UMEÅ UNIVERSITET
Department of Computing Science
Assignment 3

April 28, 2023

5DV149 Datastrukturer och algoritmer

Assignment 3 — Comparison of Table implementations

version 1.0

Name Adam Pettersson
Username hed21apn

Graders
Graders

Contents

1 Introduction

Hej, Är medveten om att denna raport ej är komplett.

Försöker återgärda det till nästa inlämning, får nämligen inte till det med data hanteringen I Matblab (har knappt någon erfarenhet av programmet sen tidigare) och vet inte riktigt vart jag ska vända mig.

I detta dokument presenteras en implementation av datatypen Tabell samt en analys av dess komplexitet. Tabellen är utformad för att lagra nyckel-värde-par och stöder flera operationer för att skapa, manipulera och söka i tabellen. Vi går igenom olika metoder och strategier som används för att implementera tabellen, såsom användning av generiska strukturer, dubbelt länkade listor och hantering av dubbletter.

Förutom en genomgång av datatypens grundläggande funktioner och användargränssnittet, kommer vi även att diskutera detaljer kring implementeringen, inklusive olika typer av tabeller såsom Table, MTFTable och ArrayTable. Dessa olika implementeringar är anpassade för att hantera olika användningsfall och kan erbjuda olika prestandafördelar.

Slutligen presenteras en förenklad asymptotisk komplexitetsanalys för tabellens funktioner. Denna analys hjälper oss att förstå hur tabellens funktioner skalar med avseende på antalet element i tabellen och ger oss en uppfattning om hur effektivt tabellen kan hantera olika arbetsbelastningar.

2 Datatypes

För att implementera datatypen Tabell användes följande metoder och strategier:

Användning av en generisk struktur för tabellerna, vilket innebär att nycklar och värden kan vara av vilken datatyp som helst, och jämförelsefunktionen för nycklar och frigöringsfunktionerna för nycklar och värden kan anpassas.

Användning av en dubbelt länkad lista (dlist) för att lagra tabellinlägg, där varje inlägg representeras av en struktur `table_entry` som innehåller en nyckel och ett värde.

Hantering av dubbletter genom att tillåta att flera inlägg med samma nyckel läggs till i tabellen, utan att kontrollera om nyckeln redan finns. Denna strategi gör det möjligt att snabbt infoga nya värden, men kan leda till att tabellen blir större och mer komplex.

Vid sökning (`table_lookup`) returneras det senast inlagda värdet för en dubbel nyckel, eftersom det är det första värdet som hittas när listan traverseras från början.

Vid borttagning (`table_remove`) tas alla dubbletter av en angiven nyckel bort genom att iterera över listan och jämföra varje inläggs nyckel med den angivna nyckeln. Om en matchning hittas, tas inlägget bort och alla frigöringsfunktioner som registrerats vid tabellens skapande anropas för att frigöra minne för nyckeln och/eller värdet.

Dessa metoder och strategier utgör grunden för implementeringen av datatypen Tabell och dess funktioner, och gör det möjligt att arbeta med tabeller på ett effektivt och flexibelt sätt. Hanteringen av dubbletter innebär att man måste göra avvägningar mellan snabbhet vid insättning och enkelhet i tabellens struktur, men detta kan anpassas efter behov och användningsområden.

2.1 The Table user interface

I detta avsnitt presenteras de viktigaste funktionerna som ingår i tabellens användargränssnitt. Varje funktion beskrivs kortfattat för att ge en översikt över dess syfte och användning.

`table_empty`: Skapar en ny tom tabell med de angivna jämförelse- och frigöringsfunktionerna för nycklar och värden.

`table_is_empty`: Kontrollerar om en tabell är tom, det vill säga om den inte innehåller några nyckel/värde-par.

`table_insert`: Lägger till ett nyckel/värde-par i tabellen. Funktionen kontrollerar inte om nyckeln redan finns i tabellen och tillåter därför dubbletter.

`table_lookup`: Söker efter en given nyckel i tabellen och returnerar det tillhörande värdet, eller NULL om nyckeln inte hittades. Om det finns dubbletter av nyckeln returneras det senast inlagda värdet.

