då att ta bort en element kommer att vara enkelt eftersom elementen med högst prioritet kommer alltid att ligga i den första noden. I denna andra sättet kommer "dequeue" metoden att göra jobbet när man ska ta bort en element, då elementerna kommer alltid att läggas till vid första noden och när man ska ta bort en element så kommer man behöva jämför för att sedan hitta elementet med högst prioritet och tar bort den.

Resultat

Utifrån grafen nedan kan vi se skillnaden mellan de två olika implementeringar. I den förta sättet så skulle data sorteras utifrån prioritering när vi adderar det till kö, denna operation är då linjär alltså tillhör mängden$O(n)$ efter varje gång någon element ska läggas till i en kö så behöver den jämförs med n andra elementen for att hamna på sin rätta position. Att tar bort en element från kö i

detta fall är enkel, då den första elemnetet i kö är alltid med högst prioritering, vilket gör att denna operationen är konstant $O(1)$. I den andra sättet så har vi det tvärtom, elementerna läggs alltid till i första noden, vilket ger en konstant operationen, däremot at ta bort en element kommer att vara en linjär operation, då man behöver söka bland n element efter den element med högst prioritet för att sedan tar bort den.



Heap

När vi implementerar en kö som en träd datastrukturer så kallas det för heap. I en heap så innehåller rooten alltid den elementet med högst prioritet samt både de högra och vänstra grenen kommer att ha heap struktur. För att vi ska kunna få effektiva resultat när det gäller tidskostand så vill man helst ha en balanserad träd. Ett sät och hålla trädet balanserad är genom att alltid hålla redan på antal element i varje gren samt de nya elementerna som ska läggas till i trädet bör infogas i de grenar med minst antal element.

Även i denna implementering så är ”add” samt ”remove” är de två viktigaste metoderna. Att addera en element i trädet kommer att vara enklare än att ta bort, då om vi vill lägga till element så först kontrolleras rooten, ifall vi inte ha någon root så behöver vi skapar en helt ny nod annars kontrolleras rooten , då elementet i den noden jämförs med elemeenten som ska läggas till, om elementen är mindre så byter de plats. I annat fall kontrolleras både högra och vänstra grenar, i den grenen det är tomt så adderas den nya elementet sedan uppdateras den nuvarande noden till antigen den västra eller höga noden. Metoden ”add” kommer att användas rekursivt så varje gång en ny element adderas kommer samtliga steg ovan att upprepas.

Att ta bort en element från trädet är lite mer komplicerat, då man behöver ta hänsyn till samtliga fall. Om ingen nod finns så finns det inte heller något element som man kan ta bort. Om bara rooten finns i trädet så innebär det att

det är den sista elementet i trädet som man kan ta bort. Annars om vi har en träd med root och flera noder då kan vi göra enligt följande:

```
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;
```

"Benchmark"

Efter att vi har implementerat trädet, så kommer vi behöva studera hur bra balanserad är den. Det kan vi göra genom att kunna ha en viss statistik om trädets djup. Man kan ta redan på djupet på trädet om vi har en metod där den lägger till ett element i trädet, metoden kommer att returnera hur djupt i trädet behövde den gå för att kunna addera till elementet. Först kommer vi att addera 64 slumpmässiga elementer till trädet sedan kommer vi att push slumpmässiga värde till trädet, sedan returnerar denaa metoden djupt på trädet. I slutet kommer vi att optimera "add" metoden så att den också kan returnerar djupt på trädet, en viss statisk kommer ockås att samlas för "add" metoden.

Resultat

Incr	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Djup (push)	3	4	4	4	4	4	4	4	4	5	4
Djup(add)	5	5	6	6	6	6	6	6	6	6	6

Table 1: Djup på träd

I tabellen ovan kan vi se statistiken över djupt på trädet som vi fick både från metoden "add" samt "push" då vi kommer att pusha element från början i rättposition i trädet beroende på djupet så kommer de elementerna som vi adderar att hamna på olika djup nivår i trädet beroende på deras prioritering. Däremot när vi adderade elementen istället så kommer de alltid att placeras i slutet av trädet där vi har en ledig höger allternativ vänster gren, detta eftersom vi vill ha en väl balanserad träd. Sedan kommer denna elementen att klättras upp i en högre nivå i trädet beroende på priotritering. Så metoden "add" returnerar hur djupt behövde den gå för att lägga till elementen i den första lediga högre alternativ vänstra gren.

