Uppgift 6

Simon Moradbakti

HT 2024

Introduktion

I den här uppgiften skulle man hitta ett sätt att implementera en kö datastruktur. Antagligen har man någon gång stött på en kö i sitt liv, kanske när man handlar på ICA eller är fast i trafiken, en så kallad bilkö. Den som helt enkelt är först i sin kö kan få hjälp, köra eller handla sina varor först. Detta gör att köer fungerar efter systemet ”först in först ut” eller engelskans förkortning ”FIFO” eller First In First Out. En kö datastruktur följer detta konceptet fast i programmeringens värld.

Användning av en länkad lista

Kö struktur och lägga till element:

1. En pekare pekar på den första noden i vår lista, detta blir början på vår kö.
2. Om listan är tom skapa då en ny nod och sätt ”head” pekaren pekandes på den nya noden. Den kommer vara den första och enda noden i kön.
3. I en icke tom lista så kommer Head pekaren alltid peka på den första noden.
4. För att lägga till en ny nod i vår icke tomma lista måste vi travisera denna lista och hitta vår sista nod. Detta blir kodraden node.next. Lägg sedan till den nya noden till sista noden.

För att ta bort element enligt FIFO:

1. Head pekaren pekar på den första noden, som lades till först.
2. Returnera värdet från head noden.
3. Uppdatera sedan head pekaren så att den pekar på nästa nod i listan med hjälp av kodraden head.next.

Vi ska nu dra slutsatsen vad nackdelen med denna implementation blir. Att ta bort en nod i en kö är inget annat än en konstant operation, då vi behöver returnera värdet av vårt head och peka pekaren på vår nästa nod. När en ny nod läggs till så vet vi inte adressen till den sista noden, så vi måste leta genom hela listan. Först då kan man lägga till den nya noden i vår lista. Detta gör att kostanden blir ganska och dessutom linjär. Vilket vi ganska enkelt kan förbättra. Delar av den reviderade koden blev denna:

```
    public Queue() {
        head = null;
        tail = null;
    }
    public void enqueue(Integer item) {
        Node newNode = new Node(item);
        if (tail == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }

    }
    public Integer dequeue() {
        if (head == null) {
            return null;
        }
        Integer item = head.item;
        head = head.next;
        if
```

En förbättring

För att förbättra denna kod kan vi lägga till en så kallad tail-pekare. Man kan fråga sig varför en tail-pekare skulle förbättra våran gamla kod. Anledningarna till det är många. En tail-pekare gör att man inte inte längre behöver traversera hela listan för att hitta slutet på listan, då man vill lägga till ett element. En tail-pekare har en pekare på just slutet av listan, den så kallade "tailen" så det på en gång går att lägga till ett element. Detta i sin tur gör att det inte längre blir det linjära tidskomplexitetet O(n) för att lägga till element, beroende av längden på listan. Av samma anledningar blir det enklare att ta bort element från en lista. Med hjälp av en tail-pekare blir det även enklare att hantera tomma köer då vi kan sätta våran head och tail till null. Vi kan därför dra slutsatsen att en länkad lista med tail-pekare har O(1) som tidskomplexitet, vilket är konstant och därför snabbare! Delar av den implementerade koden ser ut som följande:

```
public void enqueue(Integer item) {
    Node newNode = new Node(item);
    if (tail == null) {
        head = newNode;
        tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
}
public Integer dequeue() {
        if (head == null) {
            return null;
        }
        Integer item = head.item;
        head = head.next;
        if (head == null) {
            tail = null;
        }
        return item;
    }
```

benchmarks

tid (ms)	Försök
49	1
90	2
55	3
53	4
76	5
81	6

Table 1: Resultat av exekvering

Detta gav ett snitt på ungefär 67. Vi ser att detta ger vårat tidskomplexitet O(1) och vi vet även att detta alltid är snabbare än ett linjärt O(n) förutsatt att det är lista som tar längre tid att analysera än vårat snitt. Därför kan slutsatsen vara att detta är en förbättring om man kör länkade listor med mycket data, men att det inte nödvändigtvis behöver vara det för länkade listor med lite data. Men detta är en ganska liten uppoffring så därför kan det ses som en stor förbättring.

Sammanfattning

I denna uppgift implementerades en typ av kö datastruktur med hjälp av en länkad lista enligt FIFO-principen (First In, First Out). Först användes endast en pekare en så kallad head för att peka på den första noden i listan. Det gjorde att hela listan behövdes traverseras för att lägga till element detta gav oss tidskomplexiteten O(n). För att förbättra detta infördes en tail-pekare, som gör direkt åtkomst till sista noden möjligt. Det gör att vi längre inte behöver traversera hela listan. Det gör det nya tidskomplexiteten till O(1). Benchmark-resultaten visade ett snitt på 67 ms, vilket visar att förbättrad prestanda har uppnåtts, speciellt då stora datamängder hanteras. En implementation med tail-pekare var alltså mer effektiv och snabbare.