Kö

Hiba Qutbuddin Habib

Spring Fall 2022

Introduktion

Kö är en linjär datastruktur som i grunden kommer från vår dagliga liv, då i väldigt många situationer behöver vi följa kö strukturen under vardagarna. Det som kännteckna en kö är ”FIFO” principen. FIFO innebär ”först in först ut”, alltså den som står först i kö är den som går först ut ur köen. Denna principen används som grund för kö algoritmen. I denna rapport implementeras kö algoritmen med hjälp av två andra linjär datastrukturer, länkade lista samt vektorer.

Länkade lista kö

Att använda en länkade lista för att implementera en kö är en av de enklaste metoderna. Som grund behöver man en vanlig nod klass samt en kö klass som anhåller två pekare nämligen ”först samt sist” samt tre viktiga metoder som kallas ”isEmpty(kontrollerar om köen är tom), enqueue (lägger till element i kö) och dequeue (tar bort element ur kö)”.

Implementering av en kö med länkade lista kan ske på två olika sätt, det som skiljer dessa två sätt mest åt är metoderna ”dequeue” då i den ena metoden så behöver man springa till den första elementet för att radera den, medan i den andra metoden så utnyttjas en pekare som pekar direkt åt den först elementet och då kan vi enkelt radera den utan att behöva springa genom hela listan. Implementering för de både metoderna sker enligt:

*Första ”dequeue” metoden:

```
public Integer dequeue() {
    if(isEmpty()){
System.out.println("\n the quene is empty, nothin to delete :(:(");
        System.exit(-1);}
    node prev = null;
    node current = front;
    while (current.next != null){
        prev = current;
        current = current.next; }
```

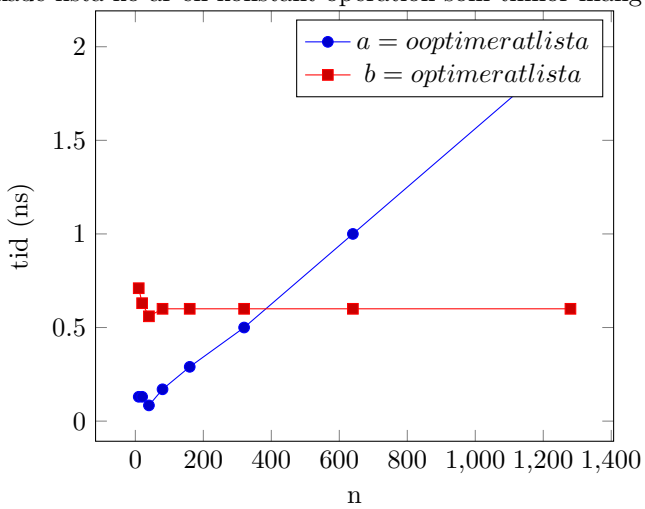
```
    if (prev == null)
        front=null;
        else prev.next = null;
        return current.item; }
```

*Andra ”dequeue” metoden:

```
public Integer dequeue() {
  if(isEmpty()){
  System.out.println("\n the quene is empty, nothing to delete :(, :(");
     System.exit(-1); }
  Node temp= front;
  System.out.println(" Removing \n" + temp.item);
  front = front.next;
  if(isEmpty()) {
    last = null; }
    length -=1;
   return temp.item;}
```

Resultat

Att lägga till en element i kö är alltid en konstant operation, då vi adderar alltid en element till slutet av kö, eftersom vi har en pekare som pekare åt den sista elementen i kö så kommer vi inte behöva gå genom hela lista, den enda man behöver göra om vi ska lägga till en element är att uppdatera den sista refrensen, vilket är en konstant operation. Alltså att lägga till en element i en länkade lista kö är en konstant operation som tillhör mängden $O(1)$.



Resultatet i grafen ovan visar skillnaden mellan de både ”dequeue” operationen. Den blå linjen visar resultat för den första metoden och då ser vi tydligt att vi får en linjär komplexitet, vilket är rimligt eftersom man behövde gå genom hela listan tills man kommer till första element som ska tas bort, det

innebär att man behöver gå genom n element och detta ger en komplexitet som tillhör mängden $O(n)$. Den röda linje visar resultatet över den andra metoden, från grafen ser man att det är en konstant komplexitet, då i detta fall behövde man endast uppdatera den sista refrensen, vilket ger en komplexitet som tillhör mängden $O(1)$. Alltså utifrån denna resultatet kan man konstatera att den andra metoden är mycket effektivare.

Bredd-först-sökning

I en tidigare uppgift tillämpade vi djup-först-sökning för en binärt söknings träd, då vi använde en stack och traversering av trädet börjar djupt ner i trädet, noderna pushas in fram till rooten, stack principen "LIFO" sist in först ut tillämpas när man återingen popar ut alla pushade noder.

