Assignment 3 - Länkade listor

Jesper Hesselgren

September 25, 2024

Introduktion

Tidigare i kursen har vi arbetat med den primitiva datastrukturen arrayer. I denna rapport kommer vi att fördjupa oss i en mer dynamisk datastruktur, länkade listor (Linked Lists), som erbjuder större flexibilitet i minnesallokering jämfört med traditionella arrayer. Vi kommer att undersöka hur man implementerar en länkad lista och utforska några grundläggande operationer, såsom att slå ihop två listor. Därefter jämför vi denna datastruktur med arrayer för att belysa skillnaderna i deras användningsområden och prestanda. Slutligen kommer vi även att kolla närmare på hur en länkad lista kan användas för att implementeras en stack.

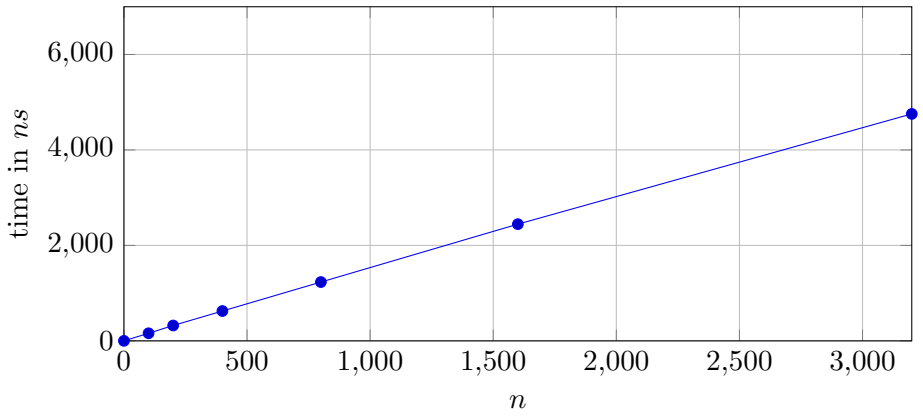
Länkade listor

En länkad lista är en dynamisk datastruktur som består av en sekvens av noder, där endast den första noden i listan är direkt tillgänglig. Varje nod innehåller både data och en referens till nästa nod, och det är dessa referenser som skapar kopplingarna mellan noderna och bildar listan. Till skillnad från arrayer, som allokerar minne för alla sina element i ett sammanhängande block, allokerar länkade listor minne för varje nod separat vid behov. Den sista noden i listan har en referens som är satt till null, vilket indikerar att det är det sista elementet i listan.

Benchmark

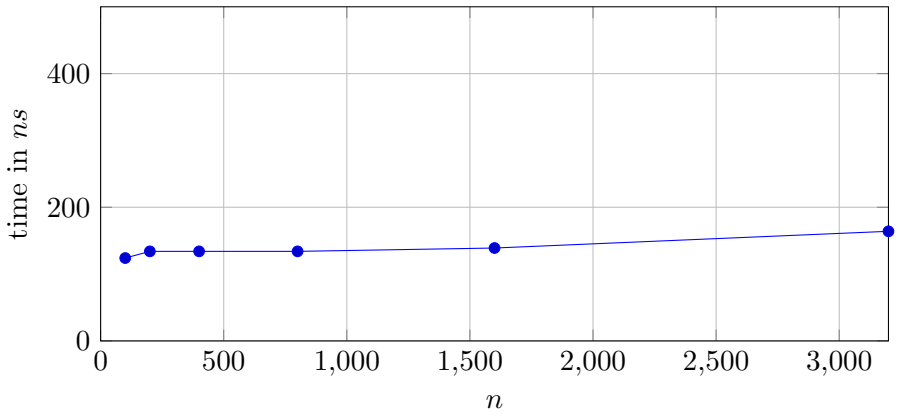
Det finns flera användbara operationer som kan utföras på länkade listor. Vi ska kolla närmare på operationen där två listor slås samman. Vi kommer dels att undersöka fallet när den första listan innehåller ett varierande antal element och vi lägger till en lista med ett fast antal element, samt det omvända fallet där den andra listan har ett varierande antal element och den första ett fast antal element.