`table_choose_key`: Returnerar en godtycklig nyckel som lagras i tabellen. Kan användas tillsammans med `table_choose_key` för att avkonstruera tabellen.

`table_remove`: Tar bort ett nyckel/värde-par från tabellen baserat på den angivna nyckeln. Funktionen tar även bort alla dubbletter av den angivna nyckeln.

`table_kill`: Förstör tabellen och frigör allt dynamiskt allokerat minne som används av tabellen och dess element. Om en `free_function` registrerades vid tabellens skapande kommer den att kallas för varje element för att frigöra eventuellt användarallokerat minne för elementens värden.

`table_print`: Skriver ut tabellen genom att iterera över nyckel/värde-paren i tabellen och anropa den angivna utskriftsfunktionen för varje par.

Dessa funktioner utgör grunden för att använda och interagera med tabellen och dess data. Genom att förstå och använda dessa funktioner kan man effektivt arbeta med tabellstrukturen och dess innehåll.

2.2 Implementation details

2.2.1 Table

Det här är en implementation av en generisk tabell (hash table) som används i datavetenskapskurserna vid Umeå Universitet. Koden är skriven i C och använder en dubbellänkad lista som grund för tabellen. Tabellen är generisk och kan användas för att lagra nyckel/värde-par av olika datatyper.

Koden innehåller en struktur för tabellen (struct table) och en struktur för

tabellens poster (`struct table_entry`). Tabellen innehåller en dubbellänkad lista, en jämförelsefunktion för nycklar och två fria funktioner (en för nycklar och en för värden). Tabellposterna består av en nyckel och ett värde.

2.2.2 MTFTable

Denna implementation använder en "move to front" lista (dlist) för att lagra nyckel-värde-par i en tabell. Dlist är en dubbellänkad lista som kan ändra storlek under körning.

Tabell-datstrukturen är utformad för att lagra nyckel-värde-par, där varje nyckel är kopplad till ett värde. Implementeringen använder en dlist för att lagra nyckel-värde-par som `table_entry-strukturer`. När ett element läses av från tabellen flyttas det automatiskt till början av listan. Detta förbättrar prestandan när samma element söks flera gånger.

I denna implementation kan nycklar vara av vilken datatyp som helst, och värden kan också vara av vilken datatyp som helst. Tabellen använder funktionspekare för att jämföra nycklar, frigöra nycklar och frigöra värden. Dessa funktionspekare måste anges när man skapar en ny tabell.

Dubbletter hanteras på så sätt att när ett nytt nyckel-värde-par infogas med en redan existerande nyckel ersätts det tidigare värdet som är kopplat till nyckeln. Funktionen `table_lookup()` returnerar alltid det senast tillagda värdet för en dubblett-nyckel. När man använder funktionen `table_remove()` tas alla dubbletter för en viss nyckel bort.

2.2.3 ArrayTable

Denna implementation använder en dynamiskt storlekad array (`array_1d`) för att lagra nyckel-värde-par i en tabell. `Array_1d` är en anpassad 1-dimensionell array som kan ändra storlek under körning. Denna implementation tillåter tabellen att ha valfritt antal element upp till `MAX_TABLE_SIZE`, som definieras som 80000.

Tabell-datstrukturen är utformad för att lagra nyckel-värde-par, där varje nyckel är kopplad till ett värde. Implementeringen använder en `array_1d` för att lagra nyckel-värde-par som `table_entry-strukturer`, vilket ger effektiv åtkomst och ändring av tabellens element.

I denna implementation kan nycklar vara av vilken datatyp som helst, och värden kan också vara av vilken datatyp som helst. Tabellen använder funktionspekare för att jämföra nycklar, frigöra nycklar och frigöra värden. Dessa funktionspekare måste anges när man skapar en ny tabell.

Dubbletter hanteras på så sätt att när ett nytt nyckel-värde-par infogas med en redan existerande nyckel ersätts det tidigare värdet som är kopplat till nyckeln. Funktionen `table_lookup()` returnerar alltid det senast tillagda värdet för en dubblett-nyckel. När man använder funktionen `table_remove()` tas alla dubbletter för en viss nyckel bort.

