Prioritetskö

Mattias Sandberg

13 Okt 2022

Introduktion

Heap kallas för en prioriterings kö där kön ej baseras på i vilken ordning elementen kom in i kön utan någon annan faktor. Detta betyder att element med högre prioritet kommer att vara närmare köns framkant och när ett element tas bort från listan så borde det alltid vara det elementet med högst prioritet. Det förekommer ofta heaps i formen av trädstrukturer och i denna uppgift ska vi implementera en heap på olika sätt och bestämma för och nackdelarna för samtliga implementationer. För en heap datastruktur så finns det två generella varianter där en kallas för ”min heap” och prioriterar det lägre värdarna och lägger det lägsta värdet som trädens rot, och en som kallas för ”max heap” där det högsta värdarna prioriteras.

En lista av element

Det lättaste sättet att implementera en prioritets kö är att använda vanliga listor. I detta program så implementerar jag en prioritets kö som innehåller heltal där lägre nummer har högre prioritet än höga nummer. I den första implementationen ska ”add” funktionen ha tidskomplexiteten O(1) och ”remove” funktionen O(n) där n är antalet element i kön. I den andra implementationen så ska vi konstruera programmet så att ”add” funktionen är kostsam tidsmässigt men ”remove” funktionen är O(1). Nedan kommer bitar av koden för programmet som har en ”add (push)” funktion som har tidskomplexiteten O(n) och en ”remove” funktion som har tidskomplexiteten O(1). I specialfallet där listans huvud har lägre prioritet än den nya noden så lägger jag till en ny nod före huvudnoden och byter huvudnodens position. Detta tyckte jag var ganska komplicerat att implementera och därmed visas lösningen nedan:

```
if ((huvud).prioritet > k){
    // Lägg till ny nod före huvudet
    temporart.nasta = huvud;
    (huvud) = temporart;
```

```
    }
    else{
    // Gå igenom listan och hitta en position för att lägga till en ny nod
        while (start.nasta != null && start.nasta.prioritet < k){
            start = start.nasta;
        }
     // Antingen i slutet av listan eller på önskad position
        temporart.nasta = start.nasta;
        start.nasta = temporart;
    }
    return huvud;
}
```

Efter benchmarking så såg jag att båda operationer hade en tidskomplexitet på O(n) men att beroende på om man vill ha en snabb exekveringstid då programmet primärt poppar eller pushar element så kan man implementera dessa två olika lösningar som jag både har beskrivit och visat ovan.

Ett väldigt speciellt träd

Eftersom O(n) är en tidskomplexitet som vi gärna vill undkomma så är lösningen ett träd då operationer i träd oftas kan göras på O(log(n)) vilket även gäller för implementationen av prioriterade köer. Att lägga till ett nytt element i denna datastruktur tar O(Log(n)) vilket är överlag bra men vi kan utveckla detta och få en ännu bättre tidskomplexitet och det ska jag beskriva nu.

En heap är en trädstruktur där rotens nod håller elementet med den högsta prioriteringen och där både den högra och vänstra grenen är heaps. Den största fördelen med en heap är att vi kan finna det minsta elementet med en tidskomplexitet på O(1). Och om vi vill ta bort något element från kön så är tidskomplexiteten fortfarande O(log(n)) vilket betyder att denna implementation är extremt gynnsam. När jag skrev programmet och skulle implementera en ”add” funktion var jag tvungen att kolla om det nya värdet skulle byta ut rotens nod eller om det skulle tryckas ned längre i trädets grenstruktur. Att ta bort ett element från prioriteringskön var trivialt då det värdet man skulle ta bort alltid låg i trädets rot då det var prioriterat. När rotens värde togs bort så blev det element som hade högst prioritering uppflyttat till trädets rot så att det är näst på tur att bli borttaget/behandlat. Därefter så sker det som en kedjeeffekt att alla element som låg under det uppflyttade elementet följer efter så att trädstrukturen behåller sina egenskaper.

Ett problem som jag stötte på var att trädstrukturen lätt kunde bli obalanserad, dvs att den högra och vänstra grenen växer/förminskas olika snabbt och därmed skapas en datastruktur som liknar en länkad lista. För att få bukt på detta så behövde jag hålla räkningen på det två olika grenstrukturerna så att jag visste vart jag skulle addera de nya elementen. Detta går bra att implementera för en "add" funktion men för "remove" funktionen så finns det inget att göra eftersom rotens element tas bort och resten av elementen följer efter och sidan av trädstrukturen beror på vilket element som har högst prioritering.

När jag implementerade min "add" funktion för att lägga till element så följde jag stegen som en riktlinje som jag ska förklara nu och resultatet blev önskvärt. Det enda "special" fallet som jag skrev var att om roten är noll så ska elementet läggas direkt till en ny nod i heapen. Först och främst så utökar vi trädet med en storlek, sedan kollar vi om det nuvarande värdet är mindre än det värdet för den nuvarande noden och sedan byter vi plats på värdena. Sedan så håller programmet koll på om den högra eller vänstra grenstrukturen är tom och den som har flest element returnerar den sitt värde till. Till sist så sätter jag den nuvarande noden till antingen höger eller vänster om roten och låter algoritmen utföra dessa operationer igen.

