Umeå universitet

26 Februari 2024

Datastrukturer och Algoritmer

Python 5DV150

OU3

Vuk Dimovic

vudi0001@student.umu.se

Kursansvarig: Ola Ringdahl

Contents

1 Introduktion

Syftet med denna laborationen är att analysera de olika datastrukturernas prestanda för att först deras styrkor och svagheter, detta utförs genom att analysera tidskomplexiteten av dessa datastrukturer. Vi har 3 olika datastrukturer som ska analyserars, detta är: Array (fält), Länkad lista(List) och en modiferad länkad lista algoritm som använder Move-To-Front algoritm (MTF), dessa 3 kommer att implementeras som en tabell.

2 Metod

Vi har blivit givna en färdig skriven kod för List, uppgiften bygger på att vi ska kunna själv modifiera List koden för MTF och utöver skapa sitt egen Fält kod. Det är viktig att ta hänsyn att länkade listor använder en cellig datastruktur, där element i en cellig lista pekar alltid på efterkommande element. I alla 3 implementationer kommer tuplar att användas, de har 2 värde. Själva värdet som är lagrad och en unik "nyckel", i rapporten kommer det tuplar att refereras som element. När vi ska konstruera Fält koden så har vi blivit tilldelat en specifik gränsyta och vår kod ska kunna hantera följande:

2.1 Luckor mellan element.

När ett element tas bort, flyttas alla efterföljande element en steg framåt för att fylla luckan, detta resulterar i att alla positioner i tabellen är fyllda.

2.2 O(1) Tids komplexitet när tabellen är tom

För att se till att det kommer alltid att kosta O(1) om tabellen är tom så skapar vi en attribut som har koll på storleken av tabellen. För att kontrollera om tabellen är tom så jämför koden storleken på denna attribut med noll, vilken är en konstant operation.

2.3 Hantering av Dubbletter

I logiken för "insert" metoden så kontrollerar koden först att den angivna nyckel finns i tabellen. Om nyckeln finns redan så ersätter den värden för den befintliga nyckeln med det nya värdet.

2.4 Borttagning av nycklar

När man försköker ta bort en nyckel som finns inte i tabellen, så kastar programmet ut en "ValueError" undantag.

2.5 Implementationen av MoveToFront

Koden uppdaterar ordningen på elementen så att när en nyckeln används, kommer den att flyttas till framsidan av tabellen. Detta uppnås genom att ta bort nyckeln från nuvarande positionen och skicka den till framsidan av tabellen. En korrekt implementering av tomma luckor och dubbletter krävs för att MoveToFront ska fungera.

3 Resultat

Här förekommer 3 tabeller som representerar hur länge det behövdes för att utföra vissa tester för hur snabba algoritmer är. Detta gjordes med den nya version av TableTest3.py i Python. Notera att dessa tider kan variera på grund av olika datorer, vi har änvant bärbara datorer som kan påverka hur snabbt operationer egentligen är.

Items	**Insert**	**R Lookup**	**Skew lookup**	**Lookup no keys**	**Remove items**
1000	0.08	0.05	0.02	0.09	0.06
5000	0.78	0.98	1.40	4.45	3.46
10000	2.63	4.63	3.27	6.88	4.95
15000	6.04	5.30	4.28	6.65	4.31
20000	7.58	6.03	5.50	8.32	4.34

Tabell 1: TableAsArray körtider för olika mängder av elementer, tidsenhet i sekunder.

Items	**Insert**	**R Lookup**	**Skew lookup**	**Lookup no keys**	**Remove items**
1000	0.33	0.20	0.12	0.70	0.23
5000	1.55	1.56	1.59	4.33	1.29
10000	4.21	4.73	5.16	7.58	2.40
15000	6.89	7.15	4.29	8.04	2.85
20000	9.35	6.46	6.15	8.78	3.00

Tabell 2: TableAsList körtider för olika mängder av elementer, tidsenhet i sekunder.