I denna uppgiften tillämpar vi bredd-först-sökning på samma binärt sökningsträd, bredd-först-sökning innebär att vi besöker noderna i trädet från vänster till höger, uppifrån och ned, alltså i olika nivår. För denna sökningen används kö datastruktur då principen "FIFO" för en kö tillämpas alltså eftersom rooten pushas in i kö först så kommer den att vara den första som också pushas ut från kö. Denna typen av traversering är bland annat bra om man vill ha information om djup och nivår i en träd.

Vektor implementering

I denna delen används först en vanlig vektor sen en cirkulär vektor för att implementera en kö. Det som kommer skilja dessa två metoder åt är främst "dequeue" samt "enqueue" metoderna.

Vanlig vektor

Här används en vanlig vektor struktur, där vektor har en begränsat längd, när vi lägger till en element så uppdateras pekaren som pekare mot den sista element. Om ett element ska tas bort så det enda man behöver hålla koll på är pekare som pekar mot första elemneten. Om vektor är full, då är det inte möjligt och lägga till element.

Cirkulär vektor

I denna delen kommer vi att använda den vanliga vektor som grund då vi kommer att försöka optimera både metoderna " enqueue" samt "dequeue". För metoden "enqueue" kommer vi nu att optimera den så om vi har raderat elementerna i början fram till index i och det finns fria luckor i vektor mellan $0(i-1)$ så kan man utnyttja dessa luckor genom att använda räknaoperationen modulos, då vi lägger alltid dem nya element i slutet, därför ska modulos operation tillämpas på pekaren som pekar mot sista elementen, detta implementeras enligt;

```
public void enqueue (T value) {
    if (isFull()) {
    System.out.println("Queue is full :( :(");
    }
    if( front == -1)
        front = 0;
/*om last pekar mot sista index i vektor så t.ex 2
om vektor har längd 3, så får man (2+1) % 3
det innebär alltså att den nya element ska ligga vid första position i vektor*/
        last = (last + 1) % size;
        arr[last] =  value;
 System.out.printf("Inserting %d\n", value);  }
```

När det gäller metoden ”dequeue” så kommer räknaoperationen modulos att tillämpas på samma sätt men denna gången ligger fokus på pekaren som pekar mot den första element, då vi raderar alltid den första elementet i en kö, detta kan implementeras enligt;

```
public  Object dequeue () {
    if (isEmpty()){
    System.out.println("\n the quene is empty, nothing to delete :(, :(");
        System.exit(-1);}
    Object value = arr[front];
        if (front == last) {
        front = last = -1;}
    else
// front pekare uppdateras på samma sätt som last pekare ovan
        front = (front + 1) % size;
        return (value); }
```

Att implementera en kö med cirkulär vektor är mer effektivare är att använda vanlig vektor, då med en cirkulär vektor har man möjligheten att återanvända det tömma uttrymmet i vektor, däremot en nackdel med en cirkulär vektor är om ma har återanvänt den tömma uttrymmet men inget element tas bort ifrån vektor, då har man inte möjligheten och tilläga nya element eftersom vektor är full, denna problem kan man lösa bland annat genom att använda dynamisk vektor.

Dynamisk vektor

Fördelen med en dynamiska vektor är att den kan expandera vid brist av utrymme., vilket gör det lättare för oss att inte tänka särskilt mycket på denna bristen, då varje gång vektor är full så reserveras en plats för dubbel så stort vektor, data kopieras över och nu har man tillgång till mer uttrymme, detta upprepas varje gång vektor blir full. Jag implementerar detta genom att skapa en metod som jag kallar för ”expand”, då den här i uppgift att skapa ännu större

vektor för att sedan kopiera över data från den gamla vektor. Denna metoden anropas sen av metoden ”enqueue”.

*Metoden ”expand”:

```
public void expand (){
   Object[] newarr= new Object[size*2];
   for ( int i=0; i< size; i++){
       if(isFull()){
       newarr[i]= arr[i]; } }
   arr=newarr;
   size=2*size; }
```

* Metod anrop:

```
public void enqueue (T value) {
    if (isFull()) {
       expand();}
```

Sammanfattning

Kö är en användbart datastruktur, detta eftersom en kö kan hantera flera typer av data samt det är en snabb och flexibel datastruktur då man kan lätt komma åt både ändarna. I denna uppgiften såg vi att man kan implementera en kö på flera olika sätt, där en länkade lista kö va en av de enklast sättet, varje sätt har också sin fördel och nackdel. Denna uppfiten har hjälpt mig bland annat med att kunna förstå vad är en kö datastruktur, hur fungerar den och hur kan vi implementera den. Man fick också en uppfattning om när är det lämpligt och använda denna datastuktur, då nämnde vi bredd-först-sökning med den är även en viktig grundsten för program med mer än en körande tråd. Denna rapport ger en inblick av vad en kö datastruktur är men köteoriet i sig är mycket djupare ämne än så.