Länkad lista

Adalet Adiljan

Hösten 2023

Introduktion

På de föregående uppgifterna har vi arbetat med grundläggande datastrukturer och arrayer. Vi har i denna uppgift fått lära oss i att hantera en komplexare variant av datastruktur som är bättre beskrivna som länkade strukturer, där komponenter är sammanlänkade med varandra genom referenser, ofta kallade pekare eller länkar. I vår metod-avsnitt diskuterar vi hur vi har löst uppgiften. I vår resultatdel jämför vi länkande listor med arrayer och undersöker hur ”append”-operationen skiljer sig mellan dem. Vi kommer i detta avsnitt att även analysera kostanden i minnesallokeringar, skillnaderna av en array- respektive länk baserad implementation av en stack.

Länkad lista

En länkad lista är den enklaste länkande strukturen. Den innehåller en sekvens med celler men har bara tillgång till den första cellen i sekvensen. Varje cell i sekvensen har egenskaper och en referens till nästa cell (ibland kallad ”next” men i detta fall är kallad tail). Om tail-variabeln har initierats till null, är denna en nullpekare som indikerar att vara den sista cellen i listan.

Metod

Följande kod ger oss ett exempel på en implementation av den länkade klassen:

```
class LinkedList {
Cell first;
Cell last;

  class Cell {
      int head;
      Cell tail;
```

```
    Cell(int val, Cell t1){
        head = val;
        tail = t1;
        }
    }
```

LinkedList är vår huvudklass för den länkade listan. Den innehåller två medlemmar first och last, som representerar den första och sista cellen i listan och används för att förflytta sig mellan cellerna och utföra operationer på dem. ”Cell” är vår inre klass som representerar varje cell i listan. Varje cell har två medlemmar: ”head” som är ett heltal och ”tail” som är en referens till nästa cell i listan. Denna snitt är alltså en grundläggande struktur som vi sedan använder för att bygga vidare andra implementationer och operationer. En särskild operation som har implementerats och används för vår analys är append-funktionen. Denna metod har i syfte att lägga till en länkad lista representerad av ”LinkedList b” i slutet av den aktuella länkade listan enligt följande:

```
public void append(LinkedList b) {
  Cell nxt = this.first;
  Cell prv = null;
  while(nxt.tail!=null){
      prv = nxt;
      nxt = nxt.tail;
  }
  nxt.tail = b.first;
  b.first = null;
  }
```

Vi initierar en variabel ”nxt” till att peka på den första, aktuella cellen i listan med hjälp av ”this” som refererar till den aktuella instansen av LinkedList. ”prv = null” initieras alltså till null, vilket används för att hålla en referens till den föregående cellen när vi itererar framåt i listan. Villkoret i vår while-loop används för att interera genom den aktuella listan tills sista cellen har nåtts och ändrar sedan ”tail”-referensen till den sista cellen för att peka på den första cell i den andra listan. När vi nu har en gemensam lista, uppdaterar vi referensen av den sista cellen i den nya listan till att peka på ”null”.

Benchmark

Metoden för prestandamätningen ska ge oss en uppfattning om den totala körtiden för append-operationen. Detta gör vi genom att variera storleken på den första länkade listan (lista A) och lägga till den i en annan länkad lista (lista B) med fast storlek. Då vi var enbart intresserade av att se hur

körtiden förändras i förhållande till den stegvis ökande array-storleken, blev vår slutliga metod följande:

```
public static void main(String[] args){
int[] cellSizes = {100, 500, 1000, 2500, 5000, 7500, 10000, 15000, 20000, 30000};
for (int n : cellSizes) {
      LinkedList listA = new LinkedList(n);
      LinkedList listB = new LinkedList(2000);
      long startTime = System.nanoTime();
      listB.append(listA);
      long endTime = System.nanoTime();
      double totalTime = (endTime - startTime)/1e6;
}
```

I denna kodexempel hade vi en fixerad storlek på lista B och låta lista A variera sig i sin storlek enligt de olika storlekarna i vår ”cellSize” array, och sedan anropa append-funktionen till att låta lista A läggas till på lista B. Man skulle även byta ordningen och låta lista B varieras i sin storlek medan lista A hade en fixerad storlek (vilket vi fastställer med ett heltal som vår argument) och låta lista B läggas till på lista A.