Items	**Insert**	**R Lookup**	**Skew lookup**	**Lookup no keys**	**Remove items**
1000	0.53	0.32	0.23	0.91	0.17
5000	1.03	2.32	1.14	5.29	1.13
10000	4.21	4.58	2.05	7.31	1.82
15000	6.33	5.72	1.71	9.93	1.78
20000	9.66	5.26	2.19	10.11	1.99

Tabell 2: TableAsMTF körtider för olika mängder av elementer, tidsenhet i sekunder.

4 Analys: Asymptotisk komplexitetsanalys

Denna sektion kommer att presentera en asymptotisk komplexitetsanalys av funktioner associerade med tabell-implementationer med datatyperna Fält och Lista.

4.1 Fält

4.1.1 Empty (__init__) O(1)

Konstruktör skapar en instans av en Array objekt med specificerad kapacitet och ställer in initiala storleken på fält till 0. Alltså en tom fält. Detta metod har tidskomplexitet av O(1), detta är på grund av att det krävs en initialiseringsprocessen skalar inte med antalet element i fältet.

4.1.2 Insert O(n)

Insert-operationen lägger till ett element i fältet. Den först kollar om tabellen är full genom att jämföra tabellens storlek med själva kapaciteten av objekten. Om tabellen är inte full så itererar metoden genom existerande element för att kolla att nyckeln redan existerar. Detta ger oss i värsta fallet tids komplexitet O(n)

4.1.3 IsEmpty O(1)

IsEmpty-metoden kontrollerar om fältet är tomt genom att verifiera om storleken är 0. Denna operation är oberoende av storleken av fälten. Detta implicerar att metoden har en konstant tids komplexitet, alltså O(1).

4.1.4 Lookup O(n)

Lookup-operationen söker efter ett specifikt element i fälten. Denna metod itererar genom varje element i fälten tills den önskat element hittades eller till fältets slut. Detta betyder att den har tidskomplexitet O(n)

4.1.5 Remove O(n)

Remove-metoden tar bort ett element från fältet. Det första steget är att hitta elementet, detta process är identisk till Lookup-metoden som implicerar att den har en tidskomplexitet av O(n). Därefter måste de återstående element skiftas för att fylla up det tomma utrymmet, vilket också har tidskompexitet O(n) i värsta fall. Det är viktig att notera att detta sker sekvensiellt, alltså kommer den totala tidskomplxeitet vara O(n)

4.2 Lista

4.2.1 Empty (__init__) O(1)

En tom tabel initieras med Directed List object, detta är en konstant tids operation O(1)

4.2.2 Insert O(n)

När ett element infogas i en riktad lista, krävs det att man itererar genom listan för att garantera att inga dubbletter av nycklar ska existera. Om tabellen är inte tom så betyder det att funktionen måste iterera över hela tabellen, detta ger i värsta fall tids komplexitet O(n).

4.2.3 Isempty O(1)

Detta metod verifierar att tabellen är inte tom. Eftersom detta är en enkel operation som involverar verifiering om listans huvudpekare är null eller inte. Denna operation är oberoende av listans storlek som implicerar att den har tids komplexitet av O(1).

4.2.4 Lookup O(n)

Att hitta en element i en lista innebär att man itererar genom listan med en unik nyckel tills elementen med samma nyckel har hittas eller slutet av listan har blivit nåd. Detta operation har en linjär tids komplexitet som i värsta fallet ger en tids komplexitet av O(n).

4.2.5 Remove O(n)

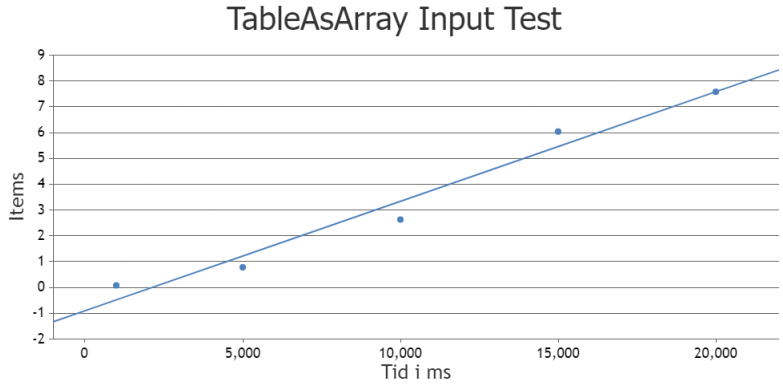
Borttagning av en element kräver först att elementen hittas. Detta motsvarar Lookup-operationen som har en tids komplexitet av O(n). Därefter pekare av de gränsade noder måste justeras. Det är viktig att notera att justeringen av pekarna är kan ha en konstant tidsoperation, men eftersom tids komplexitet av sökning steget är O(n), så kommer den övergripande tidskomplexitet att vara O(n).

