5DV160/HT17: OU3 – Tabeller

Buster Hultgren Wärn
dv17bhn@cs.umu.se

2017-12-12

Innehåll

1. Inledning

Denna uppgift består av tre olika implementationen av tabeller. En tabell är en typ av datastruktur som arbetar med olika "par". Varje par måste bestå av en nyckel och sen ett eller flera värden. Då genom nyckeln varje värde i paret kan nås.

De tre olika implementationerna som användes var: riktad lista, array och sen hash-tabell. Implementationerna kommer att beskrivas i kapitel 2.

De olika implementationernas prestanda testades också för att se vad för påverkan dem har på strukturen, vilken kommer förklaras djupare i kapitel 3. Analysen på de olika implementationernas prestanda skillnad kommer i kapitel 4.

2. Implementation

Det är viktigt att nämna att i tabellerna hanteras dubbletter av nycklar vid insättning av ett par. När paret sätts in så undersöks det alltså ifall nyckeln redan finns. Ifall nyckeln finns, så överskrivs den.

Tabell som en enkellänkad lista

Implementationen av en riktad list var redan given, och behövde inte skrivas från min del.

Den är, som tidigare nämnt, byggd på en riktad lista. Detta betyder att varje nod i listan har en referenspekare till nästa nod. Varje nod är även ett par, vilket betyder att de har en nyckel och ett värde.

Komplexiteten av fölande operationer, beskrivet i stora O, kommer att vara som följande:

Empty – O(1)
Is_empty – O(1)
Insert – O(n)
Lookup – O(n)
Remove – O(n)
Kill – O(n)

Tabell som array

Arrayen är uppbyggd med dynamiskt minne. Det vill säga att en godtycklig (fast begränsad till tillgängligt minne) mängd allokeras i heapen då tabellen skapas. Varje cell i arrayen är ett eget par, som innehåller pekare till värde och nyckel.

Tabellen går att bläddra igenom genom indexering. Det betyder att komplexiteten blir liknande som i den riktade tabellen:

Empty – O(1)
Is_empty – O(1)
Insert – O(n)
Lookup – O(n)
Remove – O(n)
Kill – O(n)

Tabell som hash-tabell

Hash-tabellen är uppbyggd lite som en blandning mellan arrayen och den riktade listan. Den består av olika "buckets", där varje bucket är början av en riktad lista. Varje nyckel i ett nytt par som stoppas in i tabellen får ett hash-värde. Detta skall vara ett så unikt värde som möjligt – d.v.s. att i bästa fall så har varenda nyckel en unik hash. Beroende på vilket hash-värde nyckeln får, stoppas de in i olika buckets.

Så ifall det finns, t.ex. 100 värden så kan det finnas fem olika buckets där optimalt varje bucket innehåller 20 stycken par. Detta leder till att stega igenom tabellen så behövs endast 20 stycken par undersökas.

Detta är dock förutsatt att hash-funktionen är optimalt byggd och att varje bucket får lika många element. Detta leder till att det blir svårt i praktiken att förutsätta hur fördelningen kommer att bli. Detta kan variera på indatan, hur många nycklar sätts in, hur ser nycklarna ut? I värsta fall så får en hink alla element, vilket leder till att komplexiteten fall kommer att vara samma som i den riktade listan:

Empty – O(1)
Is_empty – O(1)
Insert – O(1)
Lookup – O(n)
Remove – O(n)
Kill – O(n)

3. Experiment

Testresultaten gjordes genom att ett test program körde fem gånger för varje fil. Testprogrammet satte in 10 000 stycken par i tabellen och körde fem olika tester. Dessa tester är:

Test 1: Sätt in 10 000 stycken par i tabellen med unika nycklar.

Test 2: Leta upp alla 10 000 nycklar som finns i paren.

Test 3: Leta upp par 10 000 gånger från en delmängd av 34 olika nycklar.

Test 4: Leta upp 10 000 stycken nycklar som inte finns inne i tabellen.