2.3 Complexity analysis

Nedan visas den förenklade asymptotiska komplexitetsanalysen för funktionerna.

	`table`	`mtftable`	`arraytable`	...
`empty`	$O(1)$	$O(1)$	$O(1)$	
`isempty`	$O(1)$	$O(1)$	$O(1)$	
`insert`	$O(1)$	$O(1)$	$O(n)$	
`lookup`	$O(n)$	$O(n)$	$O(n)$	
`remove`	$O(n)$	$O(n)$	$O(n)$	
`choose_key`	$O(1)$	$O(1)$	$O(1)$	
`kill`	$O(n)$	$O(n)$	$O(n)$	
`print`	$O(n)$	$O(n)$	$O(n)$	

Sammanfattningsvis visar tabellen att många av tabellens operationer har en linjär asymptotisk komplexitet (O(n)). Dessa operationer inkluderar `table_insert`, `table_lookup`, `table_remove`, `table_kill` och `table_print`. Det beror på att dessa funktioner behöver iterera över tabellens element för att utföra sina respektive uppgifter.

Å andra sidan har operationerna `table_empty`, `table_is_empty` och `table_choose_key` en konstant asymptotisk komplexitet (O(1)). Detta beror på att dessa operationer inte behöver iterera över tabellens element och kan utföras oberoende av tabellens storlek.

Det är värt att notera att både `table_insert` och `table_remove` har en O(n) komplexitet, men `table_insert` kan ha en mindre komplexitet i vissa fall. I fallet där ett dubblett-nyckel hittas under insättning kommer `table_insert` att sluta tidigare och därmed ha en lägre komplexitet än O(n) i det fallet. Men i det värsta fallet kommer `table_insert` att ha en O(n) komplexitet.

3 Experiments

What did you do? How did you set up your experiment? (E.g. explain how you ran all k implementations on the same m problem sizes.) How many times did you repeat? What hardware did you run on? Under what operating system? If you used an external code package for the computations, e.g. Matlab, specify its name and version number. Anything interesting that could affect the results should be mentioned.

3.1 User instructions

Inledningsvis, kompilera filerna enligt nedan och ersätt table.c med den fil som ska testas:

```
gcc -Wall -I<path/to/include> -o tabletest tabletest.c
<path/to>/table.c dlist.c
```

3.2 Test runs

```
Testing...
Isempty returns true directly after a table is created. - OK
Isempty false if one element is inserted to table. - OK
Test of looking up non-existing key in a table with one element - OK
Looking up existing key in a table with one element - OK
Looking up three existing keys-value pairs in a table with three elements - OK
Looking up existing key and value after inserting the same key three times with different values - OK
Inserting one element and removing it, checking that the table gets empty - OK
Inserting three elements and removing them, should end with empty table - OK
Inserting three elements with the same key and removing the key, should end with empty table - OK
All correctness tests succeeded!

Insert 10000 items                  : 3 ms.
Remove all items                    : 822 ms.
10000 lookups with non-existent keys : 1546 ms.
10000 random lookups                : 814 ms.
10000 skewed lookups                : 776 ms.
Test completed.
```

Figure 1: Testrun of table

```
Testing...
Isempty returns true directly after a table is created. - OK
Isempty false if one element is inserted to table. - OK
Test of looking up non-existing key in a table with one element - OK
Looking up existing key in a table with one element - OK
Looking up three existing keys-value pairs in a table with three elements - OK
Looking up existing key and value after inserting the same key three times with different values - OK
Inserting one element and removing it, checking that the table gets empty - OK
Inserting three elements and removing them, should end with empty table - OK
Inserting three elements with the same key and removing the key, should end with empty table - OK
All correctness tests succeeded!

Insert 10000 items                  : 2 ms.
Remove all items                    : 837 ms.
10000 lookups with non-existent keys : 1544 ms.
10000 random lookups                : 1070 ms.
10000 skewed lookups                : 485 ms.
Test completed.
```