Vektor

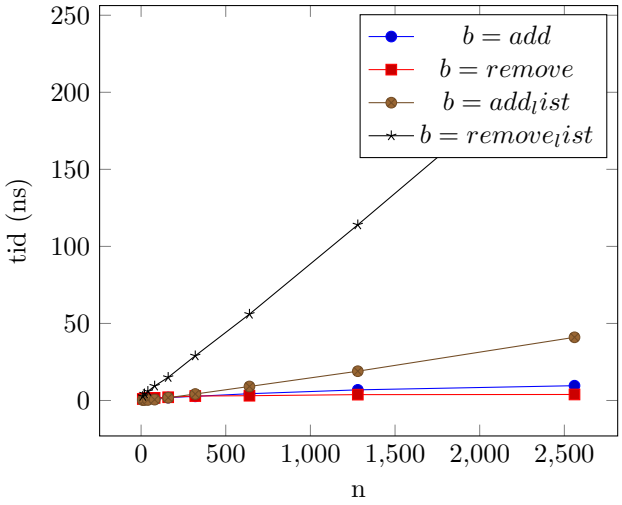
Heapen ovan kan implememteras på ett annat sätt, nämligen som en vektor. För att kunna lägga trädet ovan i en vektor så kommer vi behöva organisera det på ett sätt så att alla förälder nod samt deras barn hamnar rätt. Alltså för en nod n så kommer den högra och vänstra grenar att hamna på index $n*2+1$ respektiv $n*2+2$.

Om vi ska lägga till en element så kommer elementen att placeras i sista lediga position i vektor, sen för att elementerna ska hamna rätt som träder så kommer vi att jämföra varje element med sin förälder om elementen är mindre än sin förrälder så byter de plats. Samtliga element kommer att ha sin förälder vid index$n-1/2$ respektivt $n-2/2$.

Om man ska radera en element så kommer man först att ta bort elementet vid den första index då den är root element sedan placeras den sista element vid första index i vektor, denna elementet jämförs med sin barn(alltså högra och vänstra grenar) om det finns något barn som är mindre som byter dessa två element plats.

Uppgiften nu är att mäta tiden för både "add" samt "remove" metoden. Resultat skall sedan jämföras med den första resultat vi fick med Länkade lista.

Resultat



I grafen ovan ser vi skillnaden mellan "add" samt "remove" metoder för både implementering av prioritetskö med vektorer samt med länkadelista. Utifrån resultat i grafen kan vi se att implementering av en prioritetskö med vektorer är mycket effektivare än med länkade lista, om vi då jämför mätningar med de icke konstant resultat vi fick vid mätningar för länkde lista. Detta eftersom när vi ska addera ett element i vektor så vi placera elementet i slutet av vektor vilket är då en konstant operation sedan kommer vi jämför denna element med sin förälder alltså vi kommer ej behöva jämföra den med n element då vi skippar

flera index däremellan tills vi hämnar på förälder noden, denna operationen bör var logaritmisk om vi jämför men balanserad träd då höjden är $log(n)$. Detsamma gäller remove operationen, då med vektor implementering får vi en mycket bättre remove operation om vi jämför med den icke konstant remove operation för implementering med länkade lista, detta då vi tar bort alltid den första element i vektor vilket är en konstant operation, sedan flyttas den sista elementet i vektor till den första indexet och jämförs med sin barn, då varje nod har två barn så kommer man nog inte behöva jämföra med alla n element i vektor,alltså att ta bort ett element ur prioritetskö som är implementerat med vektorer är logaritmisk, alltså den bör tillhör mängden $O(log(n)$.