Resultat

På tabell 1 ser vi resultaten på append-operationen för den länkade listan A respektive B i de båda fall som vi har varierat storleken på den länkande listan. Första kolumnen är de olika storlekarna som vi har varierat lista A respketive B med. På andra kolumnen kallad lista A visar oss tidsmätningarna när vi har varierat storleken på A samtidigt som vi har haft B’s storlek konstant. I den tredje kolumnen ser vi resultaten på körtiden för det fall som vi har varierat storleken på lista B och låtit lista A vara konstant. Vi kan även dra slutsatsen att tidsåtgången för de respektive fall ökar linjärt med storleken på listan, dvs.

$$O(n)$$

där ”n” är storleken på listan för den lista som vi har varierat storleken på i den fallet ifråga.

I tabell 2 har vi i första kolumnen olika array storlekar som vi lät array 1 och array 2 variera med i två olika fall. I det första fallet lät vi array 1 vara i en varierande storlek samtidigt som array 2 var konstant. Resultaten i detta fall syns på kolumn 2. I vårt andra fall skulle vi byta ordningen och låta array 1 vara i en konstant storlek medan vi varierade storleken på array 2, vilket motsvarar tidsmätningarna på kolumn 3. Vi kan enligt tabellen se att det verkar som att tidskomplexiteten lutar sig till att vara mellan en linjär eller konstant. Vi kan konstatera att det är en konstant tidskomplexitet, med följande iakktagelser: I de båda fall kan vi se att körtiden är relativt

storlek	**list A**	**list B**
100	0.004	0.005
500	0.007	0.009
1000	0.03	0.03
2500	0.03	0.04
5000	0.06	0.08
7500	0.09	0.12
10000	0.12	0.17
15000	0.18	0.24
20000	0.25	0.33
30000	1.0	0.37

Table 1: Tidsmätningarna för de olika storlekarna på den länkade listan för lista A respektive för lista B, där tien är mätt i millisekunder.

size	**array 1**	**array 2**	**list(n)**
1000	0,006	0.007	0.17
1500	0,004	0.004	0.12
3000	0,005	0.005	0.13
3500	0,006	0.006	0.13
4000	0,007	0.006	0.15
4500	0,008	0.007	0.19
5500	0,008	0.008	0.21
6000	0,006	0.007	0.31
8000	0,01	0.01	0.29

Table 2: Tabell 2 är en sammanställning på resultaten av tidsmätningarna array 1 och 2 i från de respektive fall som vi ändrade storleken på dem med mätt i millisekunder.

konstant och ökar inte dramatiskt oavsett hur mycket vi ökar med arrayens storlek. Skillnadena för de respektives fall är små. Vi märker även av att körtiden är inte direkt proportionellt mot storleken på arrayerna, vilket tyder på en konstant tidskomplexitet.

Vår fjärde kolumn motsvarar resultaten på tidsmätningarna när vi låter vår länkade lista växas enligt varje storlek i kolumn 1. När vi nu ska jämföra kostnaden för att bygga en lista med tiden det tar att allokera en array av en given storlek, kan vi se enligt tabellen att tidskomplexiteten för den länkade listan är linjär. Allokeringen av en array av samma storlek förväntas ha konstant tidskomplexitet eftersom att vi reserverar fixerad storlek på minne. Jämför vi dessa två, förväntar vi oss därmed att allokeringen av en array ska vara snabbare än att konstruera en länkad lista av samma storlek (i det

fall vi bygger på med större och större storlekar).

Stack

Implementationen av ett stack kan göras både med en array och en länkad lista. I en array-baserad stack allokeras en array med en bestämd storlek för att lagra stack-elementen. När vi behöver ändra på storleken för att öka den, är det minnesoptimalt att dubbla dess storlek, vilket är dyrt i början då vi behöver skapa en ny array och kopiera alla av de element från det föregående arrayen till en ny och sedan avreferera den gamla arrayen till att istället referera till den nya.

I en länk-baserad stack har vi en lista där varje nod lagrar ett element och en referens till nästa nod (eller null om vi har nått slutet av listan). Tiden det tar att lägga till ett element på listan är konstant med tidskomplexiteten

$$O(n)$$

Beroende på stackens storlek kan exekveringstiden indikera bättre prestanda hos array implementationen, särskilt när dess storlek är relativt liten. Då storleken blir större måste storleken på arrayen tillslut att ändras vilket har en betydande inverkan på exekveringstiden. Däremot påvisar den länkade listan bra prestanda i generellt, då den kan hantera både dynamiska och stora datamängder. Sammanfattningsvis påvisar array-implementationen en bättre prestanda i små stack-storlekar medan länkade listan visar bra prestanda i stora datamängder.