4.3 Skillnader mellan Fält och Lista

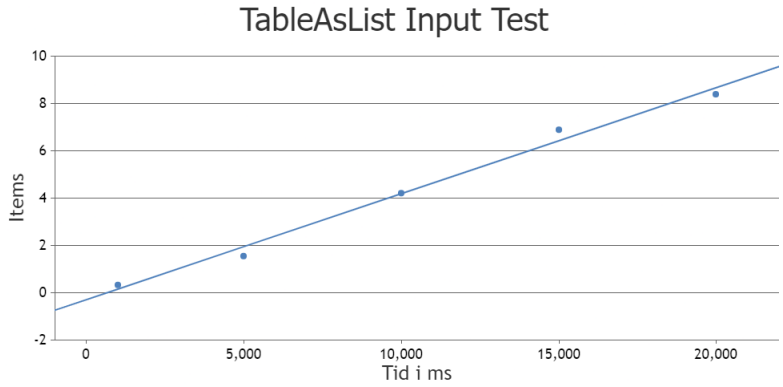
Det största skillnader ligger i hantering av datas storleksförändringar och hur det kan påverka prestanda av olika operationer. Fält offrar snabbare åtkomst till element men har statisk storlek, medans listan storlek är flexibel och kräver mer tid för vissa operationer på grund av behovet att läsa hela datastrukturen.

5 Analys: Experimentell komplexitetsanalys

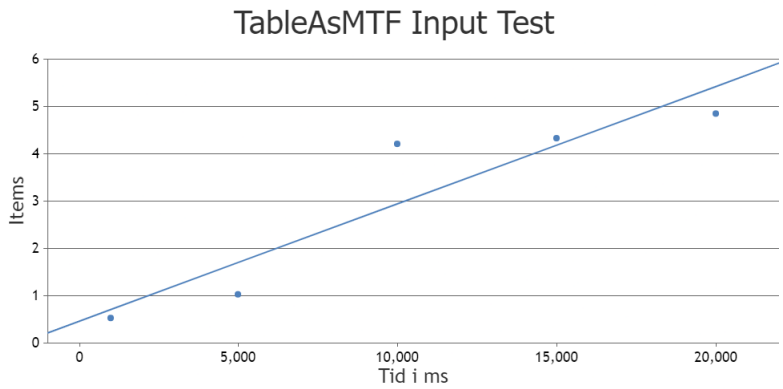
Vi kommer att analysera infognings tiden för alla 3 tabeller. Vi gör detta eftersom det analyserar hur snabbt och effektict en datastruktur kan ha tillgång, lagra, och manipulera data. Datan som presenteras här är samma data som i Resultat delen av rapporten.



Tabell 4: Detta är en graf för TableAsArray koden, dessa värde kommer från den nya TableTest3.py . Notera att den blå linjad är den fitted linje, alltså den teoretiska linjära ökning.



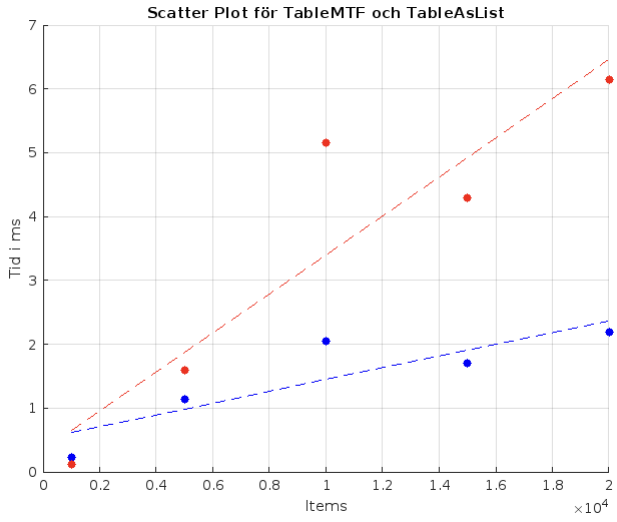
Tabell 5: Detta är en graf för TableAsList koden, dessa värde kommer från den nya TableTest3.py . Notera att den blå linje är den fitted linje, alltså den teoretiska linjära ökning.