Att ta bort ett element är något mer komplext då det finns flera fall att ta hänsyn till men nedan kommer det triviala som jag använde för att få koden att fungera korrekt. Först kollar programmet om någon av grenarna är tomma. Sedan jämför den värdena och tar det önskvärda värdet och sätter det som ny rot och minskar den delen av grenen där ett värde togs ifrån. När vi tar bort element från trädstrukturen så är det omöjligt att manipulera resultatet av trädets struktur och därmed så kan det hända att vi får ett något obalanserat träd. För att hålla prioritetskön rätt prioriterad så är det viktigt kunna flytta runt element beroende på dess prioritet. Det visar sig att det är kostsammare att flytta element upp och ned i grenstrukturen generellt sätt än att plocka bort ett element och stoppa in det på sin nya plats. Där med implementerar jag funktionen push(Integer incr) för att utgöra den väsentliga operationen. Nedan kommer ett kort kodstycke från heapen där jag implementerar en funktion som shiftar noden uppåt om det behövs för att bibehålla en balanserad heapstruktur:

```
static void shiftUpp(int p){
```

```
  while(p > 0 && L[foralder(p)] < L[p]){

    // Byt plats på förälder och nuvarande nod
    swap(foralder(p), p);

    // Uppdatera (p) till förälder för (p)
    p = foralder(p);
  }
}
```

Efter att ha utför benchmarks när jag adderar 64 element med slumpmässiga värden till heapen och kört en sekvens med push operationer där jag utökar datastrukturen med ett slumpmässigt värde får jag att tidskomplexiteten är Ordo O(log(n)) vilket är synnerligen mindre kostsamt för stora dataset än vad Ordo O(n) är vilket betyder att mina implementationer har varit fördelaktiga.

Array implementation

Även vanliga arrays visar sig vara väldigt lämpade för att implementera en heap. Tricket som jag kommer använda är att representera ett binärt träd i en array. Noden vid positionen n kommer att ha sin högra och vänstra gren vid n*2+2 och n*2+1. Trädets rot kommer alltid att ha position 0 så den högra grenen kommer att ha position 2 ochden vänstra 1. Vidare så är det viktigt att förklara att noden vid position 1 kommer att ha sin vänstra gren som position 3 och sin högra som position 4. Beroende på hur många element som fyller upp trädstrukturen så kan den antingen bli glest eller tätt. Ett komplett träd är ett träd där samtliga sektioner av trädet är fyllt skilt från löven och i detta scenario så är en array implementation optimal.

Att ta bort ett element är ganska trivialt, det kallas att vi låter värdet sjunka ned till sin rätta position. Värdet som vi ska returnera är värdet på rotens nod och vi ersätter det med nästa värde i arrayen. För att programmet ska agera som en prioritets kö (heap) så måste vi låta det elementen sjunka i arrayen och byta plats på element som har högre prioritet och tillslut så har vi tagit bort ett element och köns struktur och prioritet har återupptagits. Nedan kommer en kodsektion från prioriterad kö med Array implementation där funktionen bygger en max-heap från arrayen:

```
    static void byggaHeap(int array[], int P){
        // Index för sista löv som inte är en nod
        int startIndex = (P / 2) - 1;
```

```
        for (int k = startIndex; k >= 0; k-){
            gorheap(array, P, k);
        }
    }
```

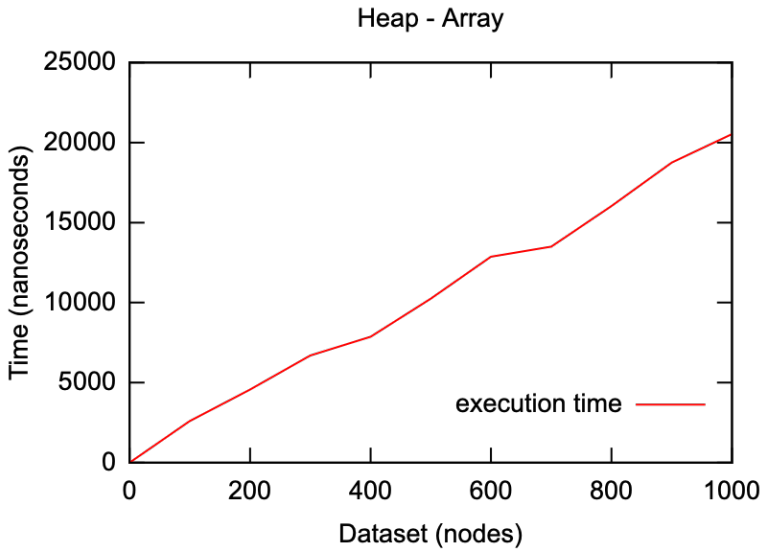

Figure 1: Graf över Heap - Array implementation

Enligt grafen ovan när jag benchmarkade programmet så visar det sig att tidskomplexiteten är Ordo O(n) vilket är samma tidskomplexitet som för linked list vilket betyder att för väldigt stora dataset så är det båda implementationerna i princip lika effektiva men det varierar mer för mindre dataset och båda implementationerna har för och nackdelar.

```
/*
                        30
                      /    \
                    60      80
                   /  \     / \
                  70   90  160 130
                 / \   / \
                120 110 180 200
*/
```