Test 5: Ta bort 10 000 stycken nycklar

Resultaten av testfallen sammansätts i en tabell:

Tabell 1: Testresultat från de olika tabellerna. (Förkortning Gen. – Genomsnittlig. Förkortning op. – operation).

Implementation	Testfall	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Gen. Summa (s)	Gen. tid för en op.	Gen. körtid (s)
Array	1	0,673497	0,678859	0,671091	0,666354	0,677045	0,6733692		
Array	2	0,676715	0,676538	0,680296	0,656172	0,676422	0,6732286		
Array	3	0,674878	0,671979	0,675015	0,655873	0,672274	0,6700038		
Array	4	1,359268	1,378712	1,36383	1,330326	1,357991	1,3580254		
Array	5	0,411581	0,411316	0,410608	0,407737	0,41205	0,4106584	0,75705708	3,7852854
Länkad lista	1	0,710864	0,723376	0,720342	0,722139	0,727301	0,7208044		
Länkad lista	2	0,636454	0,630045	0,631898	0,632348	0,633684	0,6328858		
Länkad lista	3	0,641571	0,638737	0,639389	0,639735	0,640453	0,639977		
Länkad lista	4	1,253591	1,253154	1,248968	1,247526	1,250267	1,2507012		
Länkad lista	5	0,390673	0,389196	0,389468	0,389242	0,389813	0,3896784	0,72680936	3,6340468
Hash tabell	1	0,003438	0,003274	0,004325	0,003437	0,003419	0,0035786		
Hash tabell	2	0,001792	0,001768	0,001878	0,001766	0,0018	0,0018008		
Hash tabell	3	0,001667	0,001681	0,001681	0,001666	0,001774	0,0016938		
Hash tabell	4	0,000687	0,000658	0,000658	0,000657	0,000657	0,0006634		
Hash tabell	5	0,002573	0,00264	0,002332	0,00232	0,002647	0,0025024	0,0020478	0,010239

4. Analys

I tabell 1 visar det att implementationen av array och länkad inte ger någon betydlig i hastighet skillnad. Det tabellen visar speciellt tydligt är hur hash-tabellen är, vid just detta antal element, över 300 gånger snabbare på att utföra alla sina operationer än de två andra.

Någonting annat att notera är att tiden även är olika på test 1, då olika par sätts in. Detta beror på att dupletter inte tillåts. Vid insättning av ett par måste det först undersökas ifall nyckeln redan finns, och därmed behöver tabellen stegas igenom. Ifall tabellen skulle tillåta dupletter, skulle det endast krävas att sätta in paret på första/sista plats, vilket kräver en operation.

Som tidigare diskuterat i kapitel 2 har alla samma stora O av n i de värsta fallen, vilket implicerar att denna hash-tabell inte uppnådde sitt värsta fall. Alltså, en hink fylldes inte över med alla element, utan de fördelades ut bättre över antalet hinkar så listorna blev kortare. Därmed tar det kortare tid att implementera

5. Slutsats

Givet de följande implementationerna så kan en klar slutsats dra kring vilken som, för just dessa testfall, var den mest effektiva. Detta är hash tabellen. Då det är svårt att i praktiken avgöra hur lång tid en operation kräver, så konstaterarar jag ändå i kapitel två att alla har samma värsta fall. Men, även om de olika implementationerna har samma värsta fall, så menar jag att det sällan, om någonsin, kommer att se ut på detta vis i praktiken:

Värsta fallet i hash-tabellen utgår ifrån att det bara finns en hink eller att alla element hamnar i samma hink. Detta går enkelt att ändra på så att det blir lite bättre. Hamnar alla element jämnt fördelat i två hinkar istället för en så kommer tiderna att halveras. Hamnar dem i tre så kommer tiderna att förkortas med tre. Då tabellen blir ojämnt fördelad så kommer den i de flesta praktiska lägen fortfarande vara snabbare än den länkade listan och arrayen.

Det är just därför hash-tabellen var snabbare än både arrayen och den länkade listan.

6. Referenser

Inga referenser används i denna rapport, utan hänvisas istället till författaren.