Länkade listor

Hiba Qutbuddin Habib

Spring Fall 2022

Introduktion

I de föregående uppgifter arbetade vi med de primitiva datastrukturer samt vektorer, där lärde vi oss att reservera en viss utrymme i minnet för att sedan placera saker efter varandra. I denna rapportenn kommer vi att jobba med mer komplicerade data strukturer, där saker är länkade till varandra med refrenser. Vi studerar olika enkla länkade strukturer såsom: Enkelt länkad lista samt stack (byggde på en länkade lista), då syftet är att ta reda på deras egenskaper.

Enkelt länkade lista

Uppgift 1

En enkel länkade lista är en länkade lista, där alla noder pekar åt en enda håll, från första till sista noden. Man kan beskriva länkade lista som en linjär följd av noder som länkas ihop med en pekare.

Vår uppgift är att försöka förstå egenskaper för en länkade lista, detta kommer vi göra med hjälp av två länkade listor som vi kallar för 'a' respektive 'b'. Dessa två länkade listor sätts ihop, då i en första fall kommer vi att variera storleken på listan 'a' sedan sätta ihop den med 'b' som har en fixerat storlek. I den andra fallet kommer vi att låta 'a' ha en fixerat storlek medans 'b' som kommer att ha en varierande storlek samt sätter vi ihop både listorna. Detta implementeras enligt:

```
public static void main(String[] args) {
    Random rnd = new Random();
    LinkedList a = new LinkedList();
    LinkedList b= new LinkedList();
    double min = Double.POSITIVE_INFINITY;
    for( int n=0; n<100; n++) {
    //För att ändra storleken på a som ändrar vi 'i' i loopen
        for (int i = 0; i < 10; i++) {
            a.add(new Node(rnd.nextInt(10)));}
        for (int j = 0; j < 10; i++) {
            b.add(new Node(rnd.nextInt(100)));} }
```

```
 long t0 = System.nanoTime();
 a.append(b);
 long t1 = System.nanoTime()-t0;
 if (t1< min)
     min = t1;
 ls.display();
double t= (min/100)/1000;
double t2=((min/100)/1000)/a.length;}}
```

Resultat

Tiden mätningar ger följande resultat:

n(längd)	**10**	**20**	**40**	**80**	**160**	**320**	**640**
t(us)	0.081	0.16	0.29	0.58	1.13	2.26	5.1
t/n	0.008	0.007	0.007	0.007	0.007	0.007	0.007

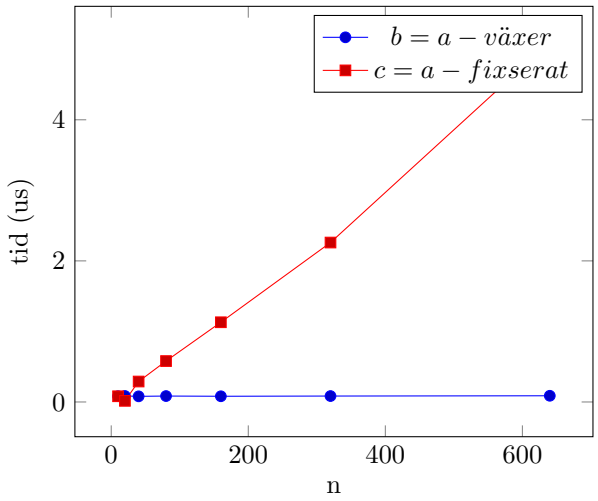
Table 1:

n(längd)	**10**	**20**	**40**	**80**	**160**	**320**	**640**
t(us)	0.082	0.085	0.08	0.084	0.081	0.084	0.088
t/n(a)	0.0082	0.0081	0.008	0.0084	0.0081	0.0084	0.0089

Table 2:

I första tabllen ovan kan vi se att värdet av $t/n(a)$ är konstatnt, detta innebär att vi har en konstant som när vi multiplicera den med längden (n) så kan vi få tiden (t), detta i sin tur betyder att denna algoritmen är en linjär algoritm som då tillhör mängden $O(n)$.Det kan bero på att i vårt första fall så ökar vi längden på listan 'a' och detta innebär att när vi ska sätta ihop 'a' och 'b' så kommer man behöva gå genom (n) antal element i första lista för att hitta sista pekaren som pekar till null, då den pekare kommer att ändra referens och kommer att börja peka till första element i listan 'b'.

I den andra tabellen ser vi att värdet (t) är konstant, detta tyder att algortimen är konstant, vilket innebär att den tillhör mängden $O(1)$. Att denna algoritmen blir konstant detta eftersom storleken på den första länkade listan 'a' är fixserade, vilket innebär att varje gång man letar efter pekare som pekar mot null så söks det genom samma längden. Även om längden på listan 'b' växer så det påverkar inte tiden, då den enda lista vi söker genom är 'a' och när pekare som pekar mot null hittas kommer endast referensen av den att ändras till början av första listan.