Tabell 6: Detta är en graf för TableAsMTF koden, dessa värde kommer från den nya TableTest3.py . Notera att den blå linjad är den fitted linje, alltså den teoretiska linjära ökning.

5.1 Skillnaden mellan TableAsList och TableAsMTF

Det är en subtil skillnad mellan TableAsList och TableAsMTF. TableAsMTF minskar tiden att hitta en efterfrågad nyckel efter första gången, eftersom nyckeln flyttas till början av listan, detta gör framtida sökningar snabbare. I TableAsList bör tiden att hytta en nyckeln inte förändras oavsett hur mycket den efterfrågas, eftersom varje sökning av en nyckel början från listans första position. Därför ska vi kolla på skewed lookup av de två för att jämföra de.



Tabell 7

Detta graf visualiserar skewed lookup av TableAsList och TableAsMTF. Den röda linjer och punkter är associerad med TableAsList och de blå linjer och punkter är för TableAsMTF. Skewed lookups är anpassad för TableAsMTF miljö eftersom framtida sökningar i TableAsMTF tenderar att vara närmare listans början efter första efterfrågan. I en miljö där vissa nycklar efterfrågas oftare än andra kan det minska den genomsnittliga söktiden signifikant

6 Analys: Övrigt

6.1 ”Allmänt bäst” algoritm

Det finns ingen algoritm som kommer att klara ut alla möjliga funktioner, alla algoritmer är speciella på sin sätt. I teorin så bör fält vara den snabbaste algoritmen när det gäller infogning av element i en tabell. Fälten tabell är statisk, som innebär att kapaciteten på tabellen är en konstant. Samt att Lista och Move-To-Front algoritmer använder en celling länkad lista.

Dock betyder det inte att fält algoritmen är ”bäst”, detta kan man se tydlig i resultaten för bortagning av element i tabellen. Där skillnaden är nästan dubbelt så stort. Det är viktig att veta:

- För tillfälle där direkt tillgång till en känd mängd av element är fält den bästa algoritmen
- För tillfälle för frekvent infogning och bortagning av element som är inte tillgänglig med en index, en länkad lista är den bästa algoritmen
- För tillfälle där små mängder av element används ofta, en move-to-front algoritm är bäst

6.2 Skillnad mellan dynamisk och statisk implementation

Som nämnt tidigare så offrar TableAsArray effektiv åtkomst till element, men orskarar brist på flexibiliet när det gäller löpande storleksförändringar, den har lägre overhead men offrar mindre flexibilitet, Dynamiska implementationer i både TableAsList och TableAsMTF erbjuder större flexibilitet men har mindre effektiv Lookup-metod.

6.3 Förbättring av gränsytan för Fält

För att förbättra gränsytan för fält kan vi optimera Remove-metoden. Genom att enbart ersätta det borttagna elementet med det sista elementet i fältet, istället för att flytta alla element för att täcka den bortagna element. Detta optimering minskar antal operationer dramatisk och ger oss en konstant tidskomplexitet, alltså O(1).

7 Reflektion

Jag tyckte att arbetet var relativ svårt. Det krävdes mycket tänkande när det gäller att förklara hur de olika datatyper är olika från varandra och hur jag motiverade deras unika egenskaper. Det som var mest svårt vart att kunna förstå vad uppgiften faktiskt kräver, i många moment av uppgiften så fanns det oklara instruktioner. Speciellt i själva formaten av rapporten och komplexitetsanalysen. Dock så tyckte jag att uppgiften vart ändå fylld med möjligheter att kunna utöka min kunskap inom datatyper och algoritmer.

8 Referenser

Datatyper och algoritmer., Lars-Erik, J., Torbjron, W., Upplaga 2 (2011)