Dubbel länkade lista

Hiba Qutbuddin Habib

Spring fall 2022

Introduktion

I en föregånde uppgift arbetade vi med enkelt länkade lista, där vi fick en bättre förståelse om egenskaper för en länkade lista. I denna uppgiften fortsätter vi på samma spår och då kommer vi attt arbeta med double länkade lista. Syftet är att förstå dess egenskaper samt skillnaden i jämförelse med en enkelt länkade lista.

"Benchmarks"

Funktionalitet för en dubbel samt enkelt länkade lista är samma, däremot vissa funktionalitet är mer effektiva för en double länkade lista.

För att kunna få en bättre förståelse över detta, så kommer vi att behöva implementera "Benchmarks" som kan användes med både datasturkturer. I denna benchmarks kommer vi att använda slumpmässiga indexer för att välja vilka noder vi ska ta bort från listan, sedan adderar vi igen samma nod till listan.

Detta kan implementeras enligt:

```
doubleLinkedLista dll = new doubleLinkedLista();
    int[] sequence = new int[k];
//skapa en vektor med random index
    for (int i = 0; i < k; i++) {
        sequence[i] = rnd.nextInt(n);
        System.out.print(sequence[i],";")}
//Fylla på lista med random värde
    for ( int i= 0; i<n; i++){
        int random_data= rnd.nextInt(100);
        dll.addFront(random_data);
        }
    for (int i= 1; i<k; i++) {
//använd slumpmässiga indea
    int index = sequence[i];
//spaea data vid den nod som ska radera
```

```
    int s=dll.findNodeAt(index);
//radera data vid den givna index
    dll.removeAt(index);
//addera igen samma data som raderades i början av listan
    dll.addFront(s);
    }
```

Dubbel Länkade Lista

En dubbel länkade Lista har samma funktionalitet som en enkel länkade lista. Skillnaden är att i en dubbel länkade lista så har varje nod också refrensen till det föregående noden, detta innebär att i en dubbel länkade lista kan man flytta sig åt två riktningar. På grund av denna extra refrensen kommer man behöva ta hänsyn til det när man använder vissa metoder, då nu om man raderar en nod eller addera en så behöver man uppdatera två refrenser. Metoderna kommer i stort sätt att likna metoder för en enkelt länkade lista med bara en liten uppdatering. Metoderna addera i början av listan samt radera vid en specifikt position i en dubbel länkade lista kan implementeras enligt följande:

Lägga till en element till början av Listan

```
 public void addFront(int data) {
    if (head == null)
       head = new nod(null, data, null);
     else {
        nod newNod = new nod(null, data, head);
//i denna metoden behövde vi uppdatera även denna refrensen
        head.previous = newNod;
        head = newNod;
        }
    size++; }
```

Radera vid en specifik position i listan

```
public void removeAt(int index){
//om head är null då har vi inget och radera
    if (head == null)return;
//om givna index uppfyller ej dessa villkor då kan vi inte förflytta oss mer i listan
    if (index<0 || index >size)return;
    nod actual= head;
    int i = 1;
    while (i< index){
//tills vi kommer till den givna
//index som behöver flytta oss framåt i listan
        actual = actual.next;
```

```
        i++;}
//om nästa pekare  är null
    if( actual.next == null){
//uppdatera pekare
        actual.previous.next = null; }
//om förra pekare är null
    else if(actual.previous == null){
//uppadera både pekare samt head elementen
        actual = actual.next;
        actual.previous = null;
        head = actual;}
    else{
//om noden ligger i mitten av listan,
//så uppdatera både pekare enligt nedan:
        actual.previous.next = actual.next;
        actual.next.previous = actual.previous; }
//uppdatera storlek av listan
        size--; }
```

Presentation av resultatet

n(längd)	**10**	**20**	**40**	**80**	**160**	**320**	**640**
t(us)	0.015	0.026	0.052	0.097	0.2	0.35	0.76
t/n	1.6	1.35	1.33	1.2	1.2	1.2	1.2

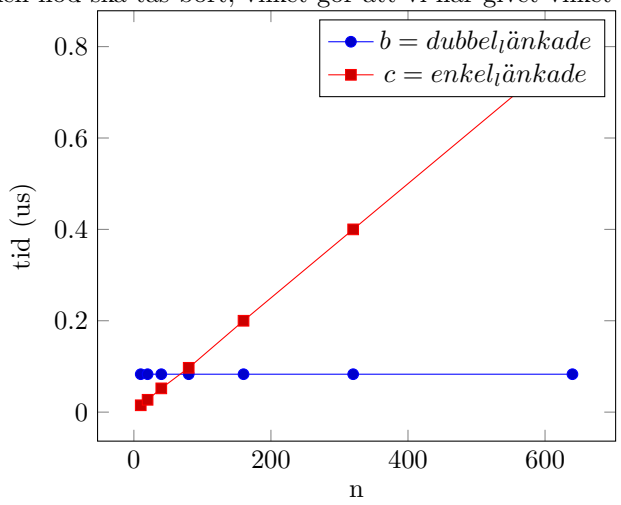
Table 1: Tidmätning för en enkel länkade lista

n(längd)	**10**	**20**	**40**	**80**	**160**	**320**	**640**
t(us)	0.083	0.083	0.083	0.083	0.083	0.083	0.083
t/n	0.008	0.004	0.002	0.001	0.001	0.0002	0.0001

Table 2: Tidmätning för en dubbel länkade lista

I tabellen ovan kan vi se resultatet över tidmätningar för de både länkade listor. I den andra tabellen kan man tydligt se att en dubbel länkade lista ha en konstant tid, då tiden är samma oberoende av listans storlek samt konstanten ”k”(Om vi tänker på den linjära funktionen $t = k * n + m$)går mot noll. Detta innebär att en dubbel länkade lista ha en komplexitet som tillhör mängden $O(1)$ när det radering av noder. Detta eftersom för en dubbel länkade lista kan man hitta en specifik nod utan att behöva traversera hela länkade listan, då den extra refrensen som en dubbel länkade lista har till skillnaden från en enkel länkade lista gör det mycket effektivare att komma åt noderna då samma nod kan nås från både håll.

I den ”benchmarks” som vi använder lägger vi bara till elementera till början av listan, detta operationen i sig är konstant, då när vi lägger element i början av listan så det enda man behöver göra att uppdatera refrenser vilket är en konstant operation, för denna anledning kommer denna operationen att tillhör mängden $O(1)$. När det gäller att radera en element från listan, så blir den en konstant samt mycket effektivare operation om pekare till noden som ska radera är given. Och det är det vi gör i ”Benchmarks” då vi skapar en sekvens med slumpmässiga värde som index, dessa index används sedan som information om vilken nod ska tas bort, vilket gör att vi har givet vilket nod ska raderas.



I grafen ovan kan vi se förhållnadet mellan en dubbel och enkel länkade lista. Utifrån den första tabellen samt grafen ovan kan vi se att en enkel länkade lista har en linjär algoritm, alltså den tillhör mängden $O(n)$. Då det är samma ”benchmarks” som tillämpas för både enkel och dubbel länkade lista, vilket innebär at även för en enekl länkade lista adderas elementen i början av listan, vilket är i sig en konstant operation, däremot när vi raderar noder med den slumpmässiga index i en enkelt länkade lista behöver man traversera listan från början tills man kommer till den givna pekaren för noden, detta innebär att i en genomsnittlig fall behöver man traversera n element, vilket ger att denna algoritm som tillhör mängden $O(n)$.