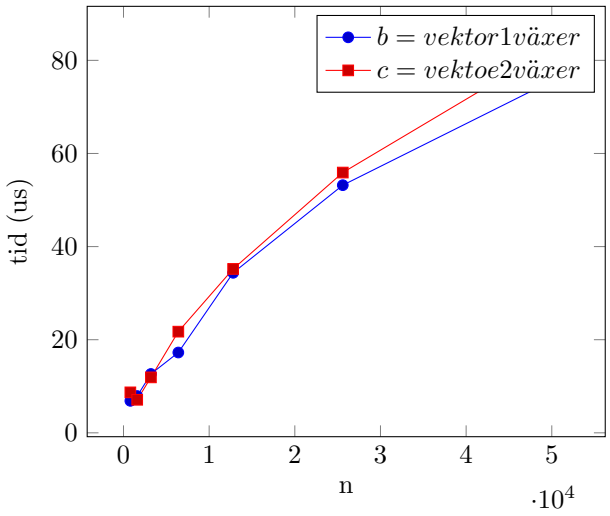
I grafen ovan kan vi tydligt se att när listan 'a' växer så är algoritmen linjär medans när listan 'b' växer så är algoritmen konstant.

Uppgift 2

I denna uppgiften kommer vi att använda vektorer istället för en länkade lista. Då om vi har två vektorer och vi ska sätta ihop de så kommer vår metod att först alokera en plats åt en ny vektor som är lika stor som de både två första verktorer, för att sedan kopiera värderna för de både vektorer till den nya vektor. För att kunna förstå hur detta fungerar så kommer vi att mäta tid vid två olika fall. Vid den första fallet kommer vi låta den första vektor växa i storlek medans den andra kommer att ha en fixerat storlek. I den andra fallet kommer vi att göra motsatsen.

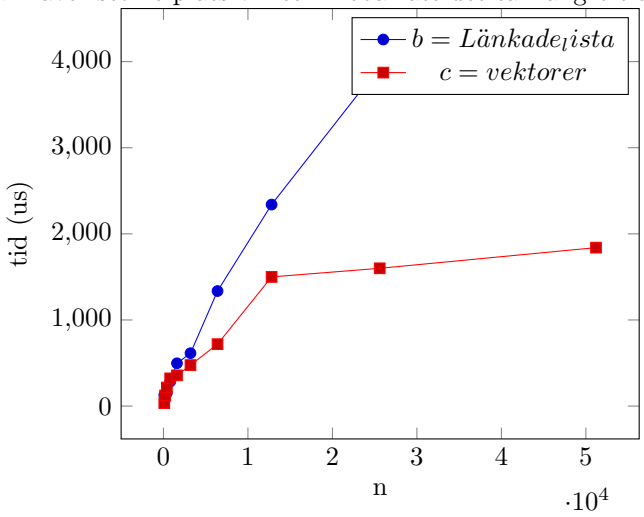
Resultat

Utifrån grafen ovan kan vi se att algoritmen är linjär. Resultatet verkar vara samma och oberoende av vilken vektor som växer samt vilken vektor som är fixserat, då i de både fall kommer vi att behöva alokera plats för en ny vektor för att sedan kopiera i den n-antal elemenet och för denna anledning tillhör denna algoritmen mängden $O(n)$.



Tid kostnad för alokering

För att skapa en vektor eller en länkade lista så behöver vi alltid allokera en plats för det. Detta tar olika lång tid beroende på vad vi vill skapa. I grafen nedan ser man att det tar längre tid att skapa en länkade list jämfört med en vektor. Detta kan bero på att när vi skapar en vektor så kommer vi behöva allokera en plats där vi ska lägga in data, medans för en länkade lista så behöver vi allokera en plats både för data samt för referenser, detta leder till att länkade lista kräver större plats vilket innebär att det tar längre tid att skapa.



Stack

I en tidigare uppgift skapade vi en dynamisk stack och då efter att vi genomförde tid analys kom vi fram till att den dynamiska stacken har en tidskomplexitet som tillhör mängden $O(nlog(n))$. Detta eftersom när den dynamiska stacken blir full så behövde vi expandera den, vilket gjorde att vi behövde allokera en större plats för att sedan kopiera allt till den nya stacken, detta gjorde att det krävdes mer tid och i en genomsnit så tillhörde den dynamiska stacken mängden $O(nlog(n))$. När vi implemterar en stack byggd på en länkade lista istället för vektorer så expandations metoden blir mycket enklare, för i detta fall är vi inte i behöv av att allokera en större plats för att skapa en ny vektor och kopiera allt, istället kommer vi behöva endast flytta på pekare när vi adderar data. "Push" samt "Pop" operationer i detta fall kommer att vara konstanta, detta eftersom vi kommer alltid att "popa" den sista pushade element, vilket innebär att vi kommer inte behöva söka efter en specifik element bland (n) antal element. Alttså när vi implementera en stack byggd på en länkande lista så kommer algoritmen att ha en komplexitet som tillhör mängden $O(1)$.