I figur 1 nedan visas grafen för fallet där antalet element i den första listan varierar. Av grafen framgår det tydligt att operationen har en tidskomplexitet på $O(n)$, vilket är logiskt. Detta beror på att append()-funktionen, som används för att slå ihop två listor, måste traversera hela den första listan tills den når det sista elementet. Därefter uppdateras den sista nodens referens så att den pekar på det första elementet i den andra listan. Alltså måste append()-funktionen traversera alla n element i den första listan, där n är längden på listan.



Figur 1: Lista A innehåller n element och vi lägger till en konstant lista b

I figur 2 nedan visas fall två, där den första listan är konstant och den andra listan, som vi ska länka på, har ett varierande antal element, n. Av grafen framgår det tydligt att denna operation har en tidskomplexitet på $O(1)$, vilket är logiskt. Som vi såg i det första fallet traverserar append()-funktionen genom hela den första listan och kopplar sedan den sista noden i den första listan till den första noden i den andra listan. Eftersom längden på den första listan är konstant, kommer append()-funktionen alltid att traversera samma antal element, oavsett hur många element den andra listan innehåller.



Figur 2: Lista A innehåller en konstant mängd element och det läggs till en lista B som innehåller n element

Länkade listor vs arrayer

För att utföra samma operation som att slå ihop två länkade listor, måste en annan metod användas när man arbetar med arrayer. Det är en betydligt mer komplex process. Först måste en ny array skapas, med en storlek som motsvarar summan av de två ursprungliga arrayarnas längder. Därefter kopieras alla element från den första arrayen till den nya arrayen, följt av att elementen från den andra arrayen läggs till. Denna process är betydligt mer resurskrävande jämfört med att slå ihop två länkade listor, då alla element i båda arrayerna måste dupliceras till en ny datastruktur. Detta leder till en högre tidskomplexitet för denna typ av operation på arrayer.

Stackar med hjälp av länkade listor

Att implementera en stack med hjälp av en länkad lista är en effektiv lösning eftersom den dynamiskt kan anpassa sin storlek. Detta innebär att vi undviker den extra minneshantering som annars krävs för att öka eller minska storleken på en stack när den implementeras med arrayer. Eftersom en stack följer LIFO-principen (Last In, First Out) sker alla insättningar och borttagningar av element i början av den länkade listan.

Push-metoden fungerar genom att skapa en ny nod, sätta denna nod som den nya toppen av stacken och låta den referera till den tidigare toppen. På så sätt blir den nya noden snabbt och enkelt den översta noden i stacken.

```
public void push(int data) {
top = new Node(data, top);
}
```

Pop-metoden tar bort det översta elementet från stacken. Först kontrolleras om stacken är tom. Om den inte är tom, returneras nodens värde samtidigt som den översta noden tas bort genom att flytta pekaren till nästa nod i listan.

```
public int pop() {
    if (top == null) {
        System.out.println(Stack is empty);
        return -1;
    }
    int data = top.data;
    top = top.next;
    return data;
}
```

Slutsats/diskussion

I denna rapport har vi undersökt länkade listor och delvis jämfört dem med arrayer. Länkade listor erbjuder större flexibilitet genom dynamisk minneshantering, vilket gör dem mer lämpade för situationer där storleken på datastrukturen är okänd eller kan förändras. Arrayer, å andra sidan, erbjuder snabbare åtkomst till slumpmässiga element och kan ofta dra fördel av cacheminnets effektivitet, eftersom arrayer lagras sekventiellt i datorns minne. Detta skiljer sig från länkade listor, där noderna kan vara spridda i minnet. Dock kräver arrayer omallokering vid storleksändringar, vilket gör dem mindre flexibla än länkade listor.Vid implementering av stackar visade sig länkade listor vara en effektiv lösning, särskilt när dynamiska insättningar och borttagningar krävs.