Figure 2: Testrun of mtftable

```
Testing...
Isempty returns true directly after a table is created. - OK
Isempty false if one element is inserted to table. - OK
Test of looking up non-existing key in a table with one element - OK
Looking up existing key in a table with one element - OK
Looking up three existing keys-value pairs in a table with three elements - OK
Looking up existing key and value after inserting the same key three times with different values - OK
Inserting one element and removing it, checking that the table gets empty - OK
Inserting three elements and removing them, should end with empty table - OK
Inserting three elements with the same key and removing the key, should end with empty table - OK
All correctness tests succeeded!

Insert 10000 items                 : 665 ms.
Remove all items                   : 958 ms.
10000 lookups with non-existent keys : 1297 ms.
10000 random lookups               : 599 ms.
10000 skewed lookups               : 648 ms.
Test completed.
```

Figure 3: Testrun of arraytable

3.3 The actual experiments

För att köra testerna på ett smidigt sätt använde jag mig utav kommandot:

```
tabletest.exe -t -n 10000 > timing10000.txt
```

Med hjälp av kommandot ovan körs alla de tester för den mängd som specificeras efter -n flaggan. Resultatet från körningen hamnar sedan i timing.txt filen.

Jag använde mig av n 100, 1000, 5000, 10000, 16000 och 25000. Detta för att kunna skissa en nogrann graf med stort spann på mängden element.

4 Results

För att kunna få en ordentlig mängd data körde jag testprogrammen tre gånger för varje antal n. Den data som presenteras nedan är medelvärdet för varje tabell körning.

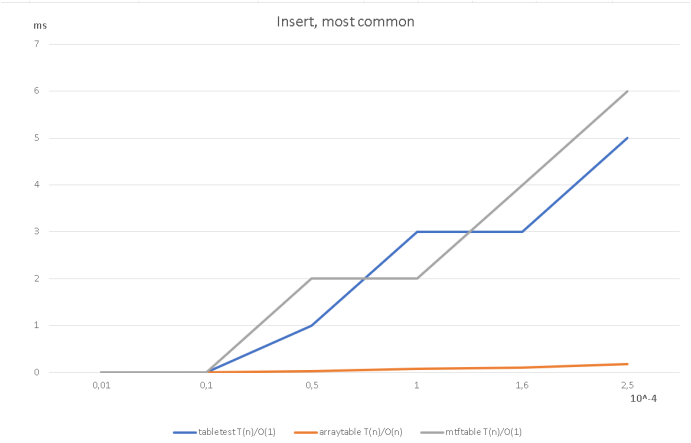

Figure 4: One

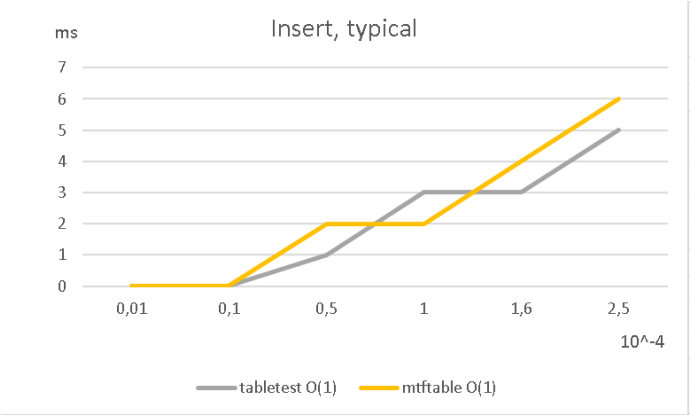

Figure 5: Two

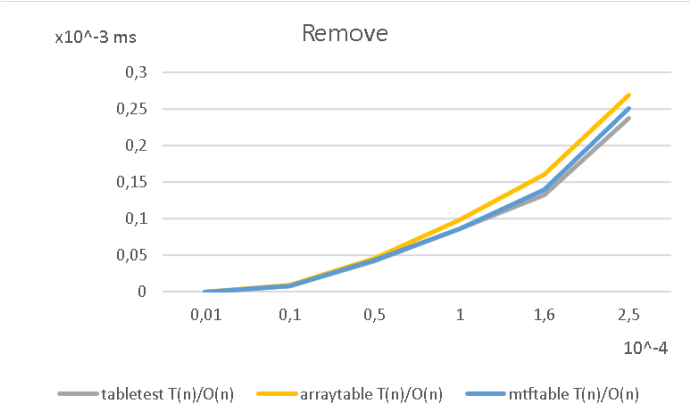

Figure 6: Three

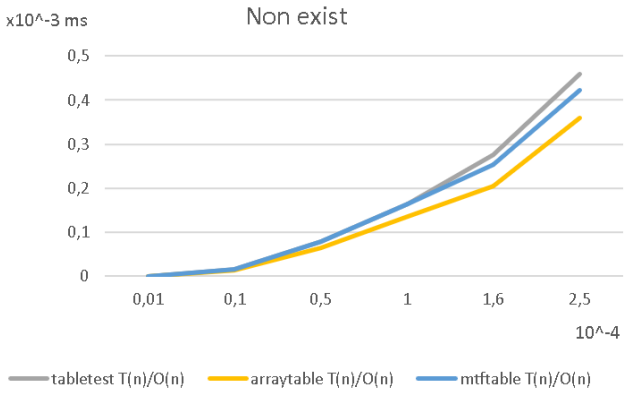

Figure 7: One

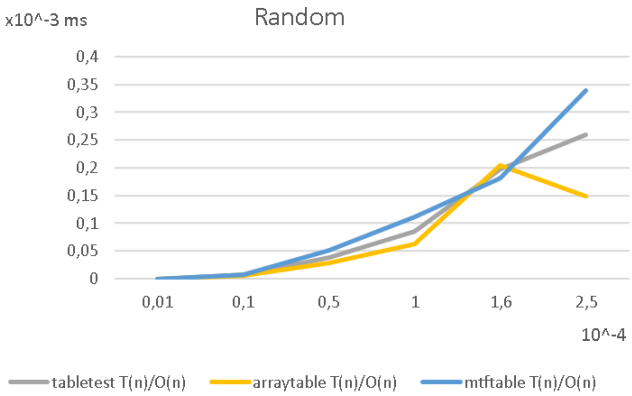

Figure 8: Two

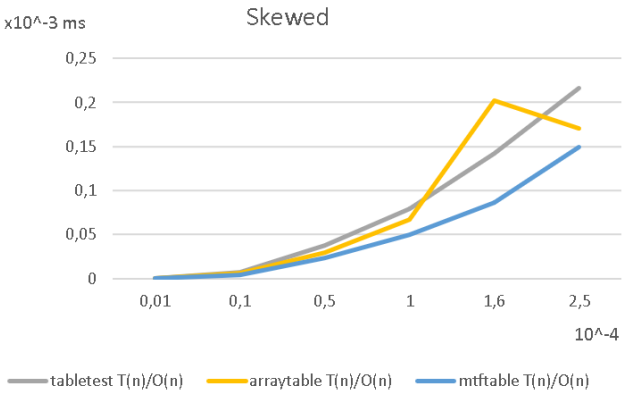

Figure 9: Three

5 Discussion

5.1 Discussion

What does it mean?

I och med att jag inte vår till hanterandet av datat kan jag inte skriva någon diskusion.

5.2 Conclusion

Samma som för diskussionen.

5.3 Reflections

Jag tyckte personligen att uppgiften var mycket lärorik för att förstå sig på tabeller i allmänhet. Det var även nyttigt och givande att få använda ordo begreppet i en mer praktiskt applicerbar miljö, något som jag tyckte verkligen ökade förståelsen för hur innefektiv kod kan påverka belastning vid användning av koden.

Denna stora utmaningen för min del vart dock lustigt nog att samanställa resultatet. Jag har sedan tidigare inte så stor erfarenhet av matlab och hade svårt att ta mig fram i det tekniska kring hur jag skulle skissa graferna enligt